Distributed Database Systems Course Final Project

Daniel Zhang-Li*
zlnn23@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

ABSTRACT

Due to the release of YMTC's SSD, storage has become much cheaper in the market, potentially bringing a greater demand for Distributed Database Systems(DDBS) individual level. In this work, we focus on the individual user's need for easier replicating their data with simplicity in design and proposed a DDBS framework philosophy for individual users. We chose the data visualization for a blog website as our idea's demonstration and implemented a DDBS with MongoDB and FastDFS. Furthermore, despite the simplicity in design, we added various fault tolerance methods on top our designed system.

KEYWORDS

Distributed Systems, Fault Tolerance, Databases

ACM Reference Format:
Daniel Zhang-Li. 2018. Distributed Database Systems Course Final Project. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 MOTIVATION

The advent of Distributed Database Systems (DDBS), focused on managing data across multiple nodes, has historically been driven by industrial-scale applications. Nevertheless, the emergence of cost-effective Solid-State Drives (SSDs) by companies like YMTC has transformed this landscape, making the deployment of distributed databases feasible even at the individual user level. This shift presents a unique opportunity to explore DDBS tailored to personal use, a domain traditionally overlooked in favor of large-scale enterprise solutions.

Personal users typically manage smaller data volumes and have fewer requests, necessitating a system design that prioritizes privacy, ease of management, and adaptability to specific user requirements. Our proposed Nginx-Centric architecture addresses these needs. In this design, all components, except Nginx itself, operate within a private network. Most services communicate through Docker's bridged network or are routed via Nginx. This approach simplifies network configuration and routing, and crucially, it enhances data security by reducing direct exposure to the web.

Moreover, this project, undertaken as a capstone for the DDBS course, aims to provide a practical and in-depth understanding of DDBS application in contemporary settings. Emphasizing the acquisition of fundamental skills and their application to real-world scenarios is pivotal for students. To this end, we[1] have selected industry-standard frameworks for implementation. However, prioritizing code clarity and simplicity, as suggested by Gabriel (1991)[6], we have opted for a straightforward Python framework over more complex Java alternatives.

In summary, this project makes several key contributions:

- **Nginx-Centric Design:** We introduce a Nginx-Centric approach where Nginx primarily manages access.
- **Ready-to-Use Application:** Our design is implemented in a management application for blog site analytics. We also provide our code as a ready-to-use tool, available at https://github.com/Danielznn16/DDBS-Final.
- **Enhanced Fault Tolerance:** Focusing on data replication for fault tolerance, we have incorporated all optional features outlined in the project manual. Our experiments demonstrate that our implementation meets the required standards effectively.

2 EXISTING SOLUTIONS

This section categorizes classic distributed storage systems into two primary types: database management systems (DBMS) and distributed file systems (DFS). We will explore prominent examples in each category, focusing on their unique approaches and features.

2.1 DBMS

MySQL. MySQL's approach to distributed databases is encapsulated in its NDB Cluster. This system ensures linear scalability and high availability, crucial for mission-critical applications. It utilizes a shared-nothing architecture, dynamically partitioning tables across nodes, thus enabling horizontal scaling. Key features include in-memory real-time access, transactional consistency, and support for multiple replicas, facilitating resilience and performance across distributed datasets[5]. However, MySQL is interacted via SQL commands, making it hard for individual users to master.

MongoDB. MongoDB's sharding mechanism distributes data across multiple machines, ideal for handling large data sets and high throughput demands. Sharded clusters in MongoDB consist of several components, such as shards (each a subset of data), mongos query routers, and config servers for metadata. Sharding occurs at the collection level with shard keys dictating data distribution.

*This project is done solo, so there is only one author.
Student ID: 2023380016

[1]The term "we" is used to refer to the author, as this project, originally designed for group participation, is undertaken individually.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

This setup allows for targeted queries and scalable read/write workloads. However, MongoDB's sharding introduces complexity and demands careful planning and maintenance[4]. MongoDB's sharding approach is further illustrated in Figure 1, which describes the interaction of components within a sharded cluster.

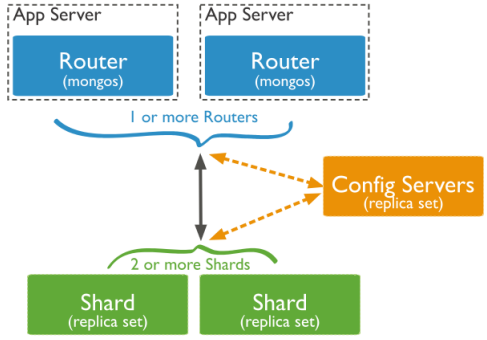

Figure 1: Interaction of components in a MongoDB sharded cluster.

2.2 Distributed File Systems (DFS)

In the domain of DFS, FastDFS, Hadoop Distributed File System (HDFS), and MinIO are notable examples. Each system has distinct operational characteristics, with FastDFS being recognized for its lightweight and user-friendly approach compared to HDFS and MinIO.

FastDFS. Developed in C, FastDFS is a lightweight and high-performance distributed file system. It is particularly efficient for large-capacity file storage and high concurrent access, often used by medium and large websites handling files between 4KB and 500MB. The system is structured around a Tracker Server for file access scheduling and load balancing, and a Storage Server responsible for file storage and synchronization. FastDFS's architecture supports dynamic addition of storage capacity and efficient file management to prevent directory overload, making it a scalable and efficient solution[1].

Hadoop Distributed File System (HDFS). HDFS, part of the Apache Hadoop project, is designed for high fault tolerance and high throughput access to large data sets on commodity hardware. It employs a master/slave architecture with a NameNode managing the file system namespace and DataNodes handling storage. HDFS stores files as a sequence of blocks, replicated across DataNodes. Its rack-aware replica placement policy is designed for improved data reliability and availability, but requires careful planning and management, making HDFS more complex and less straightforward than FastDFS[2].

MinIO. MinIO is an open-source distributed object storage server, offering S3 storage functionality and suited for unstructured data like photos and videos. It supports multiple storage backends, including local disk and cloud storage, and uses erasure coding for data protection. MinIO can be deployed in a distributed mode for optimal storage device utilization and high availability. While it offers features like S3 API compatibility and data security, the setup and management of MinIO, especially in distributed mode, are more complex compared to FastDFS[3].

Comparison. FastDFS stands out for its straightforward approach and lightweight architecture, making it more user-friendly than HDFS and MinIO. Its direct handling of file uploads to data nodes and simpler scalability appeals to users seeking an efficient DFS without the complexity of HDFS or MinIO.

3 PROBLEM DEFINITION

The objective of this project is to develop a distributed data center capable of efficiently managing a mix of structured and unstructured data. This data includes five relational tables with intricate inter-relations (see Figure 2 for details) and diverse unstructured data comprising text, images, and videos related to article contents.

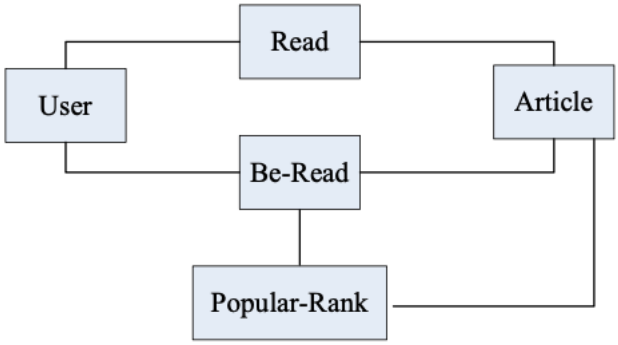

Figure 2: Structured data tables inter-relations.

3.1 Data Management

The system must effectively handle the following structured data tables:

(1) User Table: Stores user information, fragmented based on region.
(2) Article Table: Contains article information, fragmented by article category.
(3) Read Table: Manages user article reading data, with fragmentation aligned with the User table.
(4) Be-Read Table: Tracks article reading statistics, fragmented based on the Article table and subject to duplication.
(5) Popular-Rank Table: Lists popular articles, fragmented according to temporal granularity.

In addition, the system should ensure that unstructured data related to articles is stored and accessible in a distributed manner(DFS).

3.2 Functional Requirements

The implementation of the data center should encompass the following key functionalities:

- Bulk data loading, considering data partitioning and replication strategies.
- Efficient execution of data insertion, updates, and various query operations.
- Dynamic population of the Be-Read table with new records.

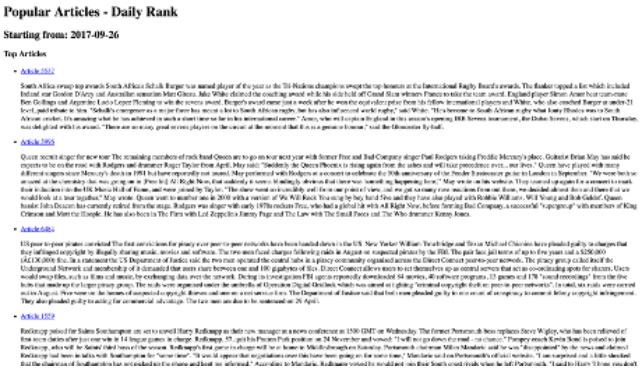

Figure 3: A demonstration page of the application launched with python.

- Advanced querying capabilities, including extracting top-rated articles along with their detailed contents from multiple data sources.
- Continuous monitoring of DBMS servers, focusing on managed data, workload, and operational status.

3.3 Optional Advanced Functionalities

Additionally, the project may explore optional advanced functionalities to enhance system robustness and scalability:

- Implementing hot/cold standby DBMS servers for improved fault tolerance.
- Enabling the system to dynamically add or remove DBMS servers.
- Facilitating data migration across different data centers.

4 METHODOLOGY

Adopting the approach of established methodologies [7], our system's implementation is structured into three principal segments: Distributed File System (DFS) for unstructured storage, Database Management System (DBMS) for structured data, and Controllers for overarching management and service facilitation. Given the personal-level application of our system, we integrate a firewall to enhance privacy measures. As depicted in Figure 4, the system architecture is categorized into four key components: the Firewall (administered via Docker), Controlling Units, DFS, and DBMS. The subsequent subsections will delve into each of these components, elucidating their initialization processes and interactivity mechanisms.

4.1 DBMS

The foundational architecture of our DBMS focuses on sharding data and replicating storage systems. For the sharding process, we developed a Python script that segments the input data into portions suitable for storage in individual shards (DB1 and DB2). Concerning replication, we created two replicas for each data shard. This setup results in a total of four MongoDB instances - two shards, each with two replicas. These instances are initiated using docker-compose, and for each shard, we established a buffer directory. These directories are linked to their respective MongoDB instances through the 'volume' configuration in Docker. By directing the shard-specific data to these buffers, we enable direct access by the MongoDB instances, thereby reducing the need for redundant data copying and minimizing IO costs.

Data loading into MongoDB is facilitated using the mongoimport command. With identical collection names across shards and replicas, the crucial aspect of data fragmentation is managed by our processing script. We divided this script into two distinct components: the first script reads and allocates the input data into the designated buffer files, while the second manages the execution of mongoimport across the MongoDB instances. This bifurcation is critical as it addresses issues related to file write completion before data import – ensuring that each buffer file is fully prepared before its content is uploaded to the respective MongoDB instance.

It is important to highlight our choice of MongoDB for this implementation, which we also recommend to users. MongoDB's user-friendly interface, particularly its compatibility with JSON-like structures and straightforward query language, significantly enhances usability and accessibility for a broader range of users.

4.2 DFS

In the realm of unstructured data distribution, where specific routing rules are not imperative, we leverage Distributed File System (DFS) services for handling replication tasks. The initial step in enabling replication involves categorizing storage nodes into distinct groups. This is accomplished by modifying the configuration files. Subsequently, FastDFS nodes are initialized using docker-compose. It is crucial to note that, unlike MongoDB, the startup sequence of tracker and storage nodes in FastDFS is of significant importance. We ensure that storage nodes commence operations only after the tracker nodes have been successfully started.

For the process of uploading files, we employ a strategy where the volume containing the files for upload is mapped directly to a designated storage node. This approach ensures that the storage node remains insulated from external internet access, yet retains the capability to receive uploads internally. To facilitate this, a shell script is executed within the storage node, which iterates over all files, uploading them sequentially. The subsequent paths of these stored files are meticulously recorded in a mapping file, formatted as $file_name \rightarrow path_in_dfs$ for each entry.

Upon the completion of the file upload process, the master node transfers the mapping file to the application container. This container, in turn, is responsible for uploading these mappings into MongoDB. This method presents a more advantageous approach compared to HDFS or Minio, primarily because the core mappings are directly accessible in MongoDB. This direct access significantly simplifies management for individual users, offering a more streamlined and user-friendly experience.

4.3 Controlling Units

Our system is managed and operated by several controlling units, each with a distinct role integral to the overall functionality. We will detail each unit as follows:

Host Master. The Host Master serves as the central controlling unit of the system, overseeing its initiation and termination processes. In a Kubernetes (k8s) environment, this would parallel the master

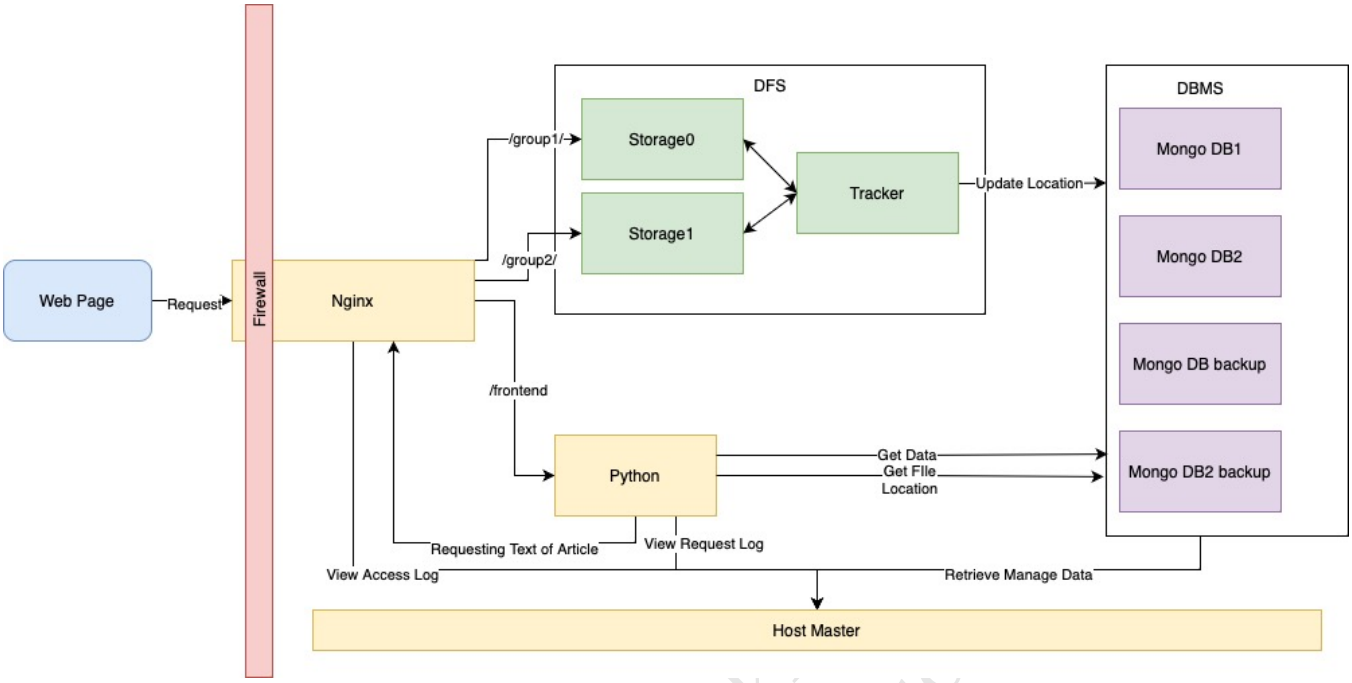

Figure 4: The proposed framework. Services other than the Nginx are behind the Firewall. Modules such as Nginx, Python(App Backend), and Host Master are controlling nodes; Modules labeled with Storage and Tracker are for DFS; MongoDB related nodes are for DBMS.

node responsible for launching containers. In our simulation of distributed launching on a single machine, this role is fulfilled by the host machine managing the Docker environment. Our design principle ensures that the Host Master does not interact directly with the actual stored data or access the ports of any modules except for Nginx. The port mappings in our provided code are primarily for observation of the system's operation. Ideally, in real-world applications, all port mappings or exposures should be minimized to enhance security. The Host Master's data interaction is limited to copying and moving intermediate results during system startup. Most data interactions, especially those involving data uploading, are confined within the serving modules like the Python container or the FastDFS storage node.

We use Mongo Compass as the monitor of the system. As Mongo Compass supports directly viewing the data, it can also serves are the monitor for the stored files in DFS.

Nginx. The Nginx container is designed to be the sole module with direct external network interactions. By channeling service requests through a specific prefix, we ensure that all external requests are managed through the single exposed port of Nginx. This design simplifies the process of associating the service with a public IP address, as each domain can only be directed to a specific port, enhancing both ease of use and security.

Python. The Python container functions as the primary serving unit. All data communications are initiated and processed within this container, bypassing the host. This approach not only facilitates easier management, since all operations occur within the Docker environment, but also heightens security. Direct requests to the host are avoided, significantly reducing the risk of service attacks impacting the host or other services deployed on it. This distinction is particularly crucial for personal users, who often utilize a single machine to deploy multiple services. By isolating service interactions within the Python container, we provide a more secure and manageable environment for individual users with diverse service requirements. A demonstration page is shown in Figure 3

4.4 Firewall Configuration

Given that our system is operating on a single machine, managing the firewall settings becomes a relatively straightforward task. We achieve this by meticulously controlling the exposure of ports. This is primarily managed through the docker-compose configuration, where only a select list of ports are designated to be exposed. All services within our system are deployed in containers that are part of a bridged network. This setup ensures that both the host and any external machines can only access a specific port if it is explicitly exposed in two key areas: the host machine's firewall settings and the docker-compose configuration.

This dual-layer approach to port exposure provides an additional security barrier. It ensures that unauthorized access is effectively prevented unless there is a deliberate configuration allowing it. By tightly controlling port exposure in this manner, we enhance the overall security posture of the system, especially important in scenarios where multiple services are running on a single host machine.

5 EVALUATION

This section presents an evaluation of the experimental features implemented in our project, demonstrating the system's ability to meet the requirements specified in the project guidelines.

5.1 Bulk Loading

The bulk loading process, as outlined in previous sections, is facilitated by scripting. The processing of input structured data and the creation of input files for uploading were completed in 13 seconds. An additional 5-second delay was incorporated for observational purposes, leading to a total of 22 seconds to load data into the MongoDB containers. However, loading files into FastDFS was more time-consuming, taking 1326 seconds. Remarkably, the subsequent process of writing the mapping to all MongoDB instances was accomplished in just 2 seconds. This identified bottleneck could potentially be alleviated by adding more storage nodes to FastDFS and distributing the upload process across multiple nodes.

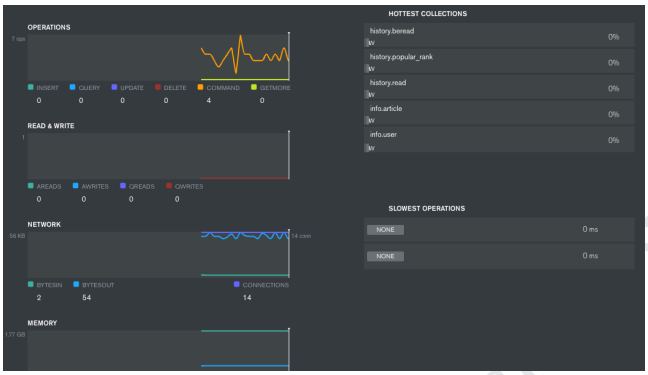

Figure 5: Mongo Compass monitoring the system.

5.2 Beread and Popular Rank

The system uses 'upsert' operations (insertion or update) for many tasks. We measured the time taken for generating Beread and Popular-Rank data, which involves querying other tables and performing insert (on first occurrence) or update operations. This approach achieves a fault tolerance level of AT_LEAST_ONCE[2]. The upper bound runtime observed was 53 seconds for Beread and 15 seconds for Popular Rank. The difference in time can be attributed to Beread's higher I/O demand and the use of aggregation for generating Popular Rank, which is a more efficient method compared to processing all relevant records on the host.

Additional implementation includes the article, popular-rank, and user pages in the application to assess real-time latency. The article page displays information and content for an article, involving single table access. The popular-rank page retrieves specific rank results and provides abstracts of articles with links for further reading. The user page showcases the reader's history and fetches abstracts from the article table, similar to the popular-rank page. While the article and popular-rank pages respond almost instantaneously, the user page typically requires a one-second wait due to its more complex data interactions and HTML rendering.

5.3 Monitoring

As depicted in Figure 5, Mongo Compass is utilized for system monitoring. For ease of observation, the port mappings for the MongoDB containers were set to 27001 → 27004 in our setup. However, for enhanced security, it is advisable to establish user accounts for MongoDB with specific permissions, restricting access to monitoring and certain privileges rather than allowing full write access. This strategy would significantly improve the system's security posture.

5.4 Optional Features

We first drop the data in the first replica for DB1 and we observe all systems are still functioning correctly, this is because we set the retry to seek all replicas before throwing an unfound error. This shows we have **hot-standby** and **Dropping a DBMS server at will**. We then restore this node from its replica. We leverage on the buffer directory shared by all DB1 replicas, to transfer the data from the other replicas and then restoring it. This shows that we can **Migrate from one Data storage to Others**. As DB1's first replica was dropped, this also means that we can **Allow new DBMS to Join**. To further expand to more nodes and data centers, simply adding more nodes in the docker-compose and migrate data is supported by our design. However, we do need to modify the python deployment for service to recognize more DBMS, which can also support automatic reloading after modifying the corresponding files with quickly restarting the python container, or if we further integrate services such as uvicorn for auto reloading on file change.

5.5 Optional Features Evaluation

In this subsection, we rigorously test various optional features to evaluate the robustness and flexibility of our system.

Initially, we simulated a failure scenario by dropping the data in the first replica of DB1. The system continued to function correctly, demonstrating the effectiveness of our failover strategy. This strategy involves attempting retries on all replicas before signaling an error due to data unavailability. This successful handling of the simulated failure illustrates the system's capability for **hot-standby** and **dynamic management of DBMS servers**.

Subsequently, we executed the recovery of the dropped DB1 replica. Leveraging the shared buffer directory used by all replicas of DB1, we were able to transfer data from the remaining replicas to restore the dropped one. This process effectively showcases our system's ability to **migrate data between storage units**. Moreover, the temporary absence and restoration of DB1's first replica highlight our system's capacity to **incorporate new DBMS nodes** seamlessly.

Our design inherently supports scalability, such as expanding to more nodes and data centers. This can be achieved by simply adding more nodes in the docker-compose file and migrating data accordingly. However, it is noteworthy that expanding the DBMS would require modifications to the Python deployment. This is necessary to enable the service layer to recognize additional DBMS nodes. To facilitate this, our system can support automatic reloading

[2]Repeated data insertions are treated as updates, while new data triggers an upsert operation, inserting only if there are no matching existing records.

by modifying the relevant configuration files and swiftly restarting the Python container. For more dynamic environments, integrating services like uvicorn could provide auto-reloading capabilities upon file changes, further enhancing the system's adaptability.

6 CONCLUSION

This report has presented a comprehensive exploration and implementation of a Distributed Database System (DDBS) tailored for individual user scenarios. Through a detailed examination of the system's architecture, methodology, and evaluation of its performance and capabilities, we have demonstrated a solution that addresses the unique challenges and requirements of personal users in managing distributed data.

Key components of the system, including the Host Master, Nginx, and Python containers, were designed with specific roles to ensure efficient management, security, and data processing. The system's architecture not only facilitates ease of use but also ensures robust security protocols, critical for individual users operating on a single machine. Our approach to firewall management and controlled port exposure further strengthens the system's security.

The evaluation of the system's performance in bulk loading, handling Beread and Popular Rank operations, and system monitoring showcases its efficiency and reliability. The use of Mongo Compass for monitoring exemplifies the system's compatibility with standard industry tools, enhancing its practicality.

Moreover, the system's design and implementation demonstrate essential features like hot-standby, dynamic DBMS server management, data migration, and scalability. These attributes highlight the system's flexibility and resilience, crucial for adapting to various user needs and potential expansion scenarios.

In conclusion, this project successfully integrates theoretical knowledge from the DDBS course with practical application, resulting in a functional and user-friendly system. It not only fulfills the academic requirements of the course project but also provides a valuable framework that can be adapted and expanded for real-world applications. The experience and insights gained from this project lay a solid foundation for future endeavors in the field of distributed database systems, particularly in the context of personal user environments.

6.1 Future Work

Looking forward, there are several avenues for further development and enhancement of this DDBS project. One primary area of focus could be the integration of advanced machine learning algorithms for predictive analytics and automated management of data. This could significantly improve the efficiency of data handling and enable predictive maintenance, reducing downtime and optimizing system performance.

Moreover, conducting comprehensive stress testing under diverse scenarios and workloads would provide deeper insights into the system's resilience and robustness. This would also help in identifying and addressing any potential bottlenecks or vulnerabilities.

ACKNOWLEDGMENTS

Special thanks to Zhenfang Lu(Fullstack engineer from ZhiPu.AI) for his advice on using FastDFS. I would also like to thank the TAs and Prof. Feng for kindness support of extending the homework due during I caught a serious fever during the semaster. Also special thanks for Prof. Feng for allowing us to do this project in solo.

REFERENCES

[1] [n. d.]. FastDFS distributed file system. https://github.com/fastdfs. Accessed: 2024-01-08.
[2] [n. d.]. HDFS Architecture Guide. https://hadoop.apache.org/docs/current/. Accessed: 2024-01-08.
[3] [n. d.]. Minio Distributed Object Storage Architecture and Performance. https://github.com/minio/docs. Accessed: 2024-01-08.
[4] [n. d.]. MongoDB Manual - Sharding. https://www.mongodb.com/docs/manual/sharding/. Accessed: 2024-01-08.
[5] [n. d.]. MySQL NDB Cluster. https://www.mysql.com/products/cluster/. Accessed: 2024-01-08.
[6] Richard Gabriel. 1991. The rise of worse is better. *Lisp: Good News, Bad News, How to Win Big* 2, 5 (1991).
[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 29–43.

Received 08 Jan 2024