Design and Implementation of a Configuration-Driven Distributed Data Center for Big Data Management

Théodore Billotte (2025400572), Louis Rollet (2025400573)

Abstract

As data volume and velocity increase, traditional monolithic database systems struggle to maintain performance and availability. This project implements a simulation of a geo-distributed data center designed for a news media platform. We propose a hybrid architecture utilizing MongoDB for structured data sharding, Hadoop HDFS for unstructured media storage, and Redis for high-speed caching. A key contribution of this work is the implementation of a "Configuration-Driven Architecture," which allows for dynamic horizontal scaling and automated data migration without code modification. We evaluate the system's ability to handle bulk ingestion, enforce complex fragmentation strategies across geographic regions (Beijing and Hong Kong), and perform live cluster expansion.

1 Introduction

The management of Big Data requires systems that ensure high availability, fault tolerance, and efficient data retrieval [6]. In the context of a news platform, data is often heterogeneous, consisting of structured metadata (users, readings) and unstructured content (videos, images). Furthermore, access patterns frequently exhibit geographical locality.

This project aims to design a distributed database system that addresses these challenges. By simulating a multi-node environment, we explore techniques for data partitioning, replication, and distributed query processing. The motivation is to move beyond static database configurations towards a flexible, scalable infrastructure capable of adapting to changing workload demands through simple configuration updates.

2 Existing Solutions

Traditional solutions for data management typically fall into two categories: vertical scaling of monolithic SQL databases or manual sharding at the application level.

- **Monolithic SQL Clusters:** While offering strong ACID guarantees, they suffer from write bottlenecks and are difficult to scale geographically without complex multi-master replication setups [9]. Furthermore, the rigidity of the relational model can be a hindrance when dealing with rapidly evolving data schemas [5].
- **Application-Level Sharding:** Logic for routing data is embedded in the application code. This makes the system brittle; adding a new shard requires code changes and downtime to redeploy the application.

Our proposed solution leverages MongoDB's native auto-sharding capabilities combined with a custom configuration layer. This hybrid approach allows for the flexibility of NoSQL while maintaining the strict data locality rules required by geo-distributed applications, overcoming the rigidity of traditional methods [4].

3 Problem Definition

The system must manage five relational entities: *User*, *Article*, *Read*, *Be-Read*, and *Popular-Rank*. The specific constraints and requirements are:

1. **Heterogeneous Data Management:** Structured data must be stored in a document store (MongoDB), while media files must reside in a distributed file system (HDFS).
2. **Geo-Fragmentation:** User data must be fragmented based on region. Users in "Beijing" must be allocated to DBMS1, while users in "Hong Kong" must be allocated to DBMS2.
3. **Category-Based Partitioning with Replication:**
   - *Science* articles must be replicated across both DBMS1 and DBMS2 to ensure high availability.
   - *Technology* articles are initially allocated only to DBMS2.
4. **Scalability:** The system must support the addition of new DBMS nodes (expansion) at runtime to handle increased load, involving automated data migration.

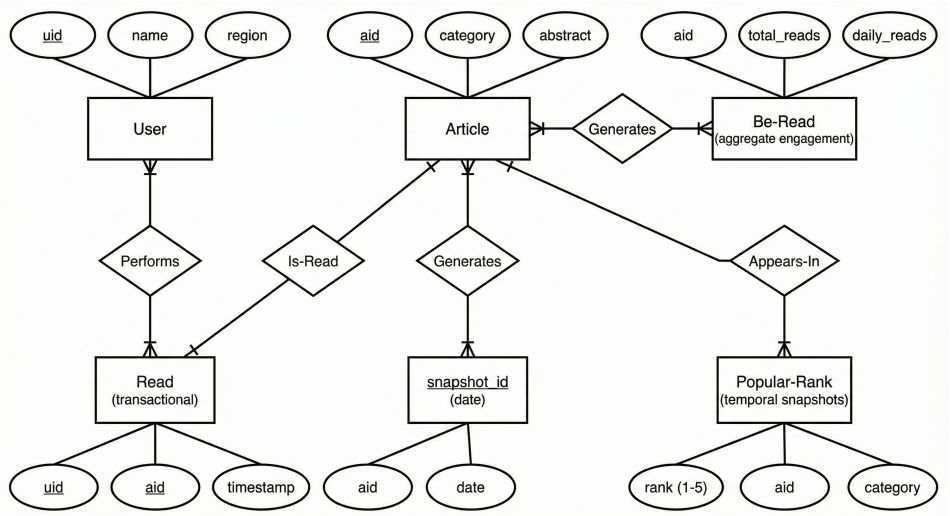

Figure 1: Entity-Relationship Diagram of the News Platform Data Model

4 Proposed Solution

4.1 System Architecture

We implemented a three-tier architecture containerized via Docker:

- **Presentation Layer:** A React.js frontend providing a real-time dashboard for monitoring system health and visualizing data fragmentation.
- **Application Layer:** A Python Flask API acting as the central gateway. It implements the *ClusterService* logic, routing queries via a `mongos` router and implementing a Cache-Aside pattern with Redis.
- **Data Layer:**
  - **MongoDB Cluster:** Configured with a Config Server and multiple Shard Servers (Replica Sets) [1].
  - **Hadoop HDFS:** A NameNode and DataNode cluster for storing text, images, and videos. This architecture is heavily inspired by the Google File System (GFS), optimized for high throughput of large datasets [7].
  - **Redis:** An in-memory key-value store for caching frequently accessed articles.

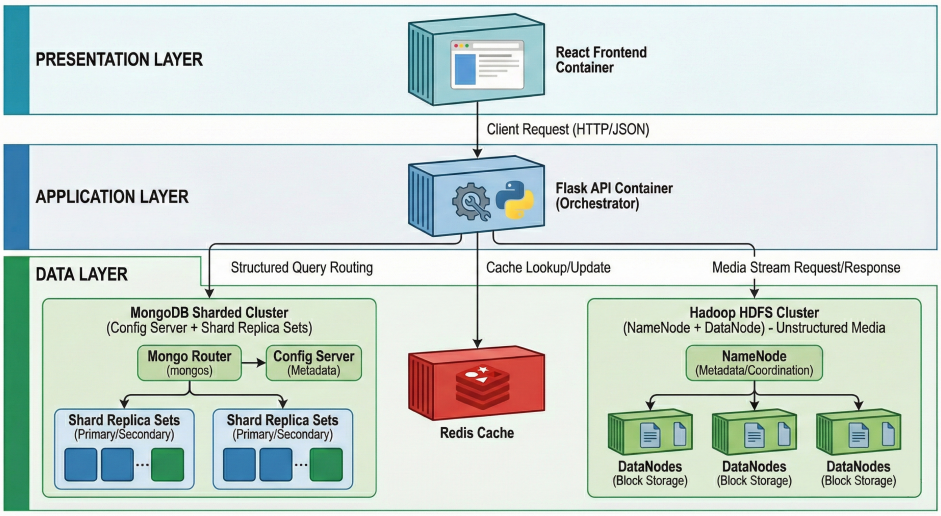

Figure 2: System Architecture and Component Interaction

4.2 Configuration-Driven Implementation

Unlike traditional setups where sharding rules are static or manual, our solution utilizes a `cluster_config.json` file. The application parses this file at startup to dynamically:

1. Register available shards with the cluster.
2. Define *Zones* (Tags) on specific shards (e.g., tagging Shard 1 with "Beijing").
3. Enforce sharding keys and chunk ranges.

This allows for the seamless addition of a third shard by simply updating the JSON file and restarting the application containers, satisfying the optional requirement for DBMS-level expansion.

4.3 Data Fragmentation and Replication Strategy

We utilized MongoDB's Zone Sharding to implement the specific project requirements. Table 1 details the strategy for all five entities.

Collection	Shard Key	Fragment 1 (Shard 1)	Fragment 2 (Shard 2)
User	{region, uid}	Region: Beijing	Region: Hong Kong
Article	{category, aid}	Category: Science	Category: Tech, Science (Replica)
Read	{user_region, uid}	Region: Beijing	Region: Hong Kong
Be-Read	{category, aid}	Category: Science	Category: Tech, Science (Replica)
Popular-Rank	{granularity}	Daily	Weekly, Monthly

Table 1: Fragmentation Strategy

To achieve the replication of *Science* articles, we implemented an application-level logic that detects the "Science" category upon insertion and creates a duplicate document with a `dup_` prefix. The Zone logic ensures the original resides on Shard 1 and the replica on Shard 2.

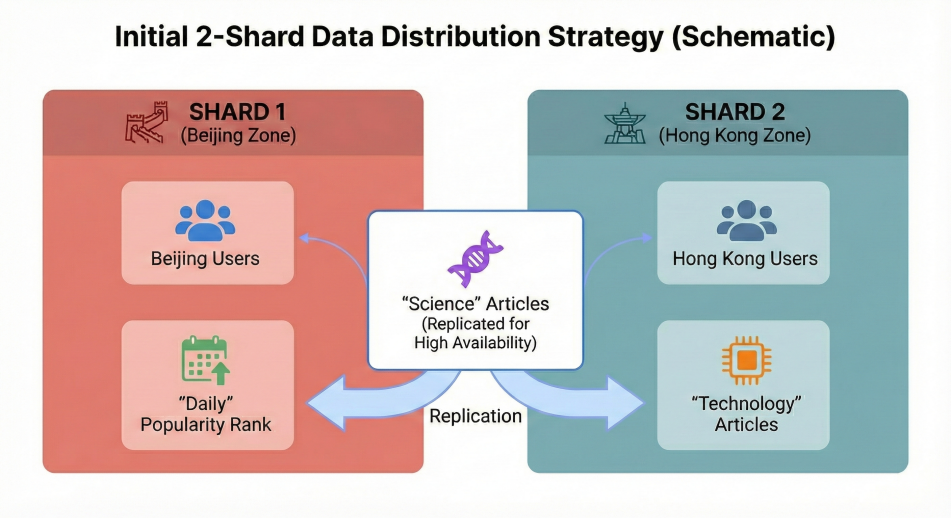

Figure 3: Initial Data Fragmentation and Replication Topology

5 Implementation Details

5.1 Bulk Data Loading

An `IngestionService` was developed to parse raw `.dat` files. It effectively handles the bulk insertion of thousands of records using `insert_many` for performance. Crucially, the ingestion process respects the configuration rules; if a record matches a replication rule defined in the config, the service automatically generates the replica before insertion.

5.2 Unstructured Data Handling

Files associated with articles are uploaded to HDFS. The API server utilizes the `hdfs` python client. When a user requests an article, the metadata is fetched from MongoDB, and the associated media files are streamed from HDFS, merged, and sent to the client.

5.3 Horizontal Expansion (Bonus Feature)

We implemented a live expansion capability. Initially, the system runs with two shards. To handle increased load on the "Technology" sector, a third shard is defined in the configuration. Upon deployment, the `ClusterService`:

1. Detects the new `rsShard3`.
2. Registers it with the cluster.
3. Updates the Zone Ranges, reassigning the "Technology" tag from Shard 2 to Shard 3.

MongoDB's internal balancer then automatically migrates the chunks, demonstrating live data migration without service interruption.

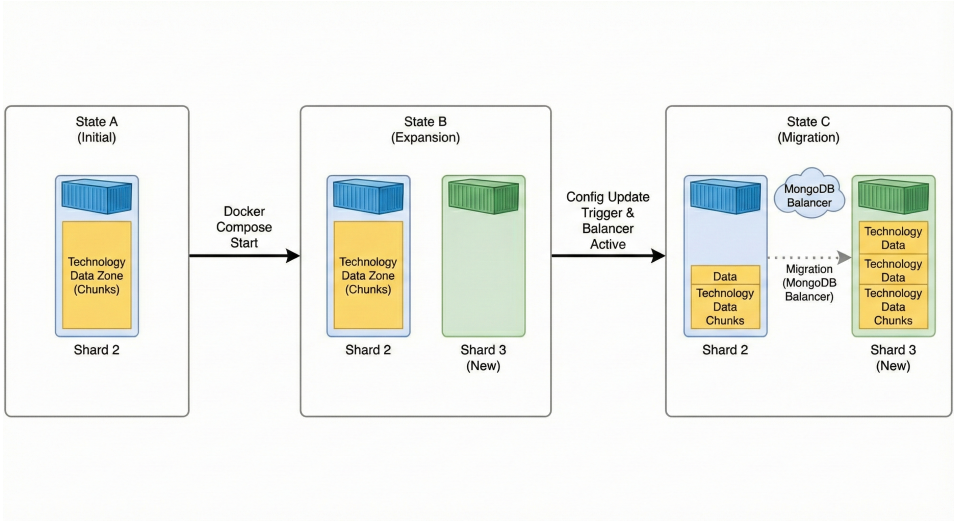

Figure 4: Dynamic Cluster Expansion and Data Migration Process

6 Solution Evaluation

6.1 Performance

The integration of Redis significantly improved read performance for popular articles. By caching the results of the complex join operation (Article + Be-Read statistics), subsequent requests experienced near-instantaneous response times (cache hit), reducing load on the MongoDB cluster. This aligns with standard web caching strategies where frequently accessed static content is served from in-memory stores to minimize latency [8].

6.2 Scalability Verification

We verified the scalability requirement by initially deploying the system with two shards. The `/api/admin/fragmentation` endpoint confirmed that "Technology" articles resided on Shard 2. After updating the configuration to introduce Shard 3 and re-zoning "Technology" to it, we observed the document count on Shard 2 decreasing and Shard 3 increasing, confirming successful DBMS-level expansion and migration.

7 Conclusion and Future Work

This project successfully demonstrates a robust, distributed data center capable of handling complex fragmentation and replication rules. The configuration-driven approach provides significant operational flexibility. Future work could include implementing automated failover for the API layer and integrating a more sophisticated load balancer for the frontend.

8 Workload Allocation

- **Théodore:** Designed the Docker infrastructure (Compose, HDFS, Mongo Cluster), implemented the *ClusterService* for dynamic sharding, and handled the Bulk Loading logic.
- **Louis:** Developed the Flask API endpoints, implemented the Redis caching layer, built the React Frontend dashboard, and managed the unstructured data integration with HDFS.

References

[1] MongoDB, Inc. *Sharding*. Available at: `https://www.mongodb.com/docs/manual/sharding/`

[2] Apache Software Foundation. *HDFS Architecture Guide*. Available at: `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`

[3] Redis Ltd. *Client-side caching in Redis*. Available at: `https://redis.io/docs/manual/client-side-caching/`

[4] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*. Springer, 2011.

[5] R. Cattell, "Scalable SQL and NoSQL data stores," *SIGMOD Record*, vol. 39, no. 4, pp. 12-27, 2011.

[6] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51-59, 2002.

[7] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Operating Systems Principles (SOSP)*, 2003, pp. 29-43.

[8] S. Podlipnig and L. Böszörményi, "A survey of Web cache replacement strategies," *ACM Computing Surveys*, vol. 35, no. 4, pp. 374-398, 2003.

[9] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems," in *Proc. 20th Int. Conf. Distributed Computing Systems (ICDCS)*, 2000.