Distributed Database System: Project Report

Zheng Zeng
School of Software, Tsinghua University, China
zengzheng17@gmail.com

Abstract

With the continuous development of computer technology and the Internet, the traditional single-node database system can no longer meet our needs. The emergence of distributed database systems has brought great improvements in performance, capacity, scalability and fault tolerance. In the distributed database course project, we need to design a distributed database system based on the existing database management system, hadoop file system and data caching tools to store and manage dozens or even hundreds of GB of user, article and read data. This article will start from the background of the problem, introduce the technologies used in the distributed database system, and introduce the specific design and implementation of the system.

1 Background and Motivation

Distributed database refers to the use of high-speed computer network to connect multiple physically dispersed data storage units to form a logically unified database. The basic idea of the distributed database is to store the data in the original centralized database in a decentralized manner on multiple data storage nodes connected through the network to obtain larger storage capacity and higher concurrent access. In recent years, with the rapid growth of data volume, distributed database technology has also developed rapidly. The traditional relational database has begun to develop from a centralized model to a distributed architecture. The relational distributed database retains the traditional database. Under the data model and basic characteristics, from centralized storage to distributed storage, from centralized computing to distributed computing. [1]

Distributed databases have three main characteristics: high scalability, high concurrency and high availability. That is to say, the distributed database must have high scalability and can dynamically add storage nodes to achieve linear expansion of storage capacity. At the same time, the distributed database must respond to the read/write requests of large-scale users in a timely manner, and can perform random read/write of massive data. In addition, the distributed database must provide a fault-tolerant mechanism, which can realize redundant backup of data and ensure high reliability of data and services.

Based on these characteristics, distributed databases can achieve the following three goals:

1. **High adaptability**: The units that use the database are often distributed organizationally and geographically distributed. The structure of the distributed database system conforms to the organizational structure of departmental distribution, allowing each department to store their commonly used data locally, input, query, maintain locally, and implement local control. Because the computer resources are close to the users, the communication cost can be reduced, the response speed can be improved, and it is more convenient and economical for these departments to use the database.
2. **High reliability**: Improving the reliability and availability of the system is the main goal of distributed databases. Distributing data across multiple sites and adding appropriate redundancy can provide better reliability, which is especially important for systems with

high reliability requirements, since failure of one site will not cause the entire system to collapse. Because users of the fault site can enter the system through other sites. Users in other sites can automatically select the access path by the system, avoid the faulty site, and use other data copies to perform operations without affecting the normal operation of the business.

3. **High utilization**: When several databases have been built in a large enterprise or department, in order to utilize mutual resources and develop global applications, a distributed database system must be developed. Build distributed systems from the bottom up. Although this method also needs to make some changes and reconstructions to the existing local database systems, compared with the integration of these databases to rebuild a centralized database, the distributed database is economical and organizational. is a better choice.

2 Existing Solutions

2.1 Hadoop Distributed File System

HDFS (Hadoop Distributed File System) is a distributed file system under Hadoop. It has the characteristics of high fault tolerance and high throughput, and can be deployed on low-cost hardware. It is one of the core components of Hadoop and exists as the lowest-level distributed storage service. The problem solved by HDFS is big data storage. They are storage systems that span multiple computers. [2] The structure of hdfs is shown in Figure 1.

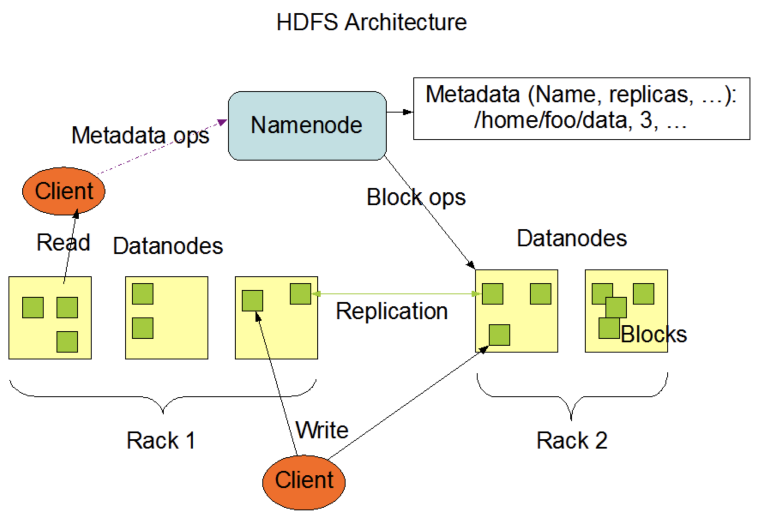


Figure 1: HDFS Architecture [2]

HDFS uses the Master and Slave structure to manage the cluster. Generally, an HDFS cluster consists of only one Namenode and a certain number of Datanodes. Namenode is the master node of the HDFS cluster, and Datanode is the slave node of the HDFS cluster. The two roles perform their respective duties and coordinate to complete the distributed file storage service.

HDFS has the following important features:

- Can store very large files. The "large file" here refers to files in the hundreds of MB, GB, or even terabytes. Large files will be divided into multiple blocks for storage. The default block size is 128MB. Each block will store multiple copies on multiple datanodes, the default is 3 copies.
- Supports streaming data access.
- Supports running on ordinary cheap servers. One of the design concepts of HDFS is to make it run on ordinary hardware. Even if the hardware fails, fault tolerance strategies can be used to ensure high data availability.

2.2 MongoDB Sharding Cluster

Sharding is the method MongoDB uses to split large collections across different servers (or a cluster). While sharding has its origins in relational database partitioning, MongoDB sharding is another story entirely. Compared with the MySQL partitioning scheme, the biggest difference between MongoDB is that it can do almost everything automatically. As long as you tell MongoDB to allocate data, it can automatically maintain the balance of data between different servers.

Database applications with high data volume and throughput will put a lot of pressure on the performance of a single machine. A large amount of queries will exhaust the CPU of a single machine, and a large amount of data will put a lot of pressure on the storage of a single machine, which will eventually exhaust the memory of the system. Instead, the pressure is shifted to disk IO.

To solve these problems, there are two basic methods: vertical expansion and horizontal expansion. Vertical scaling is adding more CPU and storage resources to expand capacity. Horizontal scaling, on the other hand, distributes the dataset across multiple servers. Horizontal scaling is sharding.

3 Problem Definition

In this project, we need to build a distributed database to monitor dozens or even hundreds of GB of data. Data to be managed and processed include structured data (5 relational tables) and unstructured data (text, images, and video). Their inter-relations are illustrated in Figure 2.

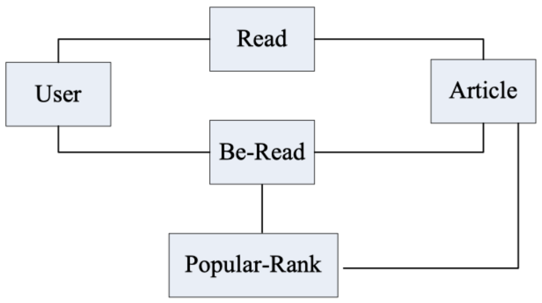


Figure 2: E-R Diagram

Based on these data, we will use some existing distributed solutions to implement a data center in a distributed scenario. Specifically, this data center should contain the following capabilities:

1. Bulk data loading with data partitioning and replica consideration.
2. Efficient execution of data insert, update, and queries.
3. Monitoring the running status of DBMS servers, including its managed data (amount and location), workload, etc.
4. Hot / Cold Standby DBMSs for fault tolerance.
5. Expansion at the DBMS-level allowing a new DBMS server to join.
6. Dropping a DBMS server at will.

4 Proposed Solutions

According to the definition of the problem, we will build our data center based on the existing distributed solutions introduced in section Existing Solutions. Finally, the overall architecture of our system is shown in figure 4:

As shown in the figure, in the application part of the whole system, I used Vue as the front-end framework and flask as the structure of the back-end framework. In this way, users can access our database system directly through the web page. Compared with ordinary desktop clients, the front-end and back-end implementations are easier to use and more portable.

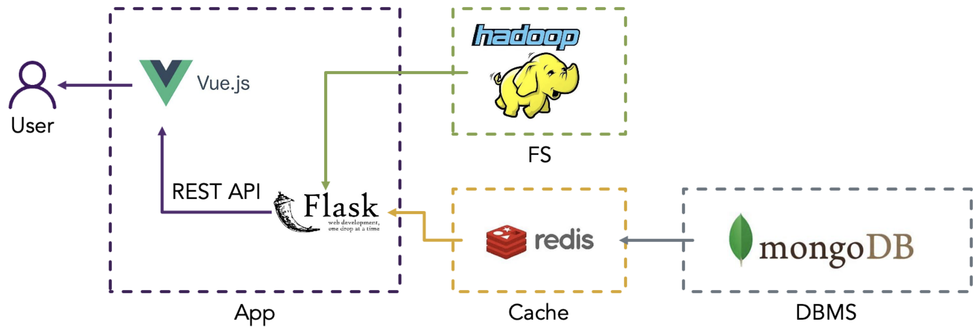


Figure 3: Overall Architecture

In the distributed database management system connected to the backend, it includes two parts: the file system and the database (data storage). Among them, the file system uses the aforementioned Hadoop Distributed File System, and the database (data storage) uses MongoDB Sharding Cluster. In order to speed up the data response and reduce the load of database queries, we use Redis as a cache between the backend and MongoBD. When the target data of the query is in the cache, it is directly read from the cache. If it is not in the cache, it will be read in MongoDB and cached in redis, which is convenient for next use. When building the system, we used multiple docker containers as multiple nodes of MongDB or HDFS to construct a distributed cluster. Next, we will introduce the specific implementation of HDFS, Redis, MongoDB and applications respectively

4.1 HDFS

An HDFS cluster consists of a namenode and a certain number of datanodes. The namenode is a central server responsible for managing the file system namespace and client access to files. The datanode in the cluster is generally one node, responsible for managing the storage on the node where it is located. Internally, a file is actually divided into one or more data blocks, which are stored on a set of datanodes. namenode performs filesystem namespace operations, such as opening, closing, and renaming files or directories. It is also responsible for determining the mapping of data blocks to specific datanode nodes. datanodes are responsible for handling read and write requests from file system clients. The creation, deletion and replication of data blocks are performed under the unified scheduling of namenode. Each file is stored as a series of data blocks, and for fault tolerance, all data blocks of the file will have copies.

Here we use the structure of a master and 2 slaves, which means that there is a total of one namenode and two datanodes.

In order to allow them to access each other's data, we use the Docker bridge to build a small network. Through this small network, master and slave can access each other's data. In addition, docker for macOS does not automatically build a bridge named docker0 to the network like Docker for Linux when starting a container, so the host cannot directly access the files of each node in this small network. To solve this problem, I created a VPN from the host to the small network, so that the files in HDFS can be accessed normally.

4.2 Redis

Redis uses memory as the data storage medium, so the efficiency of reading and writing data is extremely high, far exceeding that of databases. Taking setting and getting a 256-byte string as an example, its read speed can be as high as 110,000 times/s, and its write speed can be as high as 81,000 times/s. The data stored in Redis is persistent, and the data will not be lost after a power failure or restart. Because the storage of Redis is divided into three parts: memory storage, disk storage and log file, after restarting, Redis can reload data from disk into memory, which can be configured through configuration files. Because of this, Redis can achieve persistence change.

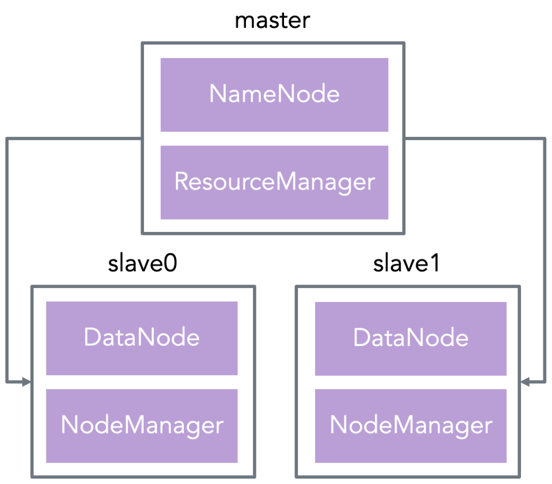


Figure 4: Overall Architecture

When using it, we use the type of data and its id as its key value, and use the serialized json string of its content as the value value. We know that the memory size of redis is limited, so when the amount of cached data exceeds a certain range, the cached data needs to be replaced. Common replacement strategies include [3]:

- voltile-lru: Pick the least recently used data from the dataset with expiration time to evict.
- volatile-ttl: Select the data to be expired from the dataset that has expiration time.
- volatile-random: arbitrarily select data eviction from the dataset with expiration time.
- allkeys-lru: pick least recently used data from dataset for elimination.
- allkeys-random: Randomly select data from the dataset for elimination.
- no-enviction: prohibit eviction of data.

Our replacement strategy is allkeys-lru. Without setting the expiration time, the lru strategy will tend to retain data that is frequently used, thereby increasing the probability of reading the cache when data is read.

4.3 MongoDB

When the amount of data is very large, MongoDB allows us to store the data on multiple machines in a sharded manner to decompose the server's pressure on CPU, memory and disk IO. [4]

A MongoDB sharded cluster consists of 3 parts:

- Shard Server: Each shard stores a certain amount of data, and shards can be configured as replica sets.
- Router Server: Acts as a router between clients and the cluster, forwarding requests to shards.
- Config Server: Stores cluster configuration and metadata, and must be configured as a replica set to ensure high availability.

MongoDB uses Shard Key to decide which shard to store and search for a piece of data. The shard key can be specified manually, but it must be an immutable field and this field exists in all documents (Document, that is, a single piece of data stored by MongoDB). Each There can only be one (composite) shard key on a sharded collection. Once sharded, the shard key cannot be modified. The choice of shard key is likely to become a bottleneck in determining performance. We usually choose those fields with indexes as shard keys.

MongoDB splits sharded data into chunks, each of which contains data within a certain range of shard keys. In order to distribute blocks evenly across shards, MongoDB has a backend balancer to migrate blocks between shards. Using Shard Key, we can easily implement data migration and management.

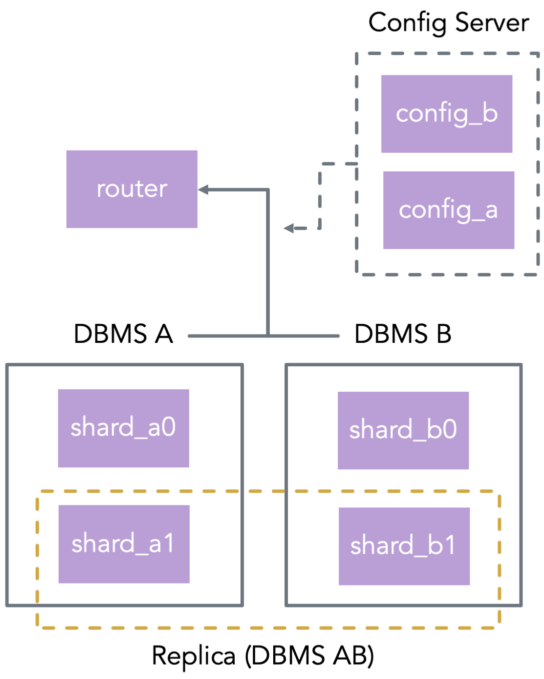
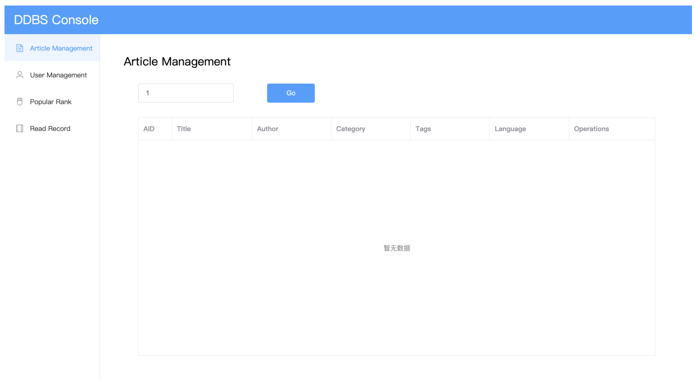


Figure 5: MongoDB

In the cluster I designed, there are 4 shard servers, 2 config servers and a router server (Figure 5). Among them, shard1 and shard2 form DBMS A, which stores Beijing's data, and shard3 and shard4 form DBMS B, which stores Hong Kong's data. In addition, shard2 and shard3 also form DBMS AB to meet the requirements of data backup and fragmentation.

4.4 Application

We build user-friendly applications based on the front-end and back-end frameworks of Vue and Flask. REST-ful APIs were developed for the addition, deletion, modification and inspection of user, article, read, be-read and popular-rank data respectively. Then use Ajax to connect the front and back ends, so that users can easily operate data and manage the database in the user interface of Figure 6.



Article Management

1

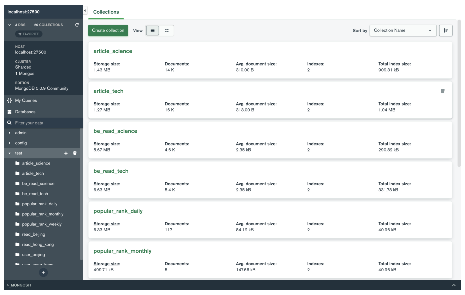
AID	Title	Author	Category	Tags	Language	Operations

暂无数据

Figure 6: User Interface

5 Solution Evaluation

To verify that our database works correctly, and to measure database performance, we use MongoDB Compass [5] (Figure 7(a)) and Hadoop Web Page (Figure 7(b)) to monitor our database and file system.



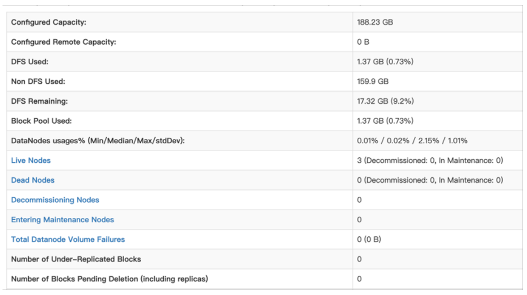
HOST
localhost:27500

CLUSTER
Sharded
1 Mongos

EDITION
MongoDB 5.0.9 Community

	Storage size:	Documents:	Avg. document size:	Indexes:	Total index size:
article_science	1.43 MB	14 K	310.00 B	2	909.31 kB
article_tech	1.27 MB	16 K	313.00 B	2	1.04 MB
be_read_science	5.67 MB	4.6 K	2.35 kB	2	290.82 kB
be_read_tech	6.63 MB	5.4 K	2.35 kB	2	331.78 kB
popular_rank_daily	6.33 MB	117	84.12 kB	2	40.96 kB
popular_rank_monthly	499.71 kB	5	147.66 kB	2	40.96 kB

(a) MongoDB Compass



Configured Capacity:	188.23 GB
Configured Remote Capacity:	0 B
DFS Used:	1.37 GB (0.73%)
Non DFS Used:	159.9 GB
DFS Remaining:	17.32 GB (9.2%)
Block Pool Used:	1.37 GB (0.73%)
DataNodes usages% (Min/Median/Max/stdDev):	0.01% / 0.02% / 2.15% / 1.01%
Live Nodes	3 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion (including replicas)	0

(b) Hadoop Web Page

Figure 7: Monitor

6 Conclusion

In this paper, we introduce the implementation of a simple distributed database system. Based on the existing distributed data solutions, combined with the characteristics of target data, an available and effective distributed database management system is constructed.

7 Future Work

In implementing this distributed database, I realized that there are still some work that can be optimized:

- Convenient data migration
- Improve query efficiency
- More flexible file management

References

[1] Alexandros Labrinidis and H. V. Jagadish. “Challenges and Opportunities with Big Data”. In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), pp. 2032–2033. ISSN: 2150-8097. DOI: `10.14778/2367502.2367572`. URL: `https://doi.org/10.14778/2367502.2367572`.

[2] Dhruba Borthakur. *HDFS Architecture Guide*. `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html` Accessed June 10, 2022.

[3] Redis Ltd. *Redis Document*. `https://redis.io/docs/clients/` Accessed June 10, 2022.

[4] MongoDB Inc. *MongoDB Document*. `https://www.mongodb.com/docs/` Accessed June 10, 2022.

[5] MongoDB Inc. *MongoDB Compass*. `https://www.mongodb.com/products/compass/` Accessed June 10, 2022.