Lab 5 Report

Colab link:
https://colab.research.google.com/drive/1u-4qwfKifT4ZoAoQC6SwMm9F5WRQULgE#scrollTo=Pa07674L6D5q

Lambda architecture : It is a data processing architecture designed to handle massive quantities of data by taking advantage of both batch and stream-processing methods

Layer representation: Each layer in the Lambda architecture serves a unique purpose.

1. The **Batch Layer** is responsible for high-throughput processing of historical data.
2. The **Speed Layer** provides real-time insights by processing live data streams.
3. The **Serving Layer** integrates results from both layers to offer comprehensive analytics.

Batch Layer

1. We implemented this layer using Apache Spark running in its Docker container.
2. The batch layer processes historical data stored in our dataset. It performs periodic computations and batch views of the data are created, which are then stored in Elasticsearch for querying.
3. We then wrote the results from spark to Elasticsearch, allowing for efficient searching and retrieval.

Speed Layer

1. We implemented this layer utilizing Kafka for real-time data ingestion and Spark Streaming for processing the incoming data streams.
2. The speed layer processes data as it arrives in real-time from Kafka topics. It performs immediate computations and transformations on this streaming data to generate real-time insights.
3. Processed streaming data is also sent to Elasticsearch for storage and quick access.

Serving Layer

1. This layer is primarily represented by Elasticsearch.
2. The serving layer combines the pre-computed views from both the batch and speed layers. It allows users to query the most recent data efficiently while providing access to historical data.

3. Kibana connects to Elasticsearch, providing a visualization interface that allows users to explore and analyze both real-time and historical data.

Data Flow:

1. The batch layer processes historical datasets periodically, while the speed layer processes incoming streams in real-time.
2. Both layers output their processed results into Elasticsearch, where they are indexed for fast retrieval.

Implementation of the architecture using **Docker for containerization** and **Elasticsearch for data storage and retrieval**.

Docker Containers Created

1. Kafka Container:
    - Name: `gold-price-analysis-kafka-1`
    - Role: Kafka serves as a distributed messaging system that allows us to publish and subscribe to streams of records. In our architecture, it is used for real-time data ingestion, allowing data to be streamed into the system for immediate processing.
2. Zookeeper Container:
    - Name: `gold-price-analysis-zookeeper-1`
    - Role: Zookeeper is used to manage and coordinate Kafka brokers. It maintains configuration information, provides distributed synchronization.
3. Spark Container:
    - Name: `gold-price-analysis-spark-1`
    - Role: Apache Spark is utilized for both batch and stream processing of data. It processes data ingested from Kafka in real-time and also handles batch jobs using historical data.
4. Elasticsearch Container:
    - Name: `gold-price-analysis-elasticsearch-1`
    - Role: Elasticsearch is a search and analytics engine that stores processed data from both the batch and streaming layers. It allows for fast querying and retrieval of data, making it suitable for real-time analytics.
5. Kibana Container:
    - Name: `gold-price-analysis-kibana-1`

- Role: Kibana provides a user interface for visualizing data stored in Elasticsearch. It allows users to create dashboards and perform searches on the indexed data.

Role of Elasticsearch:

1. Data Storage: Elasticsearch acts as the primary storage engine in your Lambda architecture. It stores both real-time data from Kafka (processed by Spark Streaming) and batch processed data (processed by Spark in batch mode).
2. Search and Analytics: Elasticsearch enables powerful search capabilities on the stored data, allowing users to quickly retrieve insights from large datasets. This is particularly useful for applications requiring real-time analytics.
3. Scalability: As a distributed system, Elasticsearch can scale horizontally by adding more nodes to handle increased data loads, ensuring that performance remains optimal even as the volume of data grows.
4. Integration with Kibana: The integration with Kibana allows users to visualize the data stored in Elasticsearch easily, creating dashboards that provide insights into trends, patterns, and anomalies in the dataset.

Views and Materialized Views:

A view is a virtual table created by a query that pulls data from one or more tables. It doesn't store data physically but generates results dynamically whenever accessed. Views are useful for simplifying complex queries, improving security by restricting user access to specific data, and presenting data in a customized format without altering the underlying tables.

Because views don't store data, they can be slower to query than regular tables, especially for complex views, as the database needs to execute the view's query each time it's accessed.

On the other hand, a materialized view is a physical copy of the result of a query. Unlike regular views, it stores the query results directly in the database. Materialized views are often used to improve performance by pre-computing and storing complex query results, especially in data warehousing and analytics where real-time updates aren't always critical.

Materialized views don't automatically reflect changes in the source tables. They need to be **refreshed** periodically (e.g., scheduled refresh or on-demand) to update the data. Since data is precomputed and stored, querying a materialized view is faster than querying a regular view, as it avoids re-executing the underlying query each time.

Why Use Views in Financial Data Management?

Simplification and Abstraction: With financial data being pulled from multiple tickers with varying structures, views can create a consistent, standardized format for querying. For example, wecan define a view that selects only relevant columns (e.g., Date, Close, Volume) across all tables, so users don't need to know each table's specific structure.

Aggregations and Analytics: Financial analysis often requires computing metrics like daily averages, moving averages, or returns. A view can encapsulate such calculations, making them easily accessible for analysis without recalculating each time.

Logical Separation: Views can separate the physical data model (actual tables) from the logical data model. This separation allows for flexibility, so the underlying tables can change without affecting how end users interact with the data.

Keeping these points in mind, the following views have been built:
1. **Daily Average Price View**: This view could show the average price per asset per day by calculating `(Open + Close) / 2`.

2. **Monthly Summary View**: A view that calculates monthly aggregates like average closing price, highest and lowest values, and total volume.

3. **Volatility View**: Financial analysts often assess volatility, which can be calculated by measuring the daily percentage change. A view can compute this directly, saving users from writing complex formulas.

Why Use Materialized Views in Financial Data Management?

Improved Query Performance: Financial data systems often require high-performance queries, especially during periods of analysis or reporting. For instance, calculating daily or monthly metrics on large datasets is computationally expensive. Materialized views allow the user to pre-compute and store this data, drastically reducing query times.

Reduction in Redundant Calculations: Frequently accessed aggregated data (e.g., monthly or yearly summaries) can be stored once and retrieved without recalculating, saving processing resources and improving user experience.

Incremental Refresh Capabilities: Materialized views can be refreshed incrementally, updating only the rows that have changed. This keeps the materialized view current without the overhead of full recalculation.

Data Archiving and Versioning: Materialized views can also serve as a snapshot or archived version of your data at a specific point in time, aiding in data versioning for historical analysis.

The following materialized views have been created:

1. **Yearly Summary Materialized View**: Create a materialized view for yearly summaries that aggregates data by year (e.g., average open, close, high, and low values).

2. **Moving Averages Materialized View**: For financial analysis, moving averages are commonly used. A materialized view can store the 7-day and 30-day moving averages of the closing prices for each ticker.

3. **Volume Trend Materialized View**: This view stores cumulative or trend-based volume metrics, providing fast access to historical volume trends.

Since the data refreshes daily, a job or trigger is scheduled to refresh the materialized view at regular intervals.

Model Completeness?

The use of views and materialized views enhances the completeness of the data model by addressing essential requirements:

Analytical Completeness: Views and materialized views offer pre-calculated, ready-to-use data for standard financial analyses, such as average prices, volatility, and moving averages. This ensures that the data model supports a wide range of financial metrics without requiring additional transformations.

Performance Optimization: Materialized views store complex aggregations, allowing users to retrieve data instantly, which is crucial for financial applications with real-time or near-real-time requirements.

Scalability: As new financial assets are added to the model, views provide a flexible way to incorporate these assets without rewriting underlying queries. The data model can scale with new data sources and analytical requirements.

Data Versioning and Historical Analysis: Materialized views enable data snapshots at specific points, supporting historical analysis and back-testing. These views also help with compliance and auditing by preserving historical data points.

Logical Data Organization: By organizing complex calculations and aggregations into views, the data model is kept clean and logically organized. Analysts can access insights directly from views rather than combing through raw data.

Data Versioning System

In financial data management, data is often updated, corrected, or enriched with additional details over time. For example:

- Stock price corrections (e.g., after dividends or stock splits).
- Updates to historical data after discovering data issues or inconsistencies.
- Changes in calculation methods for certain metrics.
- Changes in business rules, like reclassifying tickers or industries.

With data versioning, each state of the data is saved, allowing analysts to track changes, perform historical analyses on prior versions, and reproduce analyses accurately, even as the data evolves. This capability supports data integrity, transparency, and compliance.

Temporal Tables in **SQL Server** are used for the purpose of data versioning. Temporal tables are the most straightforward to integrate with existing SQL-based code since they extend standard relational databases. Temporal tables offer automatic versioning without major changes to queries, as historical queries can be done by simply specifying a date.

Database

In this setup, the database used is **MySQL**, one of the most popular relational database management systems (RDBMS). MySQL is known for its ease of use, scalability, and reliable performance, making it a popular choice for storing structured data, especially when high-performance queries and transactional support are required.

Justification for Using MySQL in This Setup

In this case, MySQL is an appropriate choice because:

- **Structured Financial Data**: Financial data, such as historical prices and volumes, fits well into the structured, tabular format of relational databases.
- **Transactional Support**: Ensuring data integrity during updates is crucial in financial applications. MySQL's ACID compliance and transaction management capabilities support these needs effectively.
- **Integration with Python**: Since the code uses Python libraries like `yfinance` to fetch data and `mysql.connector` for database interactions, MySQL is a natural choice due to its compatibility and robust support within Python ecosystems.
- **Simplicity and Reliability**: MySQL provides a straightforward setup and is a well-tested solution for financial and business applications, making it a reliable choice for both small- and large-scale financial data storage.