# DDBS2024 Project Report

Rosalie Butte*
btlsl24@mails.tsinghua.edu.cn

Lin Nianyi*
linny24@mails.tsinghua.edu.cn

## ABSTRACT

The rapid growth of digital reading platforms has increased the demand for personalized recommendations and real-time data analysis. To address the challenges of managing massive and heterogeneous data, this paper presents a distributed database system-based architecture for reading applications. The system integrates MongoDB for structured data, FastDFS for unstructured data, and a Flask-based backend with a user-friendly frontend for data visualization. Key features include efficient data storage and querying, scalable and fault-tolerant architecture, and insightful visualization tools. Performance evaluation confirms the system's reliability and efficiency, providing a robust foundation for enhancing user experience and future research in distributed data management.

## KEYWORDS

DDBS

**ACM Reference Format:**
Rosalie Butte and Lin Nianyi. 2024. DDBS2024 Project Report. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

With the rapid growth of digital reading platforms, there is an increasing demand for personalized content recommendations and real-time data analysis. Modern reading software is expected to provide a vast range of articles and, at the same time, handle large volumes of user interaction data, such as user reading history, likes and comments, and article popularity metrics. These data points are crucial for optimizing recommendation systems, enhancing user experience, and offering valuable feedback to content creators. However, the storage and processing of such large-scale data present significant challenges, particularly when dealing with various data types and high-concurrency scenarios. Traditional single-node database systems struggle to meet the needs of large-scale data management and real-time analytics, making distributed database systems a viable and efficient solution.

Distributed Database Systems (DDBS) improve scalability, fault tolerance, and query efficiency by distributing data across multiple nodes. In recent years, with the advancement of cloud computing and big data technologies, distributed database systems have shown great advantages in handling large-scale data. For reading software, a management platform based on a distributed database system can provide stable and fast responses under high-concurrency scenarios and support real-time updates and analysis of user behavior data, which is essential for personalized recommendations and data visualization.

In this paper, we propose an architecture based on a distributed database system to manage and visualize various data types from reading software, including user information, article information, user reading history, article interaction history, and article popularity metrics. The core contribution of our system are as follows:

(1) Efficient Data Storage and Querying: By leveraging a distributed database system, we aim to support efficient data storage and querying, enabling real-time analysis of large volumes of user behavior data.
(2) Scalable System Architecture: The system is designed with high scalability, capable of handling rapid growth in user numbers and data volumes while maintaining stability in high-concurrency scenarios.
(3) Data Visualization and Analysis: The system offers comprehensive data visualization features, aiding platform operators and content creators in understanding user behavior, analyzing article popularity, and optimizing recommendation algorithms.

This study provides theoretical support and practical insights for building efficient and scalable data management platforms for reading software. Our research not only helps to improve the performance and user experience of reading applications but also offers new approaches for distributed data management and analysis in other fields.

The structure of this paper is as follows: Section 2 provides an overview of related work and defines the core problems, including data processing and functional requirements. Section 3 presents the design and implementation of the proposed system, covering components such as the data file system and the database management system. It discusses in detail the storage schemes for different types of data, query optimization strategies, and the implementation of mechanisms like load balancing and fault recovery. Section 4 evaluates the performance and scalability of the system through experimental analysis and real-world application cases. Finally, Section 5 concludes the paper and suggests directions for future research.

## 2 PRELIMINARIES

### 2.1 Related Works

There are many existing distributed database systems. In this section, we will have a look at a few prominent examples and their approaches.

*2.1.1 MySQL.* The distributed database approach from MySQL is the MySQL NDB Cluster [2]. It uses a shared nothing architecture and auto-partitionning to facilitate horizontal scalability and

*Equal Contribution


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

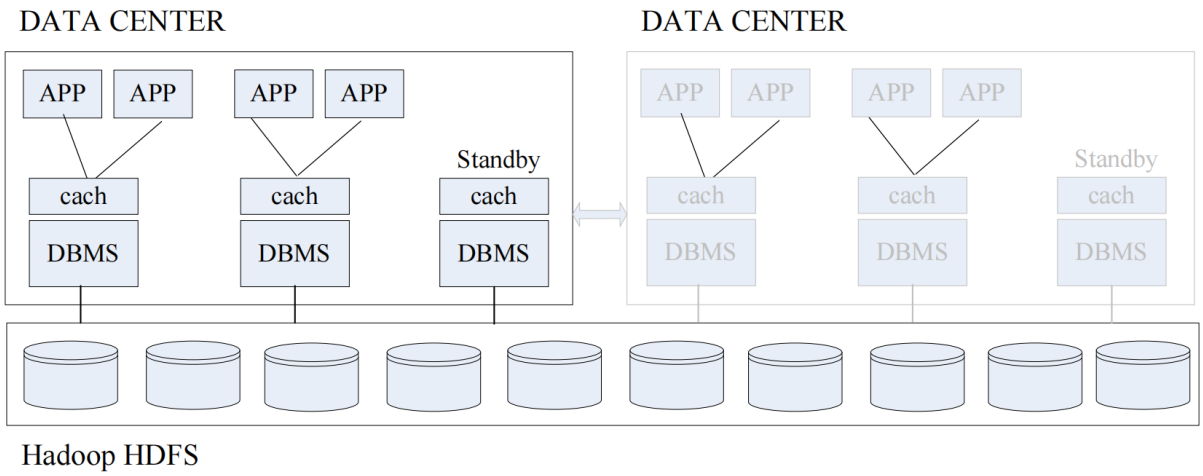


**Figure 1**

availability. To enhance performance it has an in-memory real-time database, a parallel distributed query engine and a data locality awareness, meaning that queries will always go to the closest copy of the needed information. It also supports multiple synchronous replication, achieving high availability even if one node is failing.

*2.1.2 Apache Cassandra.* Apache Cassandra is a NoSQL, eventual consistent distributed database system and supports large-scale business critical storage requirements and global availability [1]. It uses a distributed cluster membership architecture with a gossip protocol which helps for failure detection and increases scalability. It facilitates horizontal scaling and replication with the help of versioned hash-keys and a last-write-wins approach to find the newest version. It also provides its own Cassandra Query Language (CQL) which is a simpler, SQL-like query language.

*2.1.3 MongoDB.* MongoDB uses a sharding mechanism for distributed database systems [3]. Sharding means that the data is split into subsets and replicated several times, then distributed over multiple servers. This increases the overall availability and facilitates horizontal scaling. To increase data locality MongoDB introduced the concept of Zone Sharding where policies enforce data residency in specific zones.

*2.1.4 Data File System.*

- **Hadoop** [5]. Hadoop is an open-source framework for distributed storage and processing of large data sets across clusters of computers. It uses a simple programming model and is designed to scale up from single servers to thousands of machines. Key components include the Hadoop Distributed File System (HDFS) for reliable storage and MapReduce for parallel data processing. Its flexibility and fault-tolerant architecture make it ideal for big data applications in industries like finance, healthcare, and e-commerce.
- **FastDFS** [4]. FastDFS is a high-performance distributed file system tailored for handling large-scale file storage. Lightweight and efficient, it supports features like file uploading, downloading, and metadata management with built-in mechanisms for redundancy and load balancing. FastDFS is widely used for applications requiring fast and scalable file management, such as content delivery networks (CDNs) and video streaming platforms. Its minimalistic design prioritizes speed and reliability over complex data processing.

In summary, Hadoop is better suited for big data analytics scenarios, supporting complex computations and batch processing tasks such as data mining and machine learning model training. In contrast, FastDFS is ideal for use cases involving large volumes of files and frequent access, such as CDNs, image/video storage, and real-time file services.

Our system opted for FastDFS instead of Hadoop because FastDFS is comparatively more lightweight and easier to deploy. It is specifically designed for file storage and does not require Hadoop's complex dependencies, making it a better fit for our needs.

## 2.2 Problem Definition

Problem 1. *The aim of this project is to develop a distributed data system capable of efficiently managing both structured and unstructured data. The structured data includes five interrelated relational tables, while the unstructured data consists of various forms such as text, images, and videos related to the content of articles.*

*2.2.1 Data Processing.* First, we need to construct five relational tables: User, Article, Read, Be-Read, and Popular-Rank, and perform data fragmentation for each table.

- The **User** table records metadata such as user gender, region, email and so on. It is partitioned by region: users with region of "Beijing" are stored at Site 1, while users with region of "Hong Kong" are stored at Site 2. This is a fragmentation strategy without replica.
- The **Article** table records information such as the publication time, text storage path, image storage path, and video storage path. It is partitioned by category: articles with category = "technology" are stored only at Site 2, while articles with

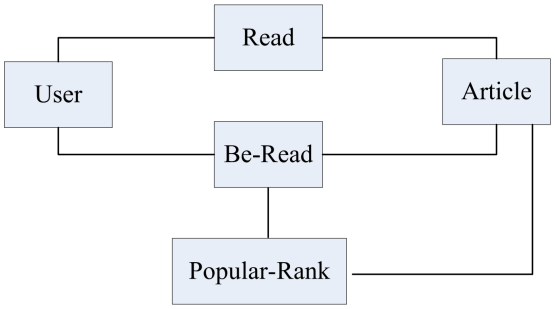


**Figure 2**

category = "science" have duplicates at both Site 1 and Site 2. This is a fragmentation strategy with duplication.

- The **Read** table logs the reading history of each user, including actions like shares, likes, and comments for each read article. Its fragmentation strategy is the same as that of the User table.
- The **Be-Read** table is derived from the Read table and records information about each article's reading activity, including the timestamps of each read and the corresponding interactions (shares, likes, comments). Its fragmentation strategy follows that of the Article table.
- The **Popular-Rank** table is generated based on the Be-Read table, recording the top five most popular articles at different time granularities (daily, weekly, monthly).

*2.2.2 Required Functions.* Implement a data center in a distributed context as Figure 1 with the following functionalities.

(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) (amount and location), workload, etc.
(5) (Optional) advanced functions
   (a) Hot / Cold Standby DBMSs for fault tolerance
   (b) Expansion at the DBMS-level allowing a new DBMS server to join
   (c) Dropping a DBMS server at will
   (d) Data migration from one data center to others

## 3 METHOD

Based on previous work, our system consists of four main components: a Distributed File System (DFS) for storing unstructured data, a Database Management System (DBMS) for storing structured data, a backend for responding to query services and a frontend for displaying user interaction pages , as shown in Figure 3. The next sections delve into these components, elucidating their initialization process and interaction mechanisms.

### 3.1 Database Management System

We use MongoDB as the database management system. MongoDB features built-in caching mechanisms and offers a monitoring tool (Mongo Compass) with user-friendly interface, greatly enhancing the system's usability. Additionally, MongoDB's compatibility with JSON-like data structures and querying language make it easy to get started. To ensure fault tolerance, we have set up backup nodes for both Site 1 and Site 2. If one of the primary databases crashes, the backup nodes can take over immediately, maintaining system continuity.

To minimize data redundancy, we pre-generate data tables locally and use the volume mechanism to map them to the *data_load/* directory within the MongoDB container. This way, each data table is mapped correspondingly in both Site 1 (Site 2) and Site 1 Backup (Site 2 Backup). We then run Mongo commands within the MongoDB container to import these data tables into the database, completing the initialization process.

The workload of loading the database can be monitored using Mongo Compass, as shown in Figure **??**.

### 3.2 Data File System

In our system architecture, the database only stores the indices of the files for articles, images, and videos rather than the actual data files. This is because unstructured data (such as images and videos) are large in size and diverse in format, making it inefficient to store them directly in a traditional database. Therefore, we use FastDFS, an efficient distributed file system, to manage unstructured data. When an external user or application sends a query request to the database, it returns only the file access index; the client then uses this index to request and retrieve the actual content from the file system.

Next, we will explain in more detail the initialization process of DFS and the interaction mechanisms of its core components.

FastDFS consists of three main components:

(1) Tracker Server: The Tracker Server acts as the coordinator of FastDFS, responsible for managing and scheduling storage servers. It needs to be started first. Upon startup, the Tracker Server loads its configuration, including network ports, log paths, working directories and so on. After initialization, it enters listening mode, awaiting connection requests from storage servers and clients.
(2) Storage Server: The Storage Server is responsible for actual file storage and registers its status with the Tracker Server. During startup, the Storage Server loads its configuration file, which contains storage paths, port numbers, cluster information and so on. It then sends a registration request to the Tracker Server, which adds it to the server list and records its status (e.g., available space and load).
(3) Client: The client serves as the interface for external access to the file system, typically initiated by applications or user terminals. Upon initialization, the client loads the address list of the Tracker Server to query storage node information in subsequent operations. When uploading a file, the client first requests the available Storage Server information from the Tracker Server. Based on the returned server address, the client uploads the file to the designated Storage Server.

Similar with the initialization of the DBMS, we employ the volume mechanism to map files that need to be uploaded from the local system to the storage container in FastDFS. This approach ensures

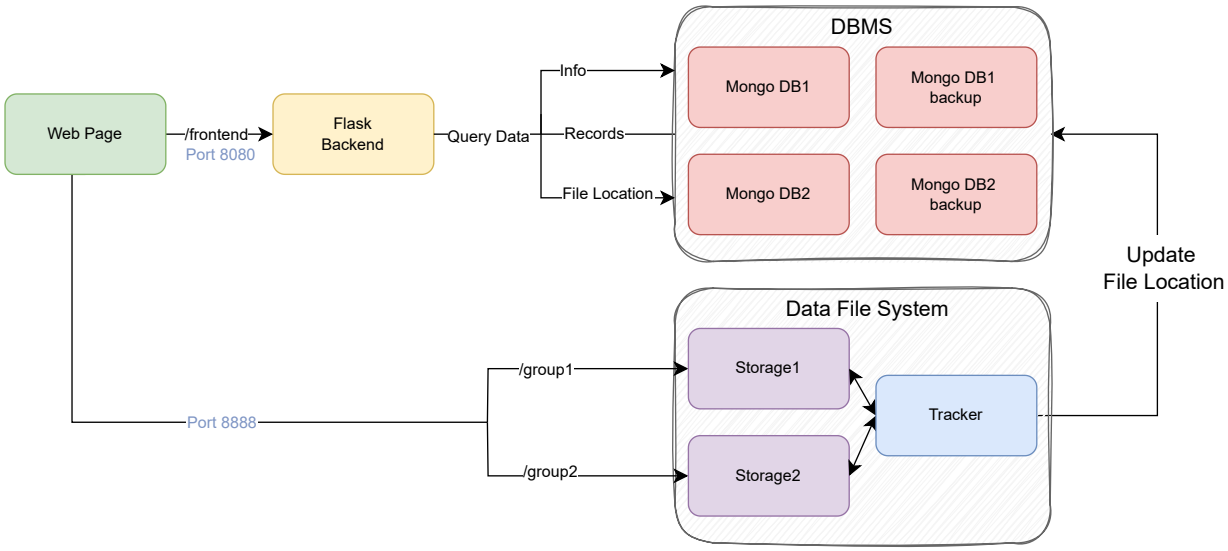


**Figure 3: The overview framework of our work.**

that storage nodes are isolated from external internet access while retaining the capability to receive uploads internally.

The file storage and access process in DFS is mainly completed through the following steps:

- **File Upload:** The client sends an upload request to the Tracker Server. The Tracker Server selects the most suitable Storage Server based on the current status of storage servers (such as remaining space and load) and returns its address. The client receives the address and uploads the file data directly to the designated Storage Server. Upon successful upload, the Storage Server returns the file access index (i.e., the file ID) and its storage path to the client. We store all file access indices in the **Mapping** Table in database for subsequent queries.
- **File Download:** When an external application queries a specific file from the database, the database returns the file access index. The client uses this index to send a query request to the Tracker Server. Based on the file index, the Tracker Server locates the Storage Server storing the file and returns its address to the client. The client then downloads the file from the specified Storage Server to retrieve the actual unstructured content.

It is worth mentioning that the Tracker Server monitors the status of each Storage Server in real time and performs scheduling based on the remaining storage space and current load to ensure even distribution of requests. In addition, FastDFS supports backup of the same file on multiple Storage Servers. When a Storage Server fails, the client can get the file from other replica servers, ensuring high availability and fault tolerance of the system. In the implementation of this system, we create two Storage nodes to achieve higher reliability.

By leveraging FastDFS for unstructured data storage, our distributed database system effectively manages structured data while providing fast access to large-scale unstructured data. The combination of the file system and database indices decouples storage and query functions, enhancing storage efficiency and offering flexibility and high availability for data expansion and maintenance.

## 3.3 Backend

The backend utilizes the Flask framework, a lightweight backend framework. The main functionalities provided by the backend include querying users or articles by ID, retrieving users' reading history, fetching the most popular articles, and more. Here are the detailed interfaces:

- **Retrieve the user list for the current page.** This interface receives requests from the frontend, processes the `pageid` parameter, and returns the user list for the current page. This list allows users to click and view detailed information.
- **Retrieve the article list for the current page.** This interface receives requests from the frontend, processes the `pageid` parameter, and returns the article list for the current page, enabling users to click and view detailed information.
- **Retrieve users' metadata and reading history.** Based on the `uid` parameter, this interface queries the `User` table to obtain the user's metadata. It then queries the `Read` table to fetch the user's reading history (including reading timestamps and interaction records). Using the `aid` field in reading history, the interface further queries the `Article` table to obtain article links. All retrieved information is aggregated and returned to the frontend.
- **Retrieve article information.** This interface uses the `aid` parameter to query the `Article` table for article details. Based on the stored filename of text, images and videos, it queries the `Mapping` table to obtain their access links in FastDFS. Subsequently, it queries FastDFS to retrieve the textual content. It also queries the `BeRead` and `Read` Tables to get the corresponding comments and statistical information

on the article, for example how many times the article was viewed or how often it was shared. The final response to the frontend includes article metadata, textual content, image links, video links, comments and statistical information.

- **Search users.** This interface receives a `search_text` parameter from the frontend and searches the `uid`, `email`, and `phone` fields in the `User` table. It then returns a list of users that match the filter criteria.
- **Search articles.** This interface receives a `search_text` parameter from the frontend and searches the `aid` and `title` fields in the `Article` table. It then returns a list of articles that match the filter criteria.
- **Retrieve the most popular articles.** This interface fetches the list of most popular articles based on the given date and the chosen granularity: daily, weekly or monthly.

It is worth mentioning that all interfaces are equipped with fault-tolerance mechanisms. If a database crashes, each backend interface can still access the backup database, thereby ensuring uninterrupted service delivery.

## 3.4 Frontend

The frontend is implemented by native HTML, CSS, and JavaScript. The main pages are as follows:

- **User List Page.** Displays a preview of 40 users per page, with clickable links to navigate to individual user information pages. A "Article Pages" button is available in the top-left corner to switch to the article list page, as shown in Figure 4.
- **Article List Page.** Displays a preview of 40 articles per page, with clickable links to navigate to individual article content pages. A "User Pages" button is available in the top-left corner to switch to the user list page, as shown in Figure 6.
- **User Information Page.** As shown in Figure 5, this page displays the user's metadata (e.g., gender, region, language, etc.) and reading history, including reading timestamp and interactions (likes, shares, and comments) for each article. Users can click on any article preview to navigate to the article content page.
- **Article Content Page.** As shown in Figure 7, this page presents the article's title, publication date, text, images, and videos. Additionally, it shows user interaction statistics, such as the number of likes, shares, and comments.
- **User Search Results Page.** Displays previews of users that match the search criteria, with clickable links to navigate to a detailed user information page.
- **Article Search Results Page.** Displays previews of articles that match the search criteria, with clickable links to navigate to a detailed article content page.
- **Most Popular Articles Ranking.** Displays a leaderboard of the most popular articles, as shown in Figure 8.

All user-related pages feature a search bar at the top, allowing users to search by `uid`, `email`, or `phone number` to access the user search results page. Similarly, all article-related pages include a search bar at the top, enabling searches by `aid` or `title` to access the article search results page.

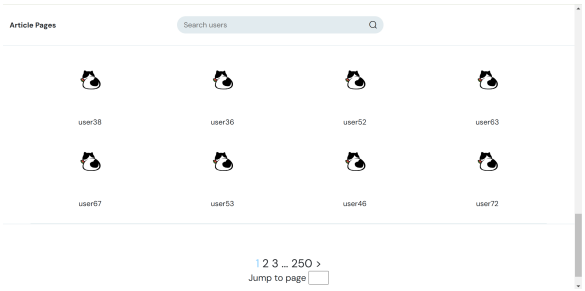


**Figure 4**

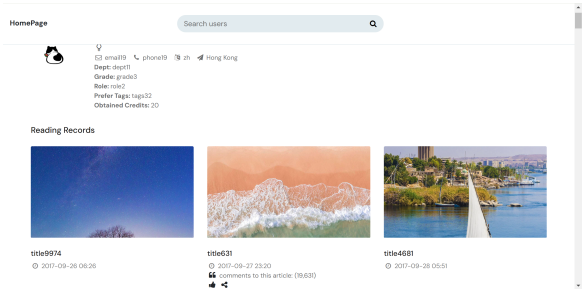


**Figure 5**

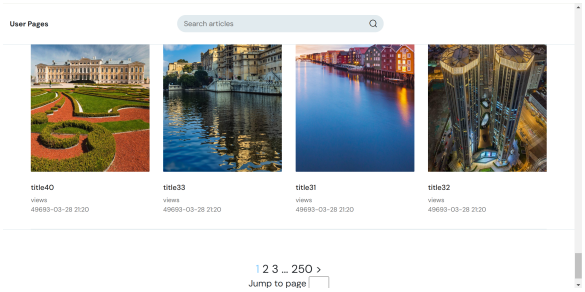


**Figure 6**

We utilized aesthetically pleasing fonts and implemented simple animation effects, resulting in a visually appealing and engaging final presentation.

# 4 EVALUATION

## 4.1 Slice and Import of the Tables

First, the `User`, `Article`, and `Read` tables are sliced to match the given fragmentation for the allocation. This operation takes 54 seconds. Next, the tables are imported into MongoDB, which takes 109 seconds.

Subsequently, the `Be-Read` and `Popular-Rank` tables are generated based on the `Article` and `Read` tables and then imported into MongoDB, taking a total of 36 seconds. This process involves complex collection operations, such as `group`, `sort` and so on. We utilize a pipeline to implement the entire workflow, making it clearer and more efficient.

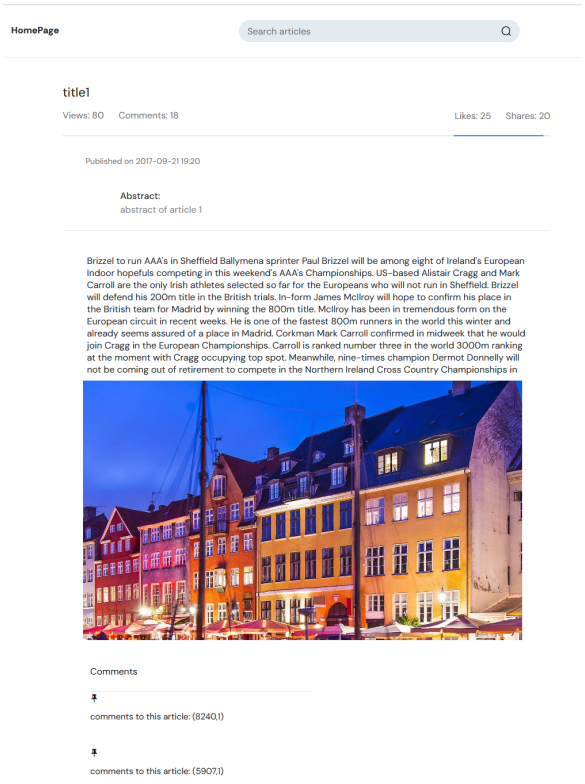


**Figure 7**

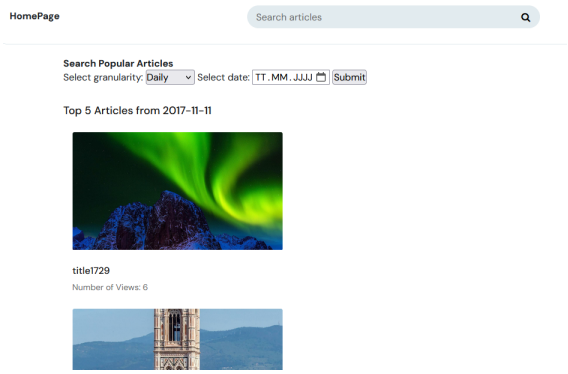


**Figure 8**

## 4.2 Uploading Files

Uploading files to the Data File System takes 1480 seconds, while storing mappings in the `Mapping` table takes 6 seconds. It can be observed that the file upload time is significantly higher than all other operations, making it the bottleneck of the entire system initialization.

## 4.3 Database Monitoring

We use MongoDB Compass to monitor data, workload, and other information in the database. The graphical interface of MongoDB Compass allows us to perform CRUD (Create, Read, Update, and Delete) operations on the database without writing any code.

In addition to routine tasks such as viewing and editing data, we can utilize the `Performance` feature of MongoDB Compass to analyze the workload of each database. From the Figure 9, we can observe that during the import of the `Read` table into `DB1`, there are several metrics monitored by MongoDB Compass at real time. The `OPERATIONS` metric shows the number of operations performed on the database. The `READ & WRITE` metric shows the counts of active reads, queued reads, active writes, and queued writes. The `NETWORK` metric visualizes the changes in network requests over time. Besides these metrics, the `MEMORY` metric provides memory usage statistics, while `HOTTEST COLLECTIONS` highlights the collections with the most activity. The `SLOWEST OPERATIONS` section displays the slowest operations reported by the `db.currentOp()` command.

## 4.4 Optional Advanced Functions

*4.4.1 Dropping a DBMS Server at Will.* With the help of Docker containers, we can shut down a database node with a single command: `docker stop [db_name]`. Additionally, we can use MongoDB's `db.getSiblingDB('db_name').dropDatabase()` command to delete a specific database within a node.

*4.4.2 Hot Standby DBMSs for Fault Tolerance.* Our system implements a Hot Standby mechanism, where the backup system runs concurrently with the main system and is always ready to take over its workload.

To simulate a failure scenario, suppose the `DB1` container is stopped and cannot provide services (this can be simulated by running 'test/stop_db.sh'). After `DB1` crashes, the entire system continues to function normally because all request logic first targets the primary database, and if it fails, the system automatically redirects requests to the backup of the primary database (though this may take longer, as a delay is required). This demonstrates not only the fault tolerance of our DBMS but also the scalability of our system, as new database nodes can be added at any time.

*4.4.3 Expansion at the DBMS Level Allowing a New DBMS Server to Join and Data Migration from One Data Center to Another.* Now consider another failure scenario where all data in `DB1` is lost. In such a case, we can restore `DB1`'s data using the backup database. Since all databases have consistent mount paths, we can use the `mongodump` command to dump data from the backup database `DB1_bak` into local storage. Through the mounting mechanism, the dumped data also appears in `DB1`'s local storage, allowing us to use the `mongorestore` command to restore `DB1`'s data.

This process effectively demonstrates our system's capability to migrate data between data centers.

## 5 CONCLUSION

To summarize, we have designed and implemented an efficient and robust distributed database system. The system uses MongoDB as the database management system, for storing the structured data,


**Figure 9: Mongo Compass**

and FastDFS to efficiently handle the unstructured data. The backend query interfaces are implemented using the Flask framework and the frontend is designed for an intuitive and visually appealing approach to query and display the data.
By implementing a hot standby mechanism, the possibility of data migration and therefore also allowing new DBMS server to join, shows that the system is fault tolerant and scalable. These features make our system more flexible to adapt to various users needs.
The evaluation of our system's performance shows the efficiency and reliability in bulk-loading data and handling query executions. Moreover, since the monitoring can be done with MongoDB Compass, a standard industry tool, it eases the use in practical use cases and increases its practicality.
Going forward, future works can further enhance this project. Possible areas could be for example to focus on eliminating the bottleneck of the bulk file upload to the Data File System or to expand the systems functionality.

In conclusion, this project successfully implements an efficient and user-friendly distributed database system. It combines theoretical knowledge while addressing real-world user needs. Furthermore, it provides a foundation that can be expanded and adapted for real-world applications or used for further research on distributed database systems.

# REFERENCES


[1] [n. d.]. Apache Cassandra Architecture Overview. https://cassandra.apache.org/doc/latest/cassandra/architecture/overview.html Accessed: 2024-12-19.

[2] [n. d.]. MySQL NDB Cluster CGE. https://www.mysql.com/products/cluster/ Accessed: 2024-12-19.

[3] [n. d.]. Sharding in MongoDB. https://www.mongodb.com/resources/products/capabilities/sharding Accessed: 2024-12-19.

[4] X. Liu, Q. Yu, and J. Liao. 2014. Fastdfs: A high performance distributed file system. *"ICIC Express Letters, Part B: Applications An International Journal of Research and Surveys"* 5 (2014), 1741–1746.

[5] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.