Contents

Manual

1.1 Prerequisites

1.1.1 Cloning the Git Repository

To obtain the project source use the SSH method to clone, run:

```
git clone git@git.tsinghua.edu.cn:fw24/ddbs_project.git
```

1.1.2 Data Generation

1. Use the provided Python program

   `genTable_sql_relationalDB10G.py`

   to generate data for articles.

2. Verify the generated directory structure:

```
articles/
 article1/
  text_a1.txt
  image_a1_1.jpg
  video_a1.mp4
 article2/
  text_a2.txt
  image_a2_1.jpg
  image_a2_2.jpg
...
```

1.1.3 Technologies to Install

The following technologies are required for setting up the system:

- **Docker**: For containerizing Hadoop services.

- **Hadoop**: For storing and processing unstructured data in HDFS.
- **Kubernetes**: For managing Vitess and MySQL shard deployment.
- **MySQL**: As the relational database management system.
- **Vitess**: For query routing and distributed MySQL management.
- **Python**: For generating data and accessing HDFS through scripts.
- **Minikube**: To set up a Kubernetes environment locally.
- **kubectl**: Kubernetes CLI tool for managing the cluster.
- **MySQL client**: For connecting to the database.
- **vtctldclient**: For managing Vitess configurations.

1.2 System Overview

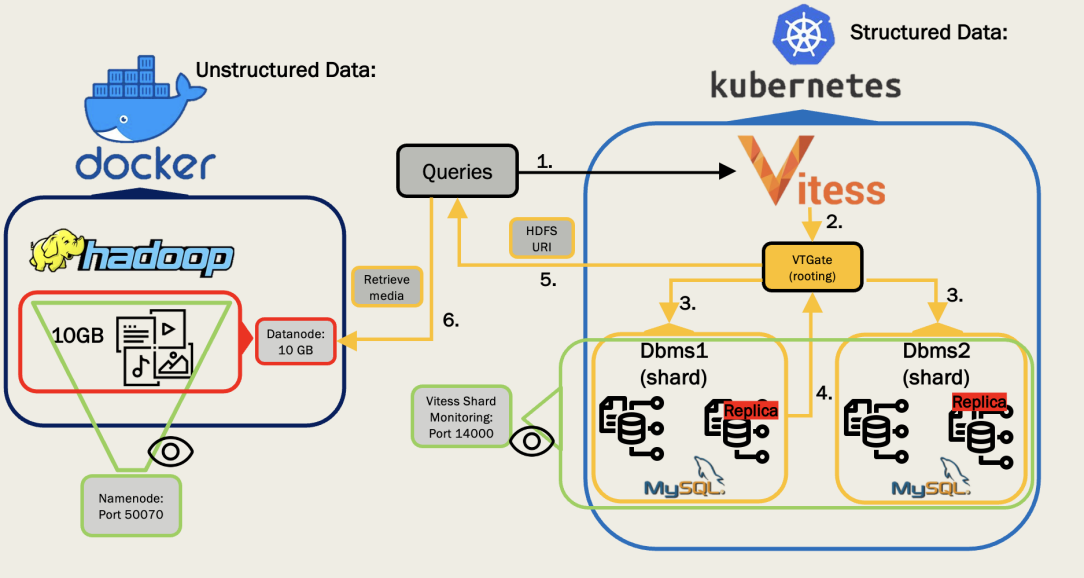

Figure 1.1: System Overview Diagram

1.3 For Docker and Hadoop

Prerequisites

1. Ensure Hadoop is running in a Docker container.
2. Generate data according to the project description:

- **Article table:** Includes text files, 1–5 images per article, and videos for 5% of the articles.
- Data should be stored in a directory structure grouping files by article.

Steps to Load Files into HDFS

Step 1: Generate Data

1. Use the provided Python program to generate 10,000 articles, each with associated text, images, and videos.
2. Confirm the directory structure is as follows:

```
articles/
 article1/
  text_a1.txt
  image_a1_1.jpg
  video_a1.mp4
 article2/
  text_a2.txt
  image_a2_1.jpg
  image_a2_2.jpg
```

Step 2: Start Hadoop Services Run the following command to start Hadoop services:

```
docker-compose up -d
```

```
# 1. Copy the articles directory into the Hadoop container:
docker cp /path/to/articles hadoop-master:/tmp/articles

# 2. Verify the files are successfully copied:
docker exec -it hadoop-master bash
ls /tmp/articles
```

Step 4: Create Target Directory in HDFS Create a target directory in HDFS to store the uploaded data:

```
hdfs dfs -mkdir /articles
```

```
# Upload the articles directory into HDFS:
hdfs dfs -put /tmp/articles/* /articles/
```

```
# 1. List the files in HDFS to ensure successful upload:
hdfs dfs -ls -R /articles

# 2. Check HDFS space usage:
hdfs dfsadmin -report
```

Step 7: Access Hadoop Web UI

1. Open the Hadoop NameNode Web UI in your browser:
   `http://<docker-host-ip>:50070`
2. Navigate to `/articles` under *Utilities > Browse the file system* to view the uploaded data.

Step 8: Monitor Job Progress Use the following command to monitor progress for large datasets:

```
hdfs dfsadmin -report
```

Alternatively, check progress via the ResourceManager UI:

`http://<docker-host-ip>:8088`

1.4 For Kubernetes and Vitess

Prerequisites

1. Install Docker Engine.
2. Start Minikube:

   ```
   minikube start --kubernetes-version=v1.28.5 --cpus=4 --memory=11000 --
       disk-size=32g
   ```

3. Install Kubernetes CLI (`kubectl`).
4. Install MySQL client and `vtctldclient`.

Steps to Set Up Kubernetes

```
# 1. Apply the required configuration files:
kubectl apply -f operator.yaml
kubectl apply -f example-cluster-config.yaml
kubectl apply -f vitesscluster.yaml

# 2. Verify services are running:
kubectl get pods
```

```
./pf.sh
```

```
# 1. Apply database schemas:
./vtctldclient --server=localhost:15999 ApplySchema \
    --sql-file="./schema/lookup_tables.sql" lookup
./vtctldclient --server=localhost:15999 ApplySchema \
    --sql-file="./schema/dbms/create_tables.sql" dbms

# 2. Apply VSchema files:
./vtctldclient --server=localhost:15999 ApplyVSchema \
    --vschema-file="./vschema/lookup_vschema.json" lookup
./vtctldclient --server=localhost:15999 ApplyVSchema \
    --vschema-file="./vschema/dbms_vschema.json" dbms
```

```
# 1. Alias MySQL connection:
alias mysql="mysql -h 127.0.0.1 -P 15306 -u user"

# 2. Connect to MySQL:
mysql
```

Steps to Set Up External jobs

```
# Prerequisite:
cd <project_directory>/external_jobs
```

```
# 1. Starting up media path service:
python3 AddMediaPaths.py
# 2. Starting up duplication across shards:
python3 duplicate_update.py

# 3. Starting up updating shard lookup:
python3 lookup_update.py
```

Monitoring

Step 4: Connect to Cluster

1. Check health of Kubernetes pods and services:

   ```
   kubectl get pods
   kubectl get svc
   ```

2. View logs for debugging:

   ```
   kubectl logs <service>
   ```

Access Web Interfaces

- Hadoop NameNode: `http://<docker-host-ip>:50070`
- Vitess Monitoring: `http://127.0.0.1:14000`

Example Operations

```
# Selecting specific shards in Vitess:
USE ‘dbms:-80‘;
USE ‘dbms:80-‘;

# Query execution:
SELECT * FROM article WHERE category=’science’;
```

1.5 Monitoring

- HDFS monitoring commands (`hdfs dfsadmin`, `kubectl logs`, etc.).
- HDFS monitoring with localhost:50070 running in your browser. It provides a basic monitoring of the storage of the media files.

- To keep an eye on the DBMS structure, use the integrated Web UI by entering 127.0.0.1:14000 in your browser.

Report

2.1 Problem Background and Motivation

- The requirements of the project stems from creating a fragmented database managing both relational data and large objects. This leads us to two requirements: 1) an object store for efficiently storing large videos/text/images, and 2) A relational database, allowing us to perform the specified fragmentation and handling complex queries.
- The required fragmentation specifies a high level of granularity and fine-tuning. We require features not overly common such as duplication of fragments and dependent table fragmentation (i.e fragment a table based on fields within a different tables).
- Disregarding the fine details, we firstly need a couple requirements: 1) Route queries to the correct fragment 2) Read queries from the correct fragment and aggregate them together.

2.2 Existing Solutions

- Their are a couple existing solutions for MySQL which allows query routing and aggregation: 1) Vitess: A horizontal scaling solution for production databases, 2) MySQL with custom middle-layer: Highly customized solution for small-scale systems, 3) Shardingsphere: Allows for flexible sharding strategies but with limited scalability, 4) MySQL federation with MySQL proxy: A combination of MySQL tools with community support but can be limited in performance.
- Due to the detailed fragmentation rules we require, we cannot pick a database-as-a-service. For MySQL particularly, there exists a couple solutions out there including Amazon Aurora and TiDB. Within Amazon, we do not see the traditional distribution of a NewSQL database but a managed distributed storage with a single DBMS; this is done through replacing InnoDB with their in-house approach. On the other hand, TiDB is a MySQL compatible NewSQL database, at the SQL-layer it uses RAFT protocol with its key-value store at its the storage layer. Both of

these solutions have built in managed fragmentation, allowing for high scalability, fault-tolerance and reliability but limited in customizability.

- There are many existing solutions for object stores. Some commercially available solutions on the cloud including Amazon S3, Google cloud and Amazaon S3. For on-premise, there exists solutions such as MinIO. We also have the choice of distributed file systems including: Hadoop file system (HDFS) and and GlusterFS.

- We chose to manage our relational data with Vitess due to its robust features that align with our project's needs, including: 1) Scalability 2) Replication 3) Node management. 1/2) Scalability and Replication: It allows use to setup many vtgates, acting as query routers. In addition, it allows us to set up many read replicas, allowing for high-throughput reads. 3) Replication and Management: Vitess provides built-in cluster management, including monitoring through mechanisms like heartbeats. It allows to easily start up and down new databases, while also handling fail-overs of the primary and replica nodes.

- A downside for Vitess is its customizability of fragmentation such as duplication across fragments. We required a separate service to implement this functionality, reducing the overall effectiveness of having duplication. Although, we get similar optimizations through Vitess and its replicated databases.

- For our videos, images, and large text, we decided to go for the Hadoop approach. Firstly, since its file-based it is cheaper to scale being able to run on commodity hardware. Secondly, it can be built to be distributed, allowing for replication across multiple nodes. Lastly is its performance, with objects stored across multiple nodes it ensures fast I/O operations for large files.

2.3 Proposed Solution

Architecture Overview

Figure 2.1 illustrates our proposed architecture, combining Docker-based Hadoop for unstructured data storage with Kubernetes and Vitess for structured data management. Specifically:

- **Hadoop in Docker** handles large volumes of unstructured data (e.g., text, images, videos). We store bulk data in HDFS, accommodating up to 10 GB or more as needed.

- **Kubernetes with Vitess** orchestrates multiple MySQL shards for structured data. Vitess provides sharding, replication, and a unified query interface, while

Kubernetes automates deployment, scaling, and failover processes.

- **Vitess MySQL Shards (DBMS1 and DBMS2)** host structured relational data. Each shard is replicated to ensure high availability, and Vitess's VTGate layer routes queries accordingly.

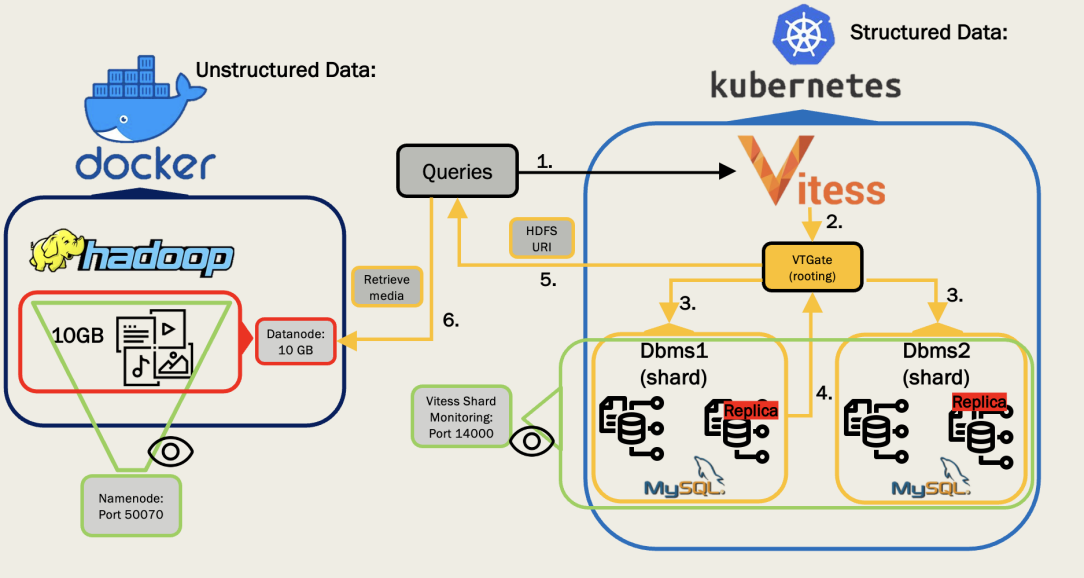

Figure 2.1: High-Level System Architecture (Docker + Hadoop + Kubernetes + Vitess).

This combined setup meets the central project requirement of building a distributed data center capable of managing both structured and unstructured datasets efficiently, while maintaining scalability and fault tolerance.

Partitioning Strategies

Our system employs both horizontal and vertical partitioning strategies:

- **Horizontal Partitioning (Sharding):** We split relational tables across multiple MySQL shards (e.g., DBMS1 and DBMS2). Vitess manages the distribution of data among shards and automatically routes queries to the appropriate shard, minimizing manual configuration.
- **Vertical Partitioning:** Unstructured data (e.g., text files, images, videos) is stored in HDFS, while the structured attributes (e.g., article metadata, user details) are kept in MySQL. This division maximizes flexibility and leverages the strengths of both Hadoop (for large file storage) and MySQL (for quick relational lookups).

Through Vitess's replication settings, each shard has replicas to maintain redundancy and support read scalability. Consequently, we can accommodate large data volumes while ensuring reliability.

Query Workflow

The query lifecycle unfolds as follows (see Figure 2.1):

1. **Clients submit queries** (e.g., data insert, update, or retrieval) through the Vitess VTGate endpoint.
2. **VTGate determines the target shard** using VSchema definitions and routes queries appropriately.
3. **MySQL shards (DBMS1 and DBMS2)** execute the queries. Each shard stores a subset of the total relational data, replicated to ensure resilience.
4. For unstructured data (e.g., media files), the system references **HDFS URIs** stored alongside the structured records.
5. If retrieval of unstructured content is requested, a **separate call to HDFS** is made, served by the Docker-based Hadoop environment.
6. **Results are returned** to the client, combining structured query output from MySQL shards with any requested unstructured data from HDFS.

Fulfillment of Project Requirements

1) Bulk Data Loading with Partitioning and Replication Our architecture supports bulk loading to Vitess-managed shards (structured records) due to efficient table-lookups (for fragmentation) and efficient router. Vitess inherently handles data partitioning among fragments, and replication is configured to maintain fault tolerance.

2) Efficient Execution of Data Operations Data insert, update, and query operations are conducted via MySQL syntax on Vitess. Because Vitess abstracts sharding logic, developers can perform SQL operations as if working on a single logical database. Queries remain performant thanks to horizontal scaling and replication.

3) Monitoring of DBMS and Data

- **Vitess Monitoring:** Vitess exposes shard monitoring on port 14000, displaying replica status, workload metrics, and query statistics.
- **Hadoop Monitoring:** The Hadoop NameNode UI (default port 50070) and YARN ResourceManager UI offer insights into HDFS usage and cluster health.

4) (Optional) Advanced Functionalities

- **Dropping a DBMS Server at Will:** With replication configured, decommissioning one shard node remains feasible without compromising data accessibility. Replicas can take over automatically.
- **Fault Tolerance and Expansion:** Additional DBMS servers can join as new shards or replicas under Vitess, providing a path for horizontal scaling and failover strategies.

Summary of Our Solution

By integrating Docker (for Hadoop) and Kubernetes (for Vitess), we have built a multi-layered data center that fulfills core requirements such as bulk loading, partitioning, replication, efficient queries, and monitoring. Optional features like dropping a DBMS server at will or adding new DBMS nodes are also supported, showcasing the system's adaptability and robustness.

2.4 Conclusion and Future Work

2.4.1 Future Improvements

- Develop a user-friendly front-end to simplify interactions with the system and improve the overall user experience.
- Explore resource optimization techniques to reduce the system's high memory footprint.
- Find optimizations of reading fragmented data. For example, duplication allows for splitting data across shards allowing for parallel reads reducing total query execution time.

2.4.2 Conclusion

- The current architecture is memory-intensive, requiring at least 10 GB of RAM to operate effectively. This is due to the use of kubernetes (minikube) with many pods. This would be effective in production databases as this is distributed across many commodity machines, but non-effective in small scale systems.
- Throughout this project, we gained considerable insight by first studying the underlying theoretical concepts and then applying them in a practical, hands-on manner.