VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

SUMMARY of MASTER'S THESIS PROPOSAL

Building an On-Premises Data Lakehouse for Big Data Storage and Analysis

Advisor: Dr. Phan Trong Nhan
Author: Dinh Thanh Phong - 2270243

Ho Chi Minh City, June 2024

Contents

1 Advantages of a Data Lakehouse

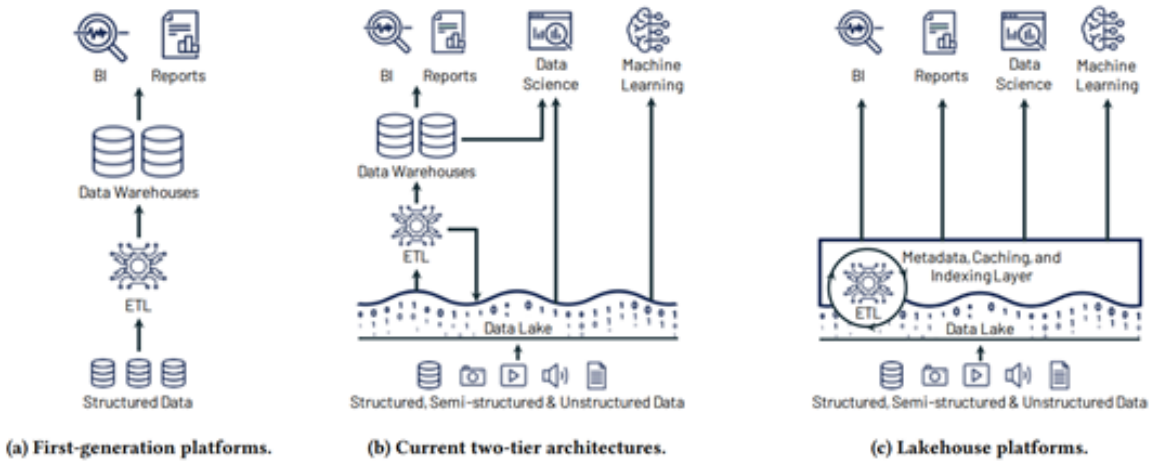

Figure 1: Comparing Data Lakehouse Architecture with Existing Data Architectures

Benefits of the Data Lakehouse: *Simplicity, Flexibility, and Low Cost*

The data lakehouse architecture offers several benefits:

	Data Warehouse	Data Lake	Lakehouse
Data	Rational data from transactional systems, operational databases & business applications	All data including structured, semi-structured, and unstructured.	Query Every kind of data Image, audio, video, and others.
Data Access	SQL Only	Open API, SQL, Python	Open API, SQL, Python
Data Format	Proprietary Format	Open format	Open Format
Governance	Fine-grained Security	Weak Governance and security	Fine-grained Security
Reliability	High with ACID Transactions	Low Quality- Data Swamps	High with ACID Transactions.
Performance	Fast query results using local language	Faster query results, decoupling of computing and storage	Faster and deeper insights without data movement.
Scalability	Scaling becomes expensive	Low cost of scaling Regardless of the data-type	Low cost of scaling regardless of the data-type

Figure 2: Comparison of Features: Data Warehouse, Data Lake, and Data Lakehouse

Unified System: By integrating data warehouse features directly into the low-cost storage of a data

lake, it eliminates the need for separate data warehouse layers.

Simplified Access: Data teams can access and utilize data more efficiently without navigating multiple systems.

Updated Data: Ensures that the most complete and up-to-date data is available for data science, machine learning, and business analytics.

2 Several existing cloud Data Lakehouse Architecture

2.1 Reference Lakehouse Architecture by Microsoft Azure

Organization of the Reference Architectures

The reference architecture is structured around the workflows of Source, Ingest, Transform, Query and Process, Serve, Analysis, and Storage:

Source

The architecture distinguishes between semi-structured and unstructured data (sensors and IoT, media, files/logs) and structured data (RDBMS, business applications). SQL sources (RDBMS) can also be integrated into the lakehouse and Unity Catalog without ETL through lakehouse federation. Additionally, data can be loaded from other cloud providers.

Ingest

Data can be ingested into the lakehouse via batch or streaming:

- Files transferred to cloud storage can be directly loaded using Databricks Auto Loader.
- For batch ingestion of data from enterprise applications into Delta Lake, the Databricks lakehouse relies on partner ingestion tools with specific adapters for these systems.
- Streaming events can be ingested directly from streaming systems like Kafka using Databricks Structured Streaming. Streaming sources can include sensors, IoT, or change data capture processes.

Storage

Data is typically stored in cloud storage systems where ETL pipelines use the medallion architecture to organize and store data as Delta files/tables.

Transform and Query and Process

The Databricks lakehouse utilizes its engines, Apache Spark and Photon, for all transformations and queries. Due to its simplicity, the declarative framework DLT (Delta Live Tables) is a good option for building reliable, maintainable, and testable data processing pipelines. Supported by Apache Spark and Photon, the Databricks Data Intelligence Platform supports both types of workloads: SQL queries through SQL warehouses and SQL, Python, and Scala workloads through workspace clusters. For data science (ML Modeling and Gen AI), the Databricks AI and Machine Learning platform provides specialized ML runtimes for AutoML and coding ML jobs. All data science workflows and MLOps are best supported by MLflow.

Serve

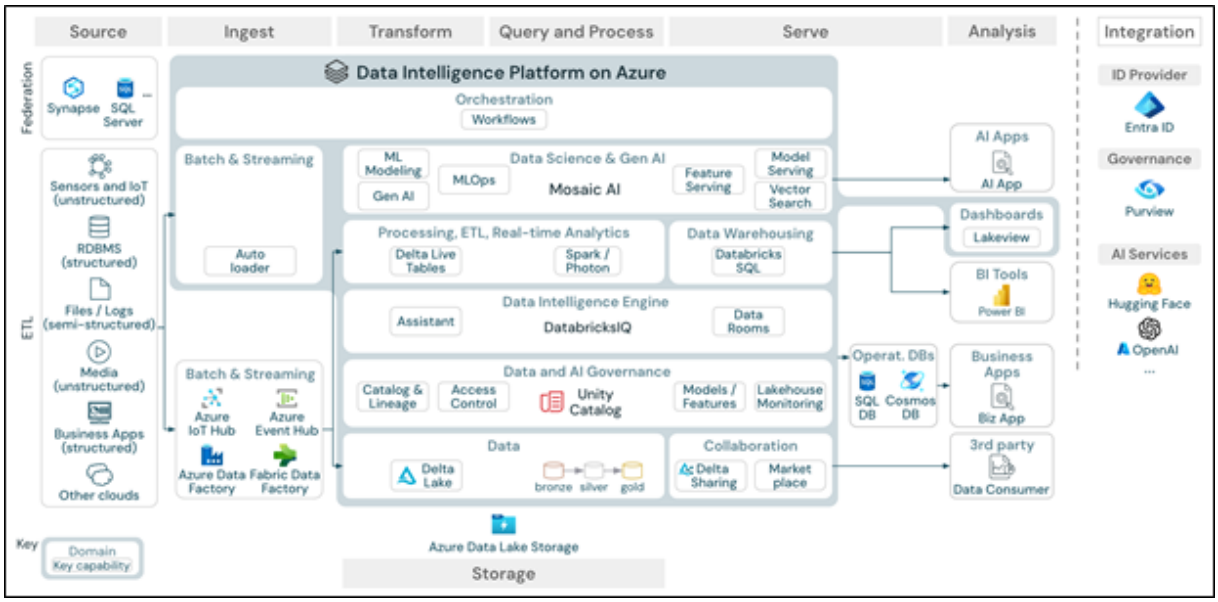

Figure 3: Data Lakehouse Architecture proposed by Microsoft using Azure Data Lake Storage

For DWH and BI use cases, the Databricks lakehouse offers Databricks SQL, a data warehouse supported by SQL warehouses and serverless SQL warehouses. For machine learning, model serving provides enterprise-level, large-scale, real-time model serving capabilities hosted in the Databricks control plane. Operational databases: External systems, such as operational databases, can be used to store and serve the final data products for user applications. Collaboration: Business partners gain secure access to the data they need through Delta Sharing. Based on Delta Sharing, the Databricks Marketplace is an open forum for exchanging data products.

Analysis

The final business applications reside within this workflow. Examples include custom clients like AI applications connected to Databricks Model Serving for real-time inference or applications accessing data pushed from the lakehouse to an operational database. For BI use cases, analysts typically use BI tools to access the data warehouse. SQL developers can also use the Databricks SQL Editor (not shown in the diagram) to query and create dashboards. The Data Intelligence Platform also provides dashboards for building data visualizations and sharing insights.

2.2 Reference Lake House Architecture on AWS

The diagram below illustrates the reference Lake House architecture on AWS.

Data Ingestion Layer

The ingestion layer in the AWS Lake House architecture uses various AWS services specifically designed to ingest data from different sources into the storage layer. The key services include:

- **Operational Database Sources**: AWS Data Migration Service (DMS) imports and replicates data from operational RDBMS and NoSQL databases to Amazon S3 or Amazon Redshift.

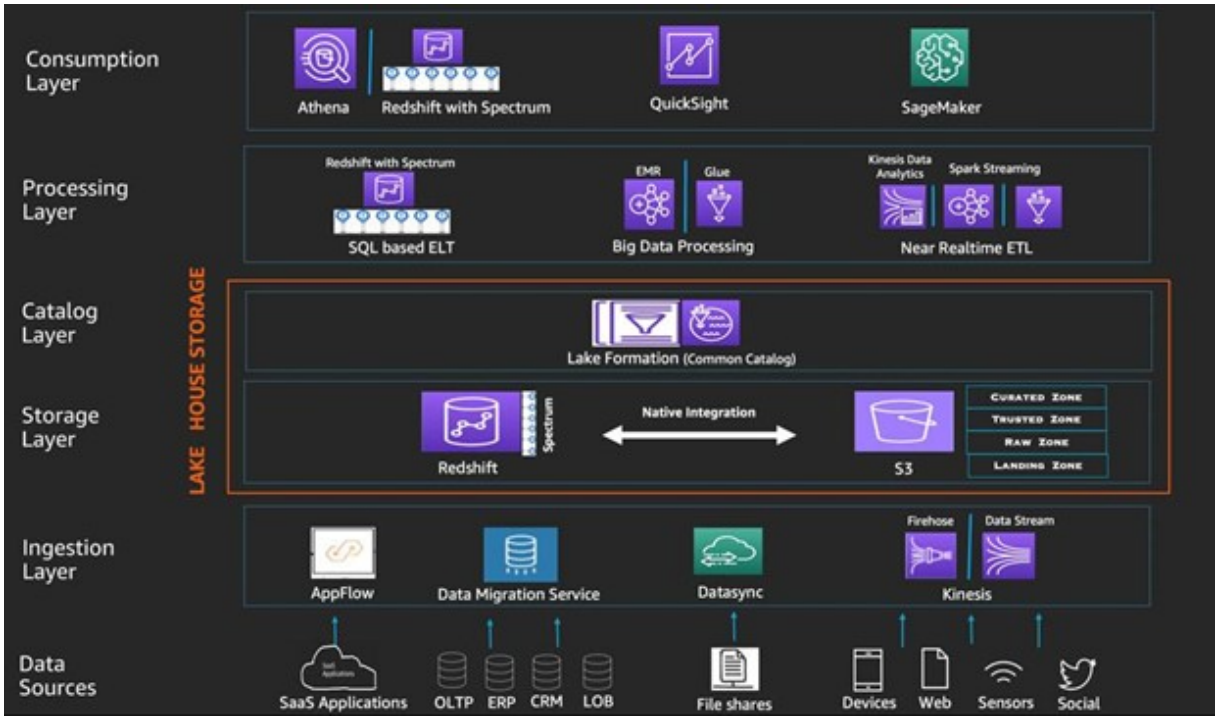

Figure 4: Proposed Data Lakehouse Architecture by AWS

- **SaaS Applications**: Amazon AppFlow ingests data from SaaS applications like Salesforce and Google Analytics into Amazon S3 or Redshift.
- **File Shares**: AWS DataSync transfers data from NAS devices to Amazon S3, handling large volumes and ensuring data integrity.
- **Streaming Data Sources**: Amazon Kinesis Data Firehose processes and transfers streaming data to Amazon S3 or Redshift.

Lake House Storage Layer

AWS Lake House integrates Amazon Redshift and Amazon S3 for storage, supporting both structured data in Redshift and diverse data types in S3. The key components include:

- **Amazon Redshift Spectrum**: Allows Redshift to query data stored in S3 using a unified SQL interface, supporting large-scale analytics without data movement.
- **Storage Zones in S3**: Data is organized into landing, raw, trusted, and curated zones to efficiently manage the lifecycle and stages of data processing.

Data Processing Layer

The processing layer includes various AWS services designed for different data processing needs:

- **SQL-based ELT**: Amazon Redshift and Redshift Spectrum support robust ELT pipelines for transforming structured data using SQL.

- **Big Data Processing**: AWS Glue and Amazon EMR provide large-scale ETL capabilities and big data processing using Apache Spark.
- **Near-real-time ETL**: Services like Kinesis Data Analytics and Spark Streaming (on AWS Glue or EMR) enable near-real-time data processing.

Data Consumption Layer

The data consumption layer democratizes data access across the organization through:

- **Interactive SQL**: Amazon Redshift (with Spectrum) and Athena allow interactive SQL queries on data in S3 and Redshift.
- **Machine Learning**: Amazon SageMaker provides tools for data exploration, feature engineering, model training, and deployment integrated with the Lake House storage.
- **Business Intelligence**: Amazon QuickSight offers serverless BI capabilities to create interactive dashboards and integrates with ML models from SageMaker.

Central Data Catalog and Metadata Management

AWS Lake Formation serves as the central catalog for metadata management across the entire data lake and data warehouse, supporting schema-on-read, granular permissions, and integration with various components of the processing and consumption layers.

3 The Need for On-Premises Data Lakehouse

We acknowledge the tremendous advantages and utilities provided by Public Cloud platforms. However, building an on-premises Data Lakehouse architecture not only optimizes data management efficiency but also ensures compliance with data security and privacy regulations. Specifically, this helps businesses meet strict requirements for the protection of citizen data in various countries, such as GDPR in Europe, HIPAA in the healthcare sector in the United States, as well as cybersecurity regulations in Vietnam and China.

At the same time, the concept of a Data Lakehouse is still relatively new in recent years, with no specific standards and limited research applied to specific fields. For instance, in the financial sector, development processes are still in the research phase. Particularly in Vietnam, where businesses are still struggling to transition from data warehouses to data lakes, the Data Lakehouse concept remains fairly novel.

Therefore, researching and proposing an on-premises Data Lakehouse architecture for businesses that require high security to ensure both efficiency and safety, without relying on external parties, is a highly urgent and practical need that many companies are currently concerned with and carefully considering.

4 Proposed On-Premises Data Lakehouse Architecture

To build a fully functional data lakehouse pipeline as mentioned above, the author proposes a solution with the following key components and specific technologies:

4.1 Proposed Architecture

- **Ingestion Layer:** The layer for ingesting data from various data sources.
- **Storage Layer:** The layer for storing data with support for structured, semi-structured, and unstructured data.
- **Catalog Layer:** This layer manages metadata and schema enforcement for structured and semi-structured data, ensuring data consistency and discoverability across the lakehouse.
- **Processing Layer:** The layer for processing data for ETL tasks, data analysis, and machine learning.
- **Serving Layer:** The layer for serving data to BI tools, reporting, and analytics.
- **Monitoring and Management Layer:** The layer for monitoring and managing to ensure the system operates stably.

4.2 Technologies Used

- **Ingestion Layer:**
  - Apache Kafka: This technology is used to ingest real-time data from various sources such as log files, databases, and API services. Kafka provides high fault tolerance and reliability in data transmission.
  - The data ingestion process involves collecting and importing data into the data lakehouse. This process can be automated or done manually using custom scripts or command-line tools, such as CRM applications, relational databases, and NoSQL databases.
- **Storage Layer:**
  - Hadoop HDFS (Hadoop Distributed File System): HDFS provides flexible and scalable storage, supporting structured, semi-structured, and unstructured data.
  - Delta Lake: Built on top of Apache Spark, Delta Lake provides storage with ACID support, time travel (the ability to revert to previous data states), and efficient metadata management.
- **Catalog Layer:**
  - Hive Metastore: The Hive Metastore is on top of Hadoop HDFS and Delta Lake Table. It ensures data consistency, facilitates data discovery, and integrates seamlessly with other components in the data lakehouse.

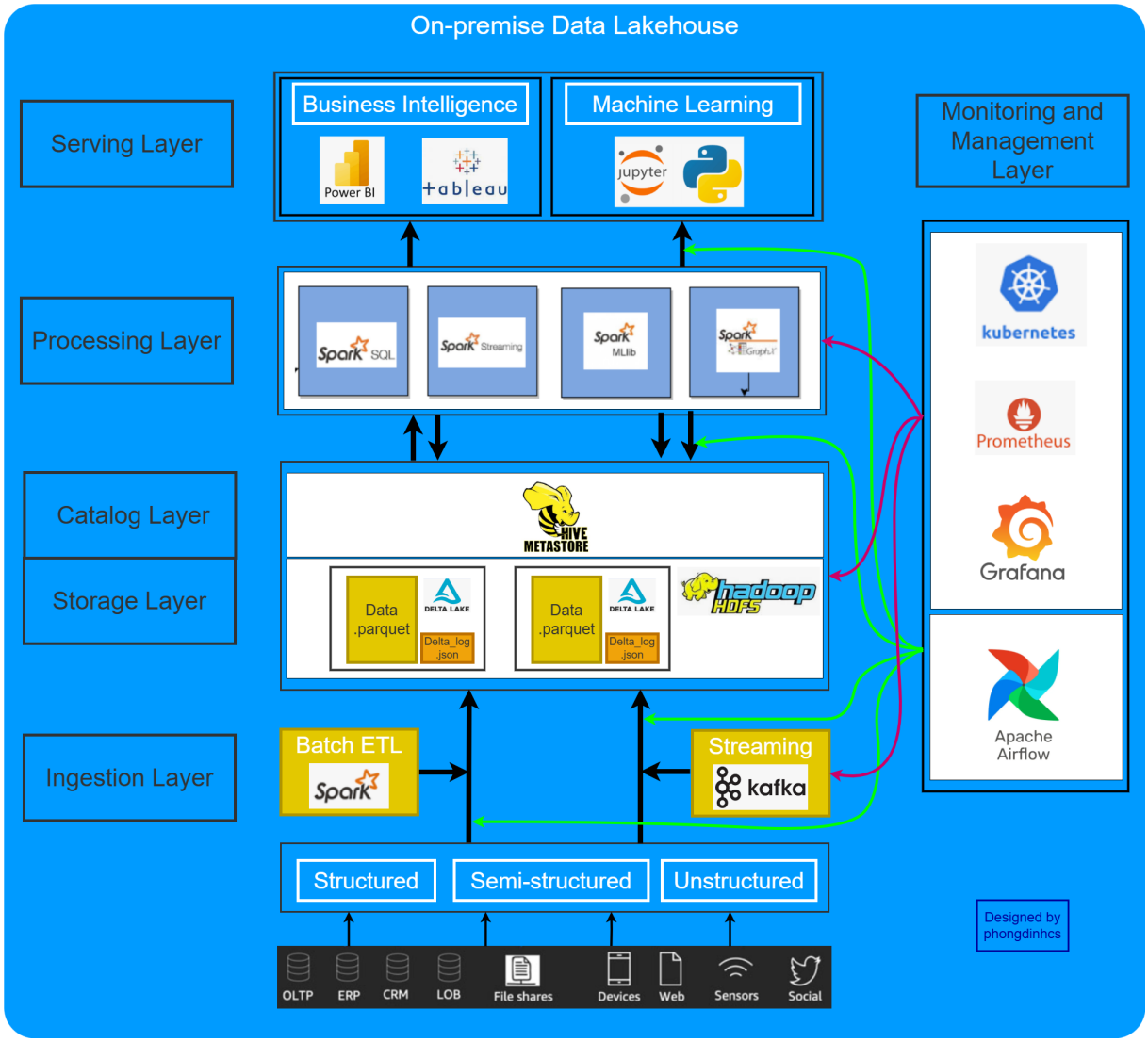

Figure 5: Proposed On-Premise Data Lakehouse Architecture

- **Processing Layer:**
  - Apache Spark: Apache Spark is used to perform ETL (Extract, Transform, Load) tasks, data analysis, and machine learning applications. Spark offers powerful and efficient parallel data processing.
- **Serving Layer:**
  - Power BI: Power BI allows access and visualization of data from the data lakehouse, providing easy-to-understand reports and dashboards for end users.
  - Jupyter Notebook: This is an integrated environment for data scientists to explore, analyze data, and develop machine learning models.
- **Monitoring and Management Layer:**

- Kubernetes with Prometheus and Grafana: Prometheus and Grafana are deployed on Kubernetes to manage and deploy containers. Prometheus is used to monitor system performance metrics, while Grafana displays these metrics in charts.
- Apache Airflow: Used to manage and orchestrate ETL pipelines, ensuring tasks are executed in the correct order and at the specified times.

5 Demo PhongDinhCS-data_lakehouse has been built and published on GitHub

5.1 Example Scenario Applying Data Lakehouse Architecture

In this report, I present an example of real-time data updates in the stock finance domain. The model includes four information tables:

- Stock Symbol Table: This table is pre-built and directly stored as a Delta Table within HDFS.
- Current Stock Price Table: Updated with simulated data from a PostgreSQL source on an online server. PySpark updates this table every 10 seconds and writes the data to a Delta Table in HDFS.
- Transaction Table: PySpark records updates to the Current Stock Price Table, capturing historical stock prices.
- Market Information Update Table Based on Electronic News Reports: This table uses a Python script to crawl data from online news articles. A Kafka producer sends this data to a Kafka topic, and a Kafka consumer retrieves the data and writes updates to an HTML store containing two pieces of information: timestamp and the raw HTML. The purpose of crawling this data in its raw form is to assess early risk if a stock is frequently mentioned in the news or to store these HTML records for future machine learning or deep learning purposes.

5.2 GitHub Repository PhongDinhCS and Docker-compose

The demo architecture has been built and is available on GitHub:
https://github.com/PhongDinhCS/PhongDinhCS-data_lakehouse

In this section, I will describe the functionality of the complete-version-1.0 of the data lakehouse architecture, available in the following GitHub repository:
https://github.com/PhongDinhCS/PhongDinhCS-data_lakehouse/tree/compelete-version-1.0

In the ‘complete-version-1.0‘ repository, I have set up several containers, including Kafka, Hadoop, Hive Metastore, PostgreSQL, and Delta Spark, which have been successfully tested with Flask API, Jupyter Notebook, and Power BI. All containers were built from official images on Docker Hub.

Image versions:

- apache/kafka:3.8.0
- apache/hadoop:3
- postgres:11
- apache/hive:3.1.3
- deltaio/delta-docker:0.8.1_2.3.0

The docker-compose.yml file includes:

```
version: '3'

services:
  kafka:
    container_name: kafka
    restart: unless-stopped
    image: apache/kafka:3.8.0
    ports:
      - 9092:9092
    volumes:
      - ./kafka/config:/mnt/shared/config
      - kafka-bin:/opt/kafka
    networks:
      lakehouse:
        ipv4_address: 172.18.0.99

  namenode:
    container_name: namenode
    restart: unless-stopped
    image: apache/hadoop:3
    hostname: namenode
    command: ["hdfs", "namenode"]
    ports:
      - 9870:9870
      - 8020:8020
      - 9000:9000
    env_file:
      - ./hadoop/config
    volumes:
      - ./hadoop/dfs/name:/hadoop/dfs/name
      - hadoop-bin:/opt/hadoop
    networks:
      lakehouse:
        ipv4_address: 172.18.0.98

  datanode1:
```

```
    container_name: datanode1
    restart: unless-stopped
    image: apache/hadoop:3
    command: ["hdfs", "datanode"]
    env_file:
      - ./hadoop/config
    volumes:
      - ./hadoop/dfs/data1:/hadoop/dfs/data
    networks:
      lakehouse:
        ipv4_address: 172.18.0.97

  datanode2:
    container_name: datanode2
    restart: unless-stopped
    image: apache/hadoop:3
    command: ["hdfs", "datanode"]
    env_file:
      - ./hadoop/config
    volumes:
      - ./hadoop/dfs/data2:/hadoop/dfs/data
    networks:
      lakehouse:
        ipv4_address: 172.18.0.96

  resourcemanager:
    container_name: resourcemanager
    restart: unless-stopped
    image: apache/hadoop:3
    hostname: resourcemanager
    command: ["yarn", "resourcemanager"]
    ports:
        - 8088:8088
    env_file:
      - ./hadoop/config
    networks:
      lakehouse:
        ipv4_address: 172.18.0.94
    depends_on:
      - namenode
      - datanode1
      - datanode2

  nodemanager:
    container_name: nodemanager
    restart: unless-stopped
    image: apache/hadoop:3
    command: ["yarn", "nodemanager"]
    env_file:
```

```
    - ./hadoop/config
  networks:
    lakehouse:
      ipv4_address: 172.18.0.93
  depends_on:
    - resourcemanager

postgres:
  image: postgres:11
  restart: unless-stopped
  container_name: postgres
  hostname: postgres
  environment:
    POSTGRES_DB: 'metastore_db'
    POSTGRES_USER: 'hadoop'
    POSTGRES_PASSWORD: 'hadoop'
  ports:
    - '5432:5432'
  volumes:
    - postgresql:/var/lib/postgresql
  networks:
    lakehouse:
      ipv4_address: 172.18.0.91

metastore:
  image: apache/hive:3.1.3
  depends_on:
    - postgres
  restart: unless-stopped
  container_name: metastore
  hostname: metastore
  environment:
    HIVE_CONF_DIR: /opt/hive/conf
    DB_DRIVER: postgres
    IS_RESUME: 'true'
    SERVICE_NAME: 'metastore'
  ports:
      - '9083:9083'
  volumes:
      - ./hive/conf/hive-site.xml:/opt/hive/conf/hive-site.xml
      - ./hadoop/core-site.xml:/opt/hive/conf/core-site.xml
      - ./hadoop/hdfs-site.xml:/opt/hive/conf/hdfs-site.xml
      - ./hadoop/yarn-site.xml:/opt/hive/conf/yarn-site.xml
      - type: bind
        source: ./hive/postgresql-42.2.27.jre6.jar
        target: /opt/hive/lib/postgres.jar
  networks:
    lakehouse:
      ipv4_address: 172.18.0.90
```

```
135
136   hiveserver2:
137     image: apache/hive:3.1.3
138     depends_on:
139       - metastore
140     restart: unless-stopped
141     container_name: hiveserver2
142     environment:
143       HIVE_SERVER2_THRIFT_PORT: 10000
144       IS_RESUME: 'true'
145       SERVICE_NAME: 'hiveserver2'
146       HIVE_CONF_DIR: /opt/hive/conf
147     ports:
148       - '10000:10000'
149       - '10002:10002'
150     volumes:
151       - ./hive/conf/hive-site.xml:/opt/hive/conf/hive-site.xml
152       - ./hadoop/core-site.xml:/opt/hive/conf/core-site.xml
153       - ./hadoop/hdfs-site.xml:/opt/hive/conf/hdfs-site.xml
154       - ./hadoop/yarn-site.xml:/opt/hive/conf/yarn-site.xml
155     networks:
156       lakehouse:
157         ipv4_address: 172.18.0.89
158
159   delta-spark:
160     container_name: delta-spark
161     restart: unless-stopped
162     image: deltaio/delta-docker:0.8.1_2.3.0
163     entrypoint:
164       - bash
165       - -c
166       - |
167         if ! hdfs dfs -test -d hdfs://namenode:8020/lakehouse; then
168           hdfs dfs -mkdir hdfs://namenode:8020/lakehouse
169           hdfs dfs -chmod 777 hdfs://namenode:8020/lakehouse
170         fi
171         tail -f /dev/null
172     volumes:
173       - hadoop-bin:/opt/hadoop
174       - kafka-bin:/opt/kafka
175       - ./delta-spark/volume:/opt/spark/work-dir/volume
176       - ./hive/conf/hive-site.xml:/opt/spark/conf/hive-site.xml
177       - ./hadoop/core-site.xml:/opt/spark/conf/core-site.xml
178       - ./hadoop/hdfs-site.xml:/opt/spark/conf/hdfs-site.xml
179       - ./hadoop/yarn-site.xml:/opt/spark/conf/yarn-site.xml
180     environment:
181       - DELTA_PACKAGE_VERSION=delta-core_2.12:2.3.0
182       - SPARK_CONF_DIR=/opt/spark/conf
183       - HADOOP_HOME=/opt/hadoop
```

```
      - HADOOP_CONF_DIR=/opt/hadoop/etc/hadoop
      - PATH=/opt/spark/bin:/opt/hadoop/bin:/opt/kafka/bin:/home/NBuser/.local/bin:/home/NBuser/.local/lib/python3.9/site-packages:$PATH
      - HADOOP_USER_NAME=hadoop
      - SPARK_MODE=master
      - SPARK_SQL_CATALOG_IMPLEMENTATION=hive
    networks:
      lakehouse:
        ipv4_address: 172.18.0.92
    ports:
      - "8888:8888"  # Jupyter Notebook
      - "7077:7077"  # Spark Master
      - "4040:4040"  # Spark Web UI
      - "5000:5000"  # Flask API
    depends_on:
      - namenode

networks:
  lakehouse:
    name: lakehouse
    driver: bridge
    ipam:
      config:
        - subnet: 172.18.0.0/16
          gateway: 172.18.0.1

volumes:
  hadoop-bin:
  kafka-bin:
  postgresql:
```

5.3 Components and Trial Running of PhongDinhCS-data_Lakehouse

The setup and execution of the PhongDinhCS-data_lakehouse containers on Docker Desktop running on Windows 10, using Windows Subsystem for Linux (WSL) for compatibility with the Linux-based containers. It is also can run on Linux Ubuntu 22.04 LTS within a VMware virtual machine. The environment enables the orchestration of multiple containers, such as Kafka, Hadoop, and Delta Spark.

The Hadoop information can be accessed through a web browser interface. It showcases the Hadoop Distributed File System (HDFS) and ResourceManager web UI, which are used to monitor the cluster's storage and job execution status.

The HiveServer2 information being checked via a web browser. It highlights the Hive Web UI, where users can manage and monitor Hive queries and configurations, demonstrating Hive's integration within the PhongDinhCS-data_lakehouse setup.

The Delta table is stored in Parquet format, accompanied by a delta_log file. Both are stored in the Hadoop Distributed File System (HDFS), showcasing Delta Lake's capabilities in managing data storage with ACID transaction properties.

Jupyter Notebook within the Delta-Spark container to test Kafka's functionality. The notebook environment allows interactive development and testing of Spark jobs, including Kafka data processing, within the PhongDinhCS-data_lakehouse framework.

Flask is used within the Delta-Spark container to implement a Web API. The API facilitates easy data sharing by exposing data stored in Delta tables through a RESTful service, which can be consumed by various client applications.

Power BI is used to retrieve data from the Web API and generate reports. It highlights the integration of Power BI with the PhongDinhCS-data_lakehouse project, showcasing its ability to visualize and analyze data sourced from Delta Lake tables via the Flask API.

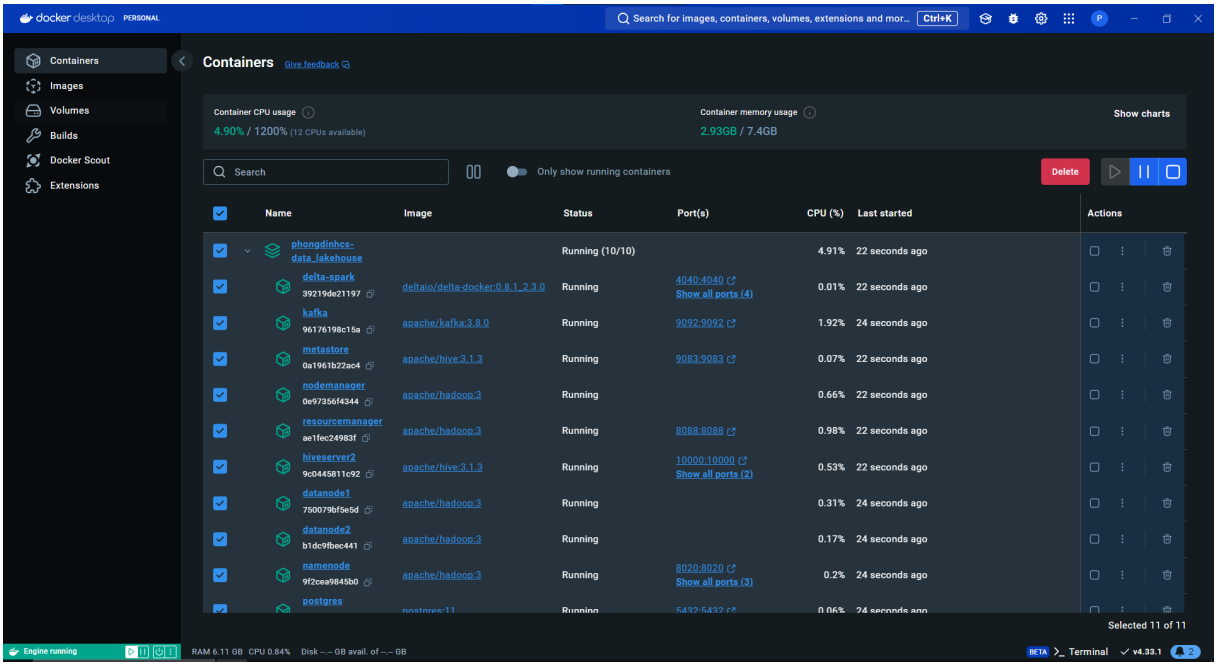

Figure 6: Running PhongDinhCS-data_lakehouse containers on Docker Desktop Windows 10 with WSL

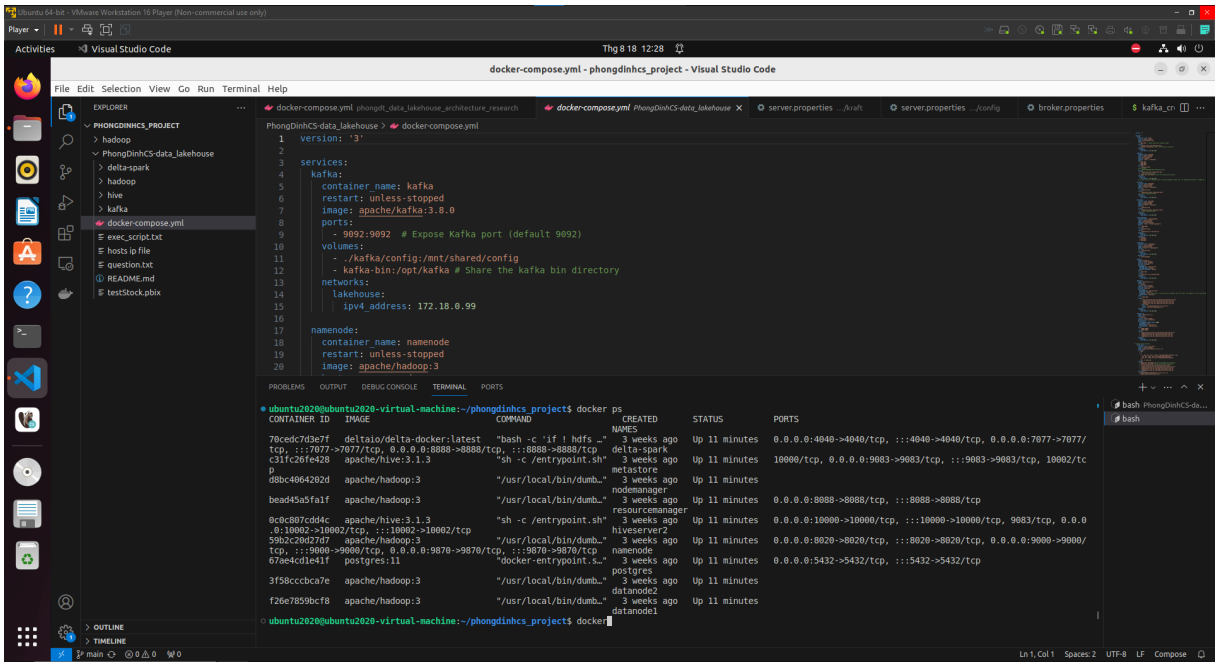

Figure 7: Running PhongDinhCS-data_lakehouse containers on Linux Ubuntu 22.04 LTS in VMware

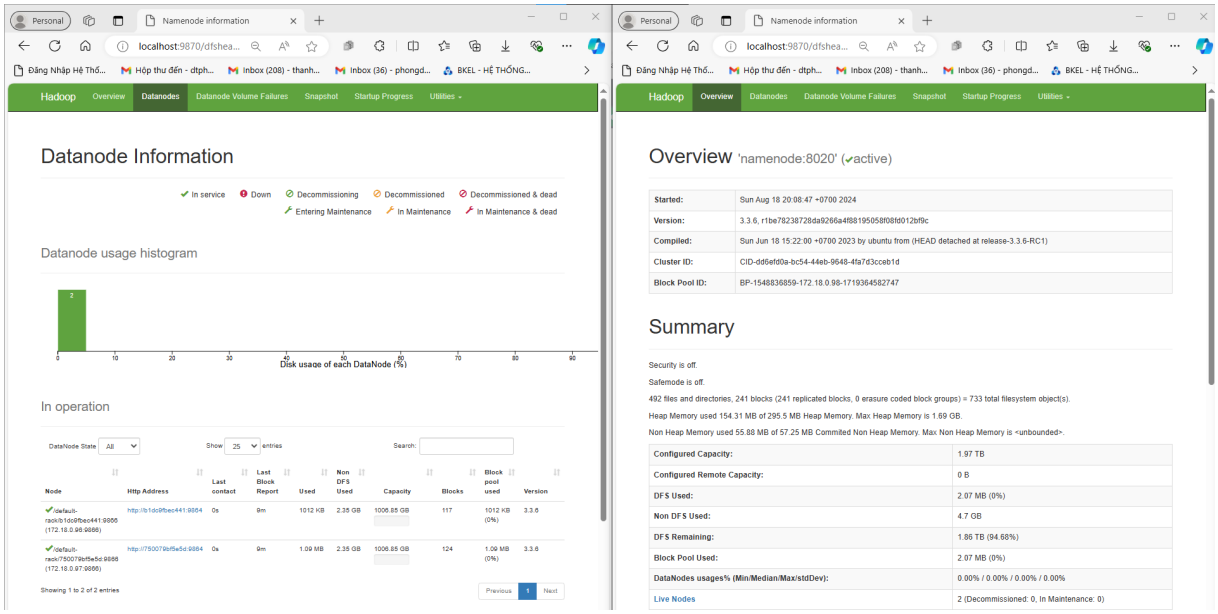

Figure 8: Checking Hadoop information from browser

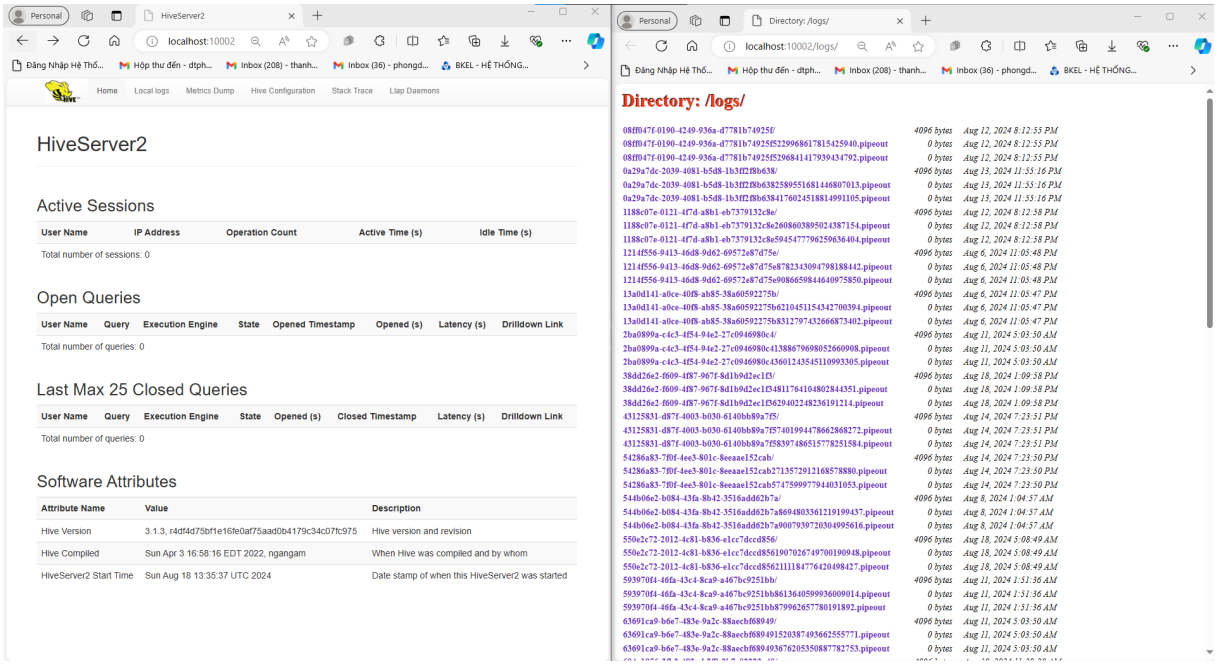

Figure 9: Checking HiveServer2 information from browser

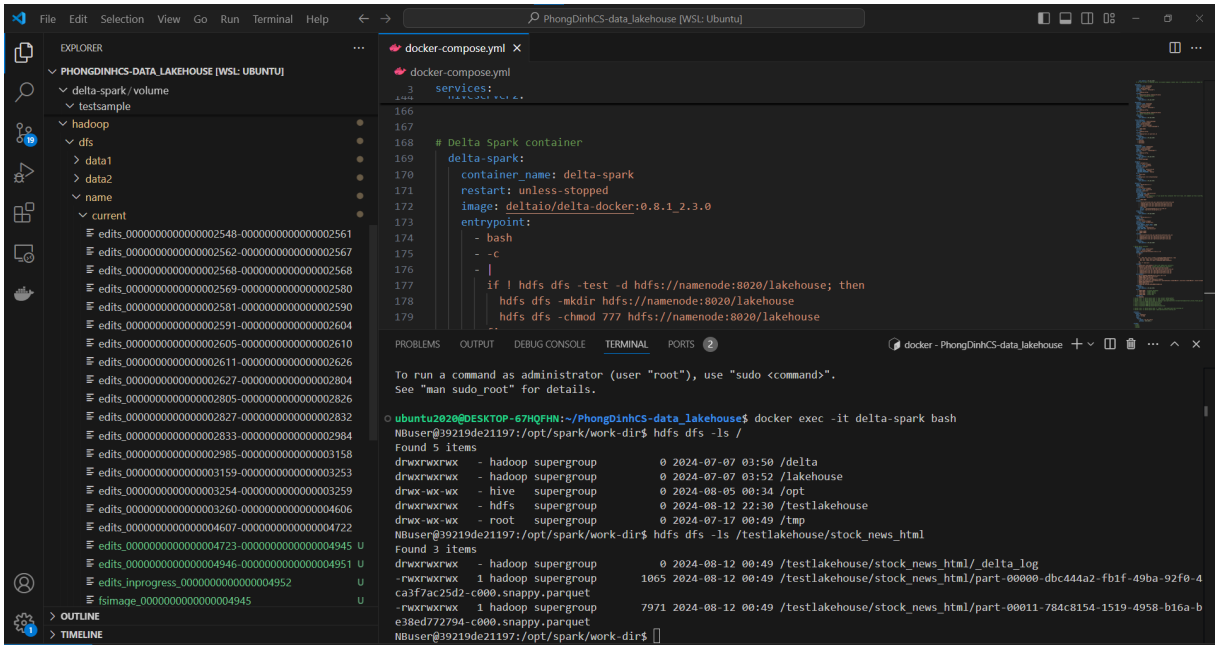

Figure 10: Data in the Delta table is stored in Parquet format along with the delta_log file and stored in HDFS

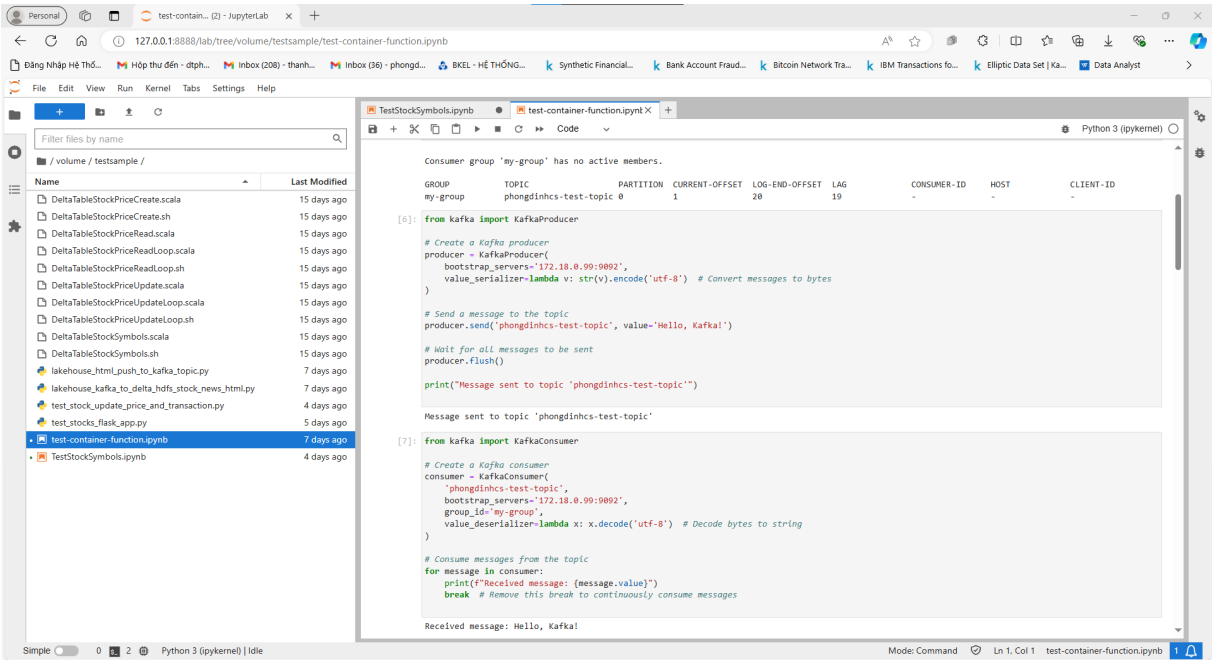

Figure 11: Running Jupyter Notebook from the Delta-Spark container to test Kafka functionality

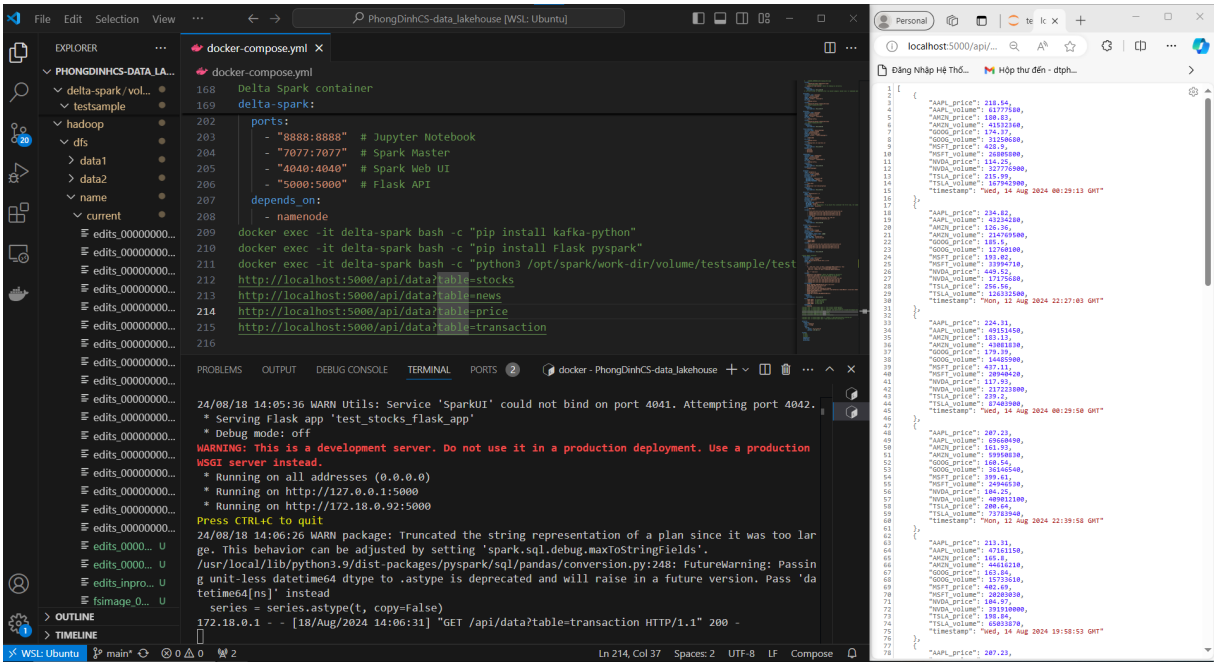

Figure 12: Using Flask in the Delta-Spark container to implement a Web API for easy data sharing

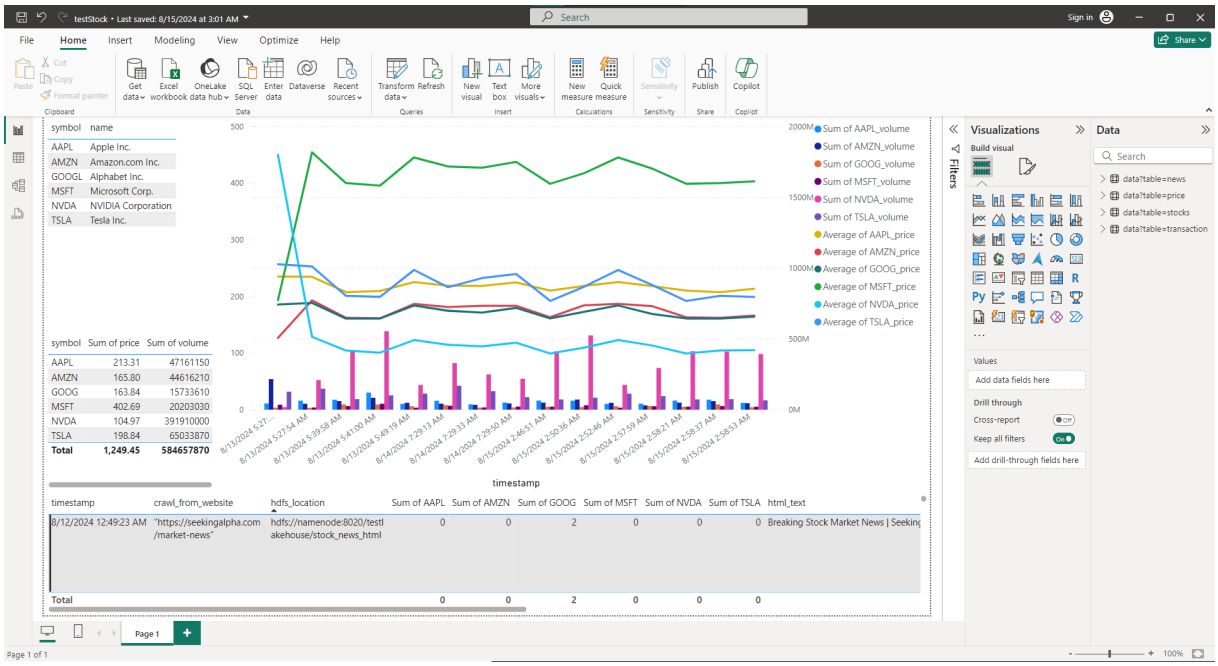

Figure 13: Using Power BI to retrieve data from the Web API and generate reports

6 Next Steps for Completing the On-Premises Data Lakehouse Architecture

In the current complete-version-1.0 repository, I have completed the setup and operation of the ingestion layer, storage layer, catalog layer, processing layer, and serving layer on the Docker and Docker Compose platform.

The next goal is to build a Monitoring and Management Layer and finalize the architecture on Kubernetes with Prometheus and Grafana to facilitate system activity control and resource monitoring.

Additionally, it is necessary to use Apache Airflow to manage automated pipelines, ensuring accurate and stable operational processes.

Finally, the On-Premises Data Lakehouse Architecture also needs a solution to integrate with a Public Cloud Data Lakehouse, such as Databricks on Microsoft Azure, to create a hybrid platform. This will ensure that the data lakehouse maintains security for personal information while providing cost-effective solutions for less sensitive data.

7 Instructions for Executing Code Commands

Setting Up Your Environment Ensure you have Docker and Docker Compose installed on your system. Clone the repository from GitHub

```
git clone https://github.com/PhongDinhCS/PhongDinhCS-data_lakehouse/tree/compelete-version-1.0
cd PhongDinhCS-data_lakehouse
```

Start all containers

```
#Build containers
docker-compose up -d

#Check the running container
docker ps
```

Accesss delta-spark container then start Jupyter Notebook

```
docker exec -it delta-spark bash -c "chmod +x /opt/spark/work-dir/startup.sh"
docker exec -it delta-spark bash "/opt/spark/work-dir/startup.sh"
```

Now you can test to access Jupyter Notebook via the web browser by the url that delta-spark show for you, it should start by http://127.0.0.1:8888/

Then you can access to the TestStockSymbols.ipynb at this location
http://127.0.0.1:8888/lab/tree/volume/testsample/test-container-function.ipynb
in this Notebook, I prepared some code to test the connection between the containers such as:

```
!curl -X GET http://172.18.0.99:9092
!/opt/kafka/bin/kafka-topics.sh --list --bootstrap-server 172.18.0.99:9092
!/opt/kafka/bin/kafka-topics.sh --create --topic phongdinhcs-test-topic --bootstrap-server 172.18.0.99:9092 --partitions 1 --replication-factor 1
!/opt/kafka/bin/kafka-topics.sh --list --bootstrap-server 172.18.0.99:9092
!/opt/kafka/bin/kafka-consumer-groups.sh --bootstrap-server 172.18.0.99:9092 --describe --group my-group
```

```

from kafka import KafkaProducer

# Create a Kafka producer
producer = KafkaProducer(
    bootstrap_servers='172.18.0.99:9092',
    value_serializer=lambda v: str(v).encode('utf-8')  # Convert messages to bytes
)

# Send a message to the topic
producer.send('phongdinhcs-test-topic', value='Hello, Kafka!')

# Wait for all messages to be sent
producer.flush()

print("Message sent to topic 'phongdinhcs-test-topic'")
```

```

from kafka import KafkaConsumer

# Create a Kafka consumer
consumer = KafkaConsumer(
    'phongdinhcs-test-topic',
    bootstrap_servers='172.18.0.99:9092',
    group_id='my-group',
    value_deserializer=lambda x: x.decode('utf-8')  # Decode bytes to string
)

# Consume messages from the topic
for message in consumer:
    print(f"Received message: {message.value}")
    break  # Remove this break to continuously consume messages

# Show tables in the current database

spark.sql("SHOW TABLES").show()

# Describe all information in metastore
from pyspark.sql import SparkSession

# Initialize Spark session
spark = SparkSession.builder \
    .appName("Metastore Metadata") \
    .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \
    .config("spark.sql.catalog.spark_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog") \
    .getOrCreate()

# Show all databases
databases = spark.sql("SHOW DATABASES").collect()

# Loop through each database
for db in databases:
    db_name = db[0]  # Access the first column (database name)
    spark.sql(f"USE {db_name}")
    tables = spark.sql("SHOW TABLES").collect()
    print(f"\nDatabase: {db_name}\n" + "-" * 40)

    # Loop through each table
    for table in tables:
        table_name = table[1]  # Access the second column (table name)

        # Count the number of rows in the table
        row_count = spark.sql(f"SELECT COUNT(*) AS count FROM {table_name}").collect()[0][0]

```

```
        print(f"Table: {table_name} (Rows: {row_count})")
        print("-" * 40)

        # Describe each table
        description = spark.sql(f"DESCRIBE TABLE {table_name}")
        schema = description.schema
        print(f"{'Column Name':<15} | {'Data Type':<10} | {'Nullable':<10} | {'Comment':<15}")
        print("-" * 50)

        for row in description.collect():
            col_name = row[0]          # Column name
            data_type = row[1]         # Data type
            nullable = row[2]          # Exact nullable status

            # Handle optional comment field if present
            if len(row) > 3:
                comment = row[3]       # Column comment (if available)
            else:
                comment = "N/A"        # Default if no comment

            # Print with better formatting
            print(f"{col_name:<15} | {data_type:<10} | {nullable:<10} | {comment:<15}")
        print("-" * 50)
```

```

from pyspark.sql import SparkSession

# Initialize Spark session with Delta configurations
spark = SparkSession.builder \
    .appName("TableDetailMetadata") \
    .config("spark.jars.packages", "io.delta:delta-core_2.12:2.3.0") \
    .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \
    .config("spark.sql.catalog.spark_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog") \
    .getOrCreate()

# Function to get table details
def get_table_details(database_name, table_name):
    try:
        details = spark.sql(f"DESCRIBE DETAIL {database_name}.{table_name}").collect()
        if details:
            detail_info = details[0].asDict()
            return detail_info
        else:
            return {"error": "Details not found"}
    except Exception as e:
        return {"error": str(e)}

# Get all databases
```

```
databases = spark.sql("SHOW DATABASES").collect()

# Dictionary to hold metadata for all tables
all_tables_metadata = {}

for db in databases:
    db_name = db.namespace
    spark.sql(f"USE {db_name}")

    # Get all tables in the database
    tables = spark.sql("SHOW TABLES").collect()
    table_metadata = {}

    for table in tables:
        table_name = table.tableName
        print(f"Fetching details for table: {table_name} in database: {db_name}")

        # Get table details
        details = get_table_details(db_name, table_name)
        table_metadata[table_name] = details

    all_tables_metadata[db_name] = table_metadata

# Display the metadata
for db_name, tables in all_tables_metadata.items():
    print(f"\nDatabase: {db_name}")
    for table_name, details in tables.items():
        print(f"\nTable: {table_name}")
        for key, value in details.items():
            print(f"{key}: {value}")
```

Finally, you can start the API Flask Pyspark by this code:

```
docker exec -it delta-spark bash -c "pip install kafka-python"
docker exec -it delta-spark bash -c "pip install Flask pyspark"
docker exec -it delta-spark bash -c "python3 /opt/spark/work-dir/volume/testsample/
    test_stocks_flask_app.py"
```

Now you can access to API from the web browser:

http://localhost:5000/api/data?table=stocks

http://localhost:5000/api/data?table=news

http://localhost:5000/api/data?table=price

http://localhost:5000/api/data?table=transaction

Additionally, you can try to update simulated data in the Delta Table by these code:

```
docker exec -it delta-spark bash -c "python3 /opt/spark/work-dir/volume/testsample/
    lakehouse_kafka_to_delta_hdfs_stock_news_html.py"
docker exec -it delta-spark bash -c "python3 /opt/spark/work-dir/volume/testsample/
    lakehouse_html_push_to_kafka_topic.py"
```

```
docker exec -it delta-spark bash -c "python3 /opt/spark/work-dir/volume/testsample/
    test_stock_update_price_and_transaction.py"dir/volume/testsample/
    test_stocks_flask_app.py"
```

Successfully!!!