Supply Chain Management for Big Data

Márton László Hévizi[1], Yishak Tadele Nigatu[2], Ricardo Esquivel [3], Mariia Starodubtseva[4]

[1]martonlaszlo.hevizi@studenti.unitn.it

[2]yishaktadele.nigatu@studenti.unitn.it

[3]r.esquiveldavanzo@studenti.unitn.it

[4]mariia.starodubtseva@studenti.unitn.it

The code is available at the following repository: https://github.com/habarcs/bigdata

Abstract—This paper proposes a system for the efficient processing of high-volume supply chain data, with a primary focus on streaming order data, by utilizing an appropriate integration of advanced big data technologies. The system is capable of delivering real-time key performance indicators, accurate demand forecasts, and comprehensive analytical insights, thereby enabling informed decision-making and operational optimization.

***Keywords*—Supply Chain, Demand Prediction, Inventory Management, Spark, Kafka, Flink, Streamlit**

I. Introduction

Global commerce has been significantly transformed, driven by increasingly complex and data-intensive operations. In the context of supply chain management, every transaction, from procurement to customer delivery, generates vast amounts of data that must be managed efficiently to optimize operations, reduce costs, and improve customer satisfaction [1].

To address these challenges, this supply chain management system leverages several big data technologies such as Apache Kafka for real-time messaging and data streaming, Apache Flink for distributed processing, Apache Spark and Facebook Prophet for machine learning, PostgreSQL for data storage, and Docker Compose for containerized orchestration. Each stage in the pipeline is modular, enabling seamless integration and scalability. The system utilizes real-time data streaming, predictive analytics, and containerized deployment to efficiently manage supply chain data and generate forecasting from historical trends.

II. System Model

A. *System architecture*

The system architecture consists of six main modules designed to ensure scalability, reliability, and efficiency for handling streamed data. The components include Data Generation, Event Streaming, Data Processing, Data Storage, Demand Forecasting, and a User Dashboard, represented in Fig. 1.

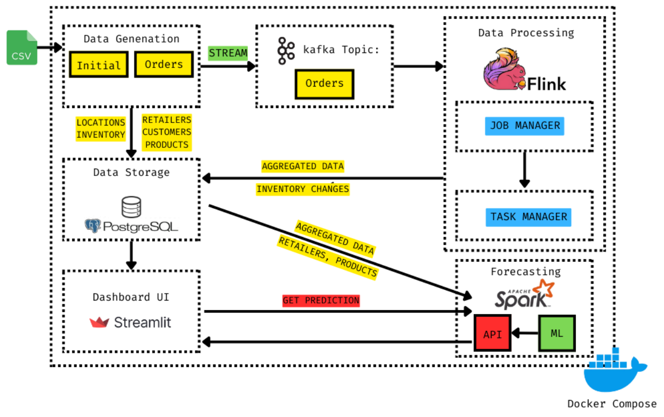


Fig. 1 System Architecture

The **Data Generation** module begins by capturing data from an external source. For this project, the primary dataset is sourced from Mendeley Data [2], which provides a CSV file containing transaction details from DataCo Global, primarily focused on orders made for US companies from all over the world, along with information about customers and retailers. Since publicly available datasets covering a complete supply chain with all entities are rare, this project focuses on the relationships between customers, retailers, and inventory, with the ultimate goal of providing a user interface for efficient inventory management, demand forecasting, as well as tracking sales and delivery performance. To enhance the dataset, we generate synthetic data to introduce additional fields and new entities, ensuring a more comprehensive dataset for these specific tasks. Available data and synthetic data are then combined and organized into several tables in a **PostgreSQL** database, including tables for `products`, `retailers`, `customers`, `locations`, and `inventory`.

In the **Event Streaming** module, the orders, being transactional in nature, are produced as **Kafka** events to enable real-time data processing and ensure scalability. Each order of the original dataset is processed through a well-defined pipeline, where data is transformed, enriched and streamed to Kafka with unique keys derived from timestamps for efficient consumption. The integration of Kafka ensures that this process can scale dynamically and handle large volumes of transactional data in real time.

The **Data Processing** module deploys an **Apache Flink** streaming pipeline, leveraging its distributed architecture for real-time data processing. The JobManager orchestrates the pipeline by managing job execution, resource allocation, and fault tolerance, while the TaskManager executes the actual processing tasks in parallel across its configurable task slots. Flink consumes order data from the Kafka topic **`orders`**, transforms and aggregates it, and stores it in PostgreSQL for the long term. It also updates inventory levels for retailers and allows them to monitor and manage supply for products with low stock. This setup ensures seamless updates to the `inventory` and `daily_aggregates` tables, providing accurate inventory and demand insights to drive efficient supply chain management.

The **Data Storage (PostgreSQL database)** module utilizes PostgreSQL to organize and persistently store both reference and aggregated data from the pipeline. As previously mentioned, the database schema is designed to accommodate different aspects of the supply chain, including tables for `products`, `retailers`, `inventory`, `customers`, `locations`, and aggregated metrics (see Fig. 2). The `inventory` table stores real-time stock levels and reorder thresholds for each product-retailer combination, while the `daily_aggregates` table captures processed aggregated metrics from the Flink pipeline. Additionally, the `data_gen` and `kafka_sent` tables track the status of data generation and the number of Kafka events sent, ensuring data consistency and avoiding duplication. This allows all components to be stateless, easily scalable, and idempotent, allowing for truly fault-tolerant operation.

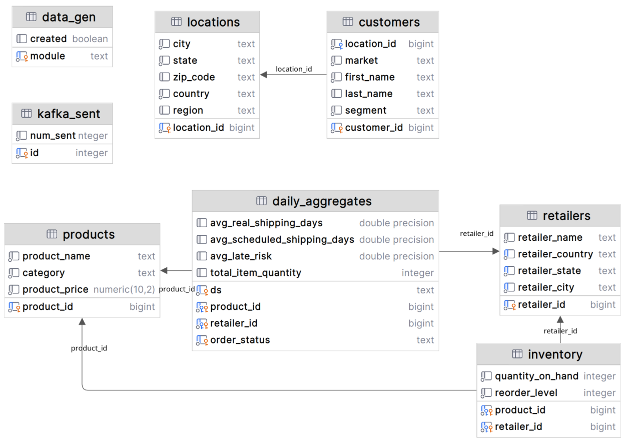


Fig. 2 PostgreSQL database structure

The **Forecasting (Machine Learning)** module leverages **Apache Spark** for distributed data processing, specifically for machine learning tasks. A Spark instance is initialized to manage resources and connect to the PostgreSQL database, where it retrieves relevant historical metrics from the `daily_aggregates` table based on a user-specified date range. The data is then enriched with metadata from `products` and `retailers`. The machine learning process is initiated via a RESTful **API**, which interfaces with the final dashboard, allowing users to specify the time range, products, and retailers for the forecast. The API dynamically constructs queries to fetch only the relevant subsets of data based on these parameters. This approach ensures that only the necessary data is processed, minimizing resource usage and optimizing performance for the forecast generation.

The **User Interface (Dashboard)** module is developed with **Streamlit** and provides sections for Home, Inventory Management, Demand Forecasting, Delivery Performance, Sales Performance, and Real-time Streaming. It connects to the PostgreSQL database to retrieve information about retailers, products, and inventory, as well as daily aggregated data. To facilitate real-time insights, the dashboard subscribes to the Kafka topic **`orders`**, providing immediate visibility into ongoing sales activity. Additionally, it includes functionality to request demand forecasts via a REST API, invoking the forecasting module that generates predictive analytics. This integration of real-time data processing, forecasting, and inventory management allows the dashboard to offer actionable insights into various aspects of the supply chain.

B. ***Technologies***

The technologies chosen for this project were selected to address the critical requirements of real-time data processing, efficient storage, and seamless scalability, as well as provide valuable insights, forecasts on the data, and a user-friendly interface.

Docker [3] is used to simplify the process of building, deploying, and running applications using containers. In this project, Docker is utilized to encapsulate various services and tools, such as Kafka, PostgreSQL, Flink, Spark, and Streamlit, ensuring seamless integration and interaction between them. The `compose.yaml` file defines the configuration of these services, specifying dependencies, health checks, ports, and environmental variables. This setup enables a modular and scalable big data pipeline where each service operates in an isolated environment, minimizing conflicts and easing deployment.

Apache Kafka [4] is a distributed event streaming platform designed to handle real-time communication between the various components of the pipeline. It enables applications to publish, subscribe to, store, and process data streams efficiently and reliably. In this project, Kafka serves as the backbone for real-time data processing, where it acts as a producer-consumer system. Data is produced by a Kafka producer script (`kafka_producer.py`), which streams structured data into Kafka topics, facilitating its ingestion and processing. This architecture allows scalable data pipelines, ensuring seamless data flow between components while maintaining fault tolerance and high throughput for large volumes of data.

Apache Flink (PyFlink) [5] serves as the distributed computing system for large-scale data processing in this project, enabling efficient transformations, aggregations, and preprocessing. It is utilized to consume real-time streamed data produced by Apache Kafka, parsing and transforming it into a structured format. The processed data is enriched by joining it with additional customer, product, and retailer information fetched from a PostgreSQL database, ensuring comprehensive and context-rich insights for further analysis.

PostgreSQL [6] is used in this project as a relational database management system to store and manage structured data. It serves as a repository for key datasets, including customer, product, and retailer information, which are used to enrich the real-time data streams consumed from Apache Kafka. It also serves as a destination for storing the processing results, allowing downstream tasks such as the UI to fetch it.

Apache Spark (PySpark) and Prophet [7, 8] are used in the pipeline for forecasting sales data. Spark connects to a PostgreSQL database via a REST API to retrieve data on sales, products, and retailers. This data is joined within Spark, and Prophet models are trained to forecast future sales quantities, providing scalable and accurate predictive insights.

The pipeline implements **Streamlit** [9] to build an interactive dashboard for monitoring and optimizing supply chain performance. The system retrieves data from a PostgreSQL database and generates key performance indicators (KPIs) and visual analytics for monitoring inventory, sales, and delivery performance. Additionally, it provides demand forecasts derived from predictive models, enabling informed decision-making. Real-time streaming capabilities allow for dynamic visualization of changes in sales, ensuring up-to-date monitoring and insights.

III. Implementation

This Real-Time Data Processing and Forecasting System is designed to ensure seamless ingestion, processing, and storage of data streams. The project is hosted on GitHub, accessible at the designated repository [https://github.com/habarcs/bigdata]. The detailed implementation of the project is described below.

The system's orchestration is managed via Docker Compose, enabling the containerized components of the system to be both scalable and fault-tolerant. It automates health checks for the system and ensures that containers are started in the correct order. The architecture supports easy scaling, allowing additional Kafka brokers or Flink task managers to be added with minimal effort to handle increased resource demands.

The `SupplyChainDataset.csv` file, located in the `data_gen/data` folder, contains the raw structured data utilized in this project. It includes supply chain information, such as customer details, order specifics, and product data.

In the `data_gen` folder, `create_initial_data.py` script is responsible for populating the PostgreSQL database with initial data. It includes a suite of functions that parse the CSV file to generate unique tables for products, retailers, inventory, locations, and customers, ensuring data integrity by preventing duplication. Additionally, the script employs synthetic data generation to augment the dataset. The database schema is specified in the `db_schema.sql` file, located in the `postgres_startup` folder. This schema defines the primary keys, relationships, and constraints for the created tables, providing a structured and consistent foundation for the database.

The transactional part of the dataset, specifically the orders, is streamed using Kafka. In the `data_gen` folder, the Kafka cluster is configured with a single instance, functioning as both the controller and message broker. To accommodate higher demand or redundancy requirements, additional brokers can be added to ensure scalability. Kafka advertises listeners on `kafka:9092`, available within the Docker network, and `localhost:9092`, accessible from the host machine running the Docker containers. All communication occurs in plaintext, without encryption.

The system uses a single topic for the orders (named **`orders`**), with the partition size set to 3 by default and no replication. In the `data_gen` module, the `kafka_producer.py` script generates events every second by extracting a row from the CSV file and transforming it into a format suitable for an order event. The event key is generated based on the order date, ensuring that data generated in close temporal proximity is stored within the same partition. The number of messages sent is tracked and stored in the PostgreSQL database, allowing the system to resume processing from the last point in the event of a failover.

Flink is responsible for processing the orders streamed from the Kafka topic `orders`, reading and aggregating the data in real-time for future use, and updating the inventory for the retailers. The system consists of two machines: a JobManager, which schedules tasks, and a TaskManager, which executes them. In the event of a bottleneck, Docker Compose facilitates the scaling of TaskManagers to improve processing throughput. Flink is configured in standalone mode as an application cluster, executing the processing defined in the `processing/__main__.py` file.

Flink communicates with the services using the Postgres and Kafka connectors. Through the Table API, it updates the inventory table based on the sold item quantity for each retailer and product. Additionally, it aggregates data in a daily resolution, calculating the total quantity sold, average shipping times and other metrics displayed later on the dashboard, storing this information in a separate aggregation table.

Spark is utilized to run machine learning models for demand forecasting. To optimize resource usage, it does not run continuously but is instead triggered on-demand via an HTTP request. A Flask server running on the Spark machine accepts POST requests at the URL `spark:9003/start-forecast`, which include a list of product and retailer IDs, a start and end date, and the forecast duration. Upon receiving a request, the server first validates the input and then initiates a Spark session to execute the Facebook Prophet prediction algorithm.

The API for interacting with Spark is specified in the `spark_api.py` file, while the machine learning logic is implemented in `spark.py`, both located in the `ml` folder. The data required for the prediction is sourced from the PostgreSQL database, where aggregated order information is stored. This data is further enriched with relevant customer and retailer details to improve the forecast's accuracy. Spark DataFrames are used for efficient data manipulation throughout the process. Given the speed of the algorithm, the system is able to respond synchronously to the POST request.

Finally, Streamlit is used for the dashboard and data visualization, with the code located in the `dashboard` folder. It connects to the database to load all relevant data into the session, from which it generates graphs and plots using Plotly. For real-time information, Streamlit subscribes to the Kafka `orders` stream to display up-to-date insights on current usage. Additionally, it triggers the demand prediction feature by connecting to the Spark machine's API, fetching predictions, and plotting the results on the dashboard.

Execution

To execute the Big Data Supply Chain Optimization system, it should first be cloned from the GitHub repository in an appropriate project directory. Docker and Docker Compose must be installed on the system. These tools manage the deployment and interaction of the project's services, as defined in the `docker-compose.yml` file.

To build the necessary Docker images and start all the defined services, execute the following command inside the project directory:

```
>> docker compose up -d --build
```

To connect to the PostgreSQL database container and interact with it:

```
>> docker exec -it
supply_chain_big_data-sql-database-1 psql -U
postgres
```

To connect to the Kafka container and work with message queues:

```
>> docker exec -it
supply_chain_big_data-kafka-1 bash
```

Inside the container, Kafka tools are located in the `/opt/kafka/bin/` directory. For instance, to list messages from the `orders` topic:

```
>> /opt/kafka/bin/kafka-console-consumer.sh
--bootstrap-server localhost:9092 --topic
orders
```

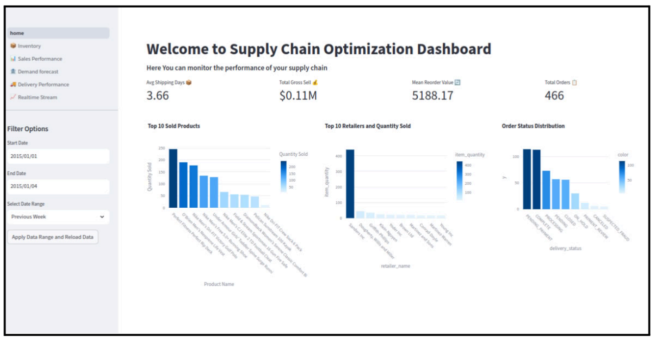
Once all the containers are running, access the dashboard at: `http://localhost:8501/`

IV. Results

The system effectively integrates supply chain data to create a real-time monitoring and analytics platform. It utilizes a scalable big data pipeline designed to handle high-volume and high-velocity data streams. By leveraging distributed processing frameworks such as Apache Spark, Flink and Kafka, the system ensures robust data ingestion, processing, and analysis. Additionally, a machine learning model is incorporated to provide predictive insights, enhancing decision-making capabilities.

The final output is a user interface designed to assist users in displaying analytics. Through a sidebar, users can select various types of analytics, such as inventory management, sales performance, and demand forecasting, as illustrated in Figures 3, 4, and 5.

To ensure that the graphs are meaningful, users should wait for a sufficient number of orders to accumulate before displaying the data. As the orders are not prepopulated, this approach guarantees that the visualizations reflect real-time trends and provide actionable insights. This feature simulates a real-life data flow, capturing how supply chain data fluctuates throughout the day.

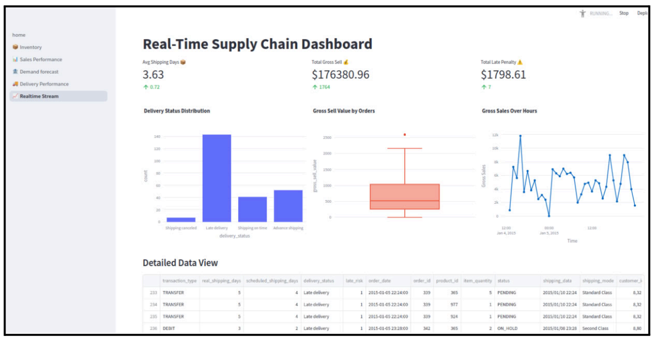


Welcome to Supply Chain Optimization Dashboard

Here You can monitor the performance of your supply chain

Avg Shipping Days	Total Gross Sell	Mean Reorder Value	Total Orders
3.66	$0.11M	5188.17	466

Fig. 3 Home Page



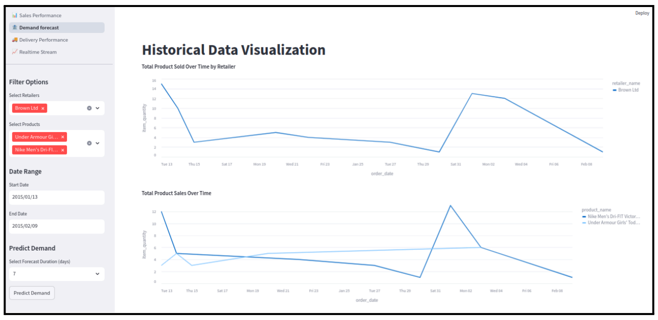
Real-Time Supply Chain Dashboard

Avg Shipping Days	Total Gross Sell	Total Late Penalty
3.63 ↑ 0.72	$176380.96 ↑ 1764	$1798.61 ↑ 7

Detailed Data View

	transaction_type	real_shipping_days	scheduled_shipping_days	delivery_status	late_risk	order_date	order_id	product_id	item_quantity	status	shipping_data	shipping_mode	customer_i
233	TRANSFER	5	4	Late delivery	1	2015-01-05 22:24:00	339	365	5	PENDING	2015/01/10 22:24	Standard Class	8,32
234	TRANSFER	5	4	Late delivery	1	2015-01-05 22:24:00	339	977	1	PENDING	2015/01/10 22:24	Standard Class	8,32
235	TRANSFER	5	4	Late delivery	1	2015-01-05 22:24:00	339	924	1	PENDING	2015/01/10 22:24	Standard Class	8,32
236	DEBIT	3	2	Late delivery	1	2015-01-05 23:28:00	342	365	2	ON_HOLD	2015/01/08 23:28	Second Class	8,80

Fig. 4 Real-Time Stream Landing Page



Historical Data Visualization

Fig. 5 Demand Forecasting landing page

V. Conclusions

During this project, we developed a functional supply chain optimization system. While it successfully operates with practice data, additional features and security measures would be required for production deployment. Our architecture offers potential for further exploration, as it has yet to be tested for its full capacity and performance bottlenecks. Furthermore, by incorporating data that represents a complete supply chain, the system could be extended to manage other components, such as raw materials, manufacturers, and distributors. Nonetheless, the current outcome is fully operational and scalable, delivering satisfactory results.

References

[1] McKinsey & Company, *What is supply chain?*. Available: https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-supply-chain#/. [Accessed: Jan. 22, 2025].

[2] Constante, Fabian; Silva, Fernando; Pereira, António (2019), “DataCo SMART SUPPLY CHAIN FOR BIG DATA ANALYSIS”, Mendeley Data, V5, doi: 10.17632/8gx2fvg2k6.5

[3] Docker, "Docker Documentation," *Docker Manuals*, 2024. [Online]. Available: https://docs.docker.com/. [Accessed: Jan. 24, 2025].

[4] Apache Spark, "Documentation," *Apache Spark*. [Online]. Available: https://spark.apache.org/documentation.html. [Accessed: Jan. 24, 2025].

[5] Apache Flink, "Apache Flink Documentation," *Apache Flink Documentation*. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-stable/. [Accessed: Jan. 24, 2025].

[6] PostgreSQL Global Development Group, "PostgreSQL 17.0 documentation," *PostgreSQL Documentation*, 2024. [Online]. Available: https://www.postgresql.org/docs/current/. [Accessed: Jan. 24, 2025].

[7] Apache Spark, "PySpark Overview," *PySpark 3.5.4 documentation*. [Online]. Available: https://spark.apache.org/docs/latest/api/python/index.html. [Accessed: Jan. 24, 2025].

[8] Prophet Team, "Prophet Quick Start," *Prophet*, 2024. [Online]. Available: https://facebook.github.io/prophet/docs/quick_start.html. [Accessed: Jan. 24, 2025].

[9] Streamlit, "Streamlit docs," *Streamlit documentation*. [Online]. Available: https://docs.streamlit.io/. [Accessed: Jan. 24, 2025].