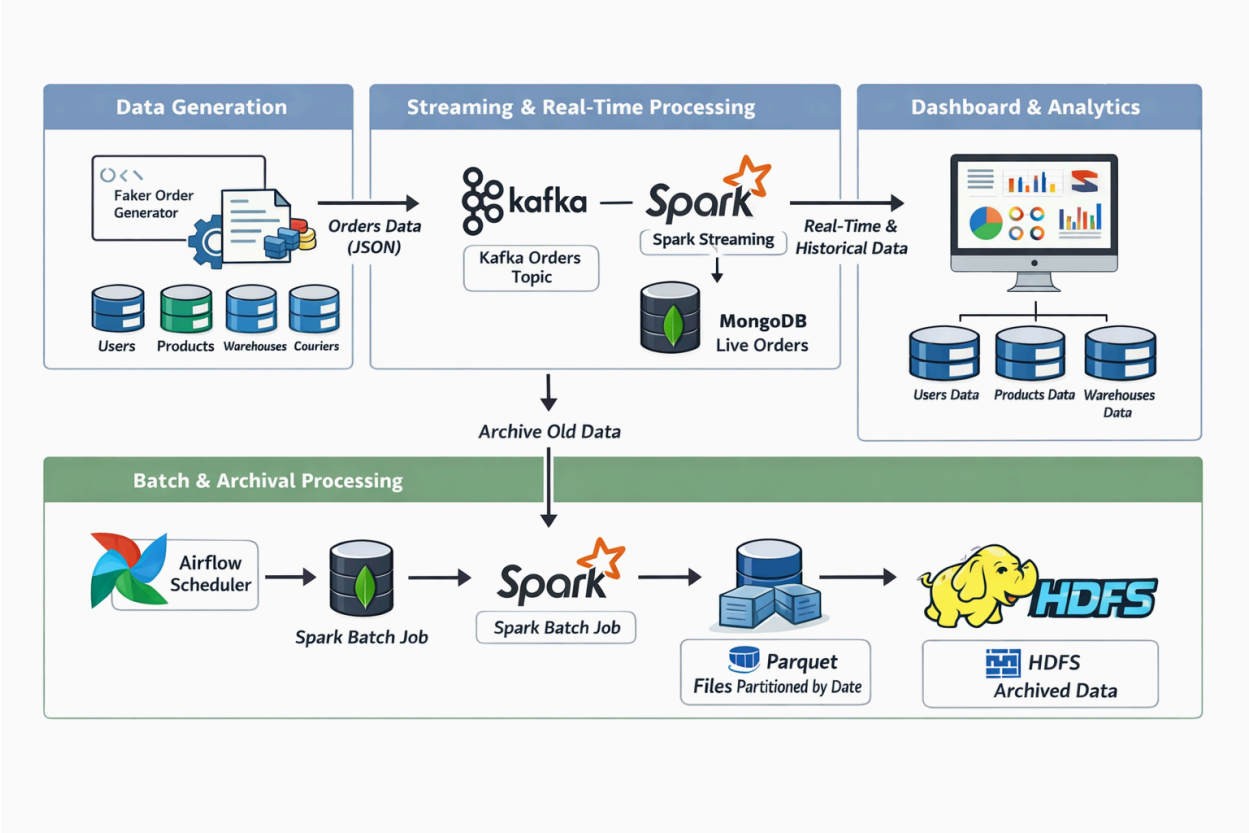
BDA Term Project

Real Time BDA Pipeline

M. Ibrahim Ayoubi 26269 | Kisa Fatima 27076

TABLE OF CONTENTS

BDA Pipline



The **data flow** begins with synthetic order generation using Faker, which simulates customers, products, warehouses, couriers, and inventory. Orders are sent in real time to **Kafka**, which acts as a streaming layer. **Spark Streaming** consumes this data, performs processing and joins with dimension tables, and stores the structured data in **MongoDB** for live analytics.

To manage storage and enable historical analytics, a **batch/archival layer** periodically flushes old MongoDB data to **HDFS** using Spark jobs, while Airflow orchestrates the scheduling of both streaming and batch jobs. Finally, a **Streamlit dashboard** reads the live data and dimension tables from MongoDB to display KPIs and interactive visualizations, enabling managers to monitor orders, revenue, deliveries, and inventory in near real time.

The accompanying **architecture diagram** illustrates the full pipeline, showing the flow from data generation, through streaming and batch processing, to live dashboard visualization.

Data Details

Selected Domain

The selected domain for this project is **E-commerce Real-Time Order and Inventory Analytics**.
Modern e-commerce platforms generate continuous streams of orders and inventory updates that must be processed instantly. Delayed processing can lead to stock inconsistencies, order failures, and poor customer experience. Therefore, real-time data analytics is required to monitor orders, update inventory dynamically, and ensure accurate and timely order fulfillment.

The system simulates a real-world online shopping platform where:

- Customers place orders in real time
- Orders consist of multiple products
- Inventory levels change dynamically
- Deliveries are tracked continuously
- Management requires live insights through dashboards

Data Generation Approach

In this project, real-time data is generated using a **Python-based streaming generator** built with the **Faker library**, NumPy, and randomization techniques.
The goal is to simulate a realistic **e-commerce environment** where customer orders are continuously created and streamed into the system.

The data generation logic is implemented in:

`data_generator/generate_realtime_data.py`

The generated data is structured and consistent, making it suitable for analytical processing and BI dashboards.

Schema Design Overview

The schema is designed using a **fact–dimension model**, which is commonly used in data warehousing and BI systems.

The database consists of:

- **Dimension tables** → Static reference data
- **Fact tables** → Continuously generated transactional data

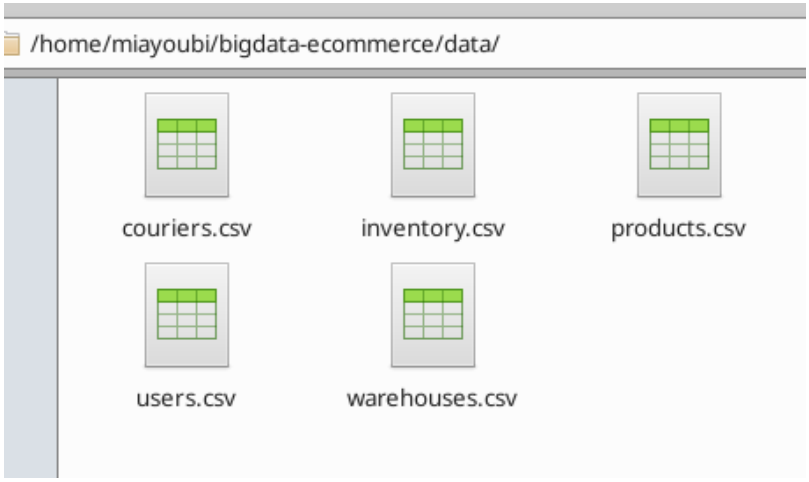
This design supports efficient joins, aggregations, and analytical queries required for dashboard visualization.

Dimension Tables

Dimension tables contain descriptive information and are generated once at the start of the system.
These tables remain mostly static and are used for joining with streaming data.

The following dimension tables are used:

- **Users** – Stores customer-related information
- **Products** – Contains product catalog details
- **Warehouses** – Holds warehouse and storage details
- **Couriers** – Stores delivery personnel information
- **Inventory** – Maintains stock availability per warehouse



Fact Tables (Streaming Data)

Fact tables contain real-time transactional data and are generated continuously.

The following fact tables are used:

- **Orders** – Represents customer purchases
- **Order Items** – Contains individual products within each order
- **Deliveries** – Tracks order delivery status and time

Each new order dynamically links with:

- A user
- One or more products
- A warehouse
- A courier

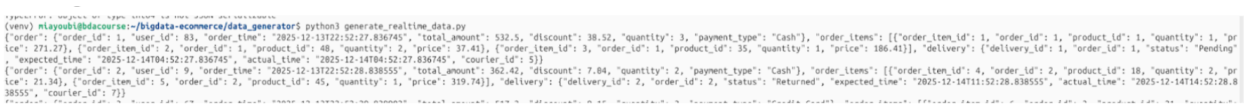
KPIs Supported by the Schema

The schema supports the following business KPIs:

- Total Revenue
- Average Order Value
- Total Quantity Sold
- Inventory Stock Levels
- Delivery Delay
- Orders per Location
- Sales per Product Category

These KPIs are later visualized on the real-time dashboard.

Generated Data:



Real-Time Data Generation Logic

The data generation process follows a structured workflow implemented in Python:

1. **Initialization Phase**
    - Faker library is initialized with a fixed seed for consistency.
    - Dimension tables (users, products, warehouses, couriers) are generated once.
    - These tables remain static throughout execution.

2. **Order Generation**
    - A random user is selected from the users table.
    - A random number of products (1–5) is chosen per order.
    - Product price and quantity are used to compute total amount.
    - Discount is applied randomly.
    - Payment method is selected randomly.

3. **Order Item Creation**
    - Each order contains multiple order items.
    - Each item references an existing product.
    - Quantity and price are calculated per item.

4. **Delivery Generation**
    - A courier is assigned randomly.
    - Expected and actual delivery times are generated.
    - Delivery status is assigned (Pending, Shipped, Delivered).

5. **JSON Structuring**
    - All information is combined into a single JSON object:
        - Order
        - Order items
        - Delivery details

6. **Continuous Streaming**
    - The process runs in an infinite loop.
    - A new order is generated every second.
    - This simulates real-time data flow.

Kafka-Based Streaming

Kafka Producer Integration

Apache Kafka is used as the **real-time data ingestion layer** in the system.

A Kafka producer is initialized in the data generator script using:

```
KafkaProducer(
    bootstrap_servers="127.0.0.1:9092",
    value_serializer=lambda v: json.dumps(v).encode("utf-8"))
```

Kafka Streaming Process

1. Each generated order is converted into a JSON message.
2. The message includes:
    - Order details
    - Order items
    - Delivery information
3. The message is published to the Kafka topic: `orders`
4. Kafka acts as a buffer between:
    - Data generation layer
    - Processing and analytics layer
5. The producer continuously pushes data every second, simulating live traffic.

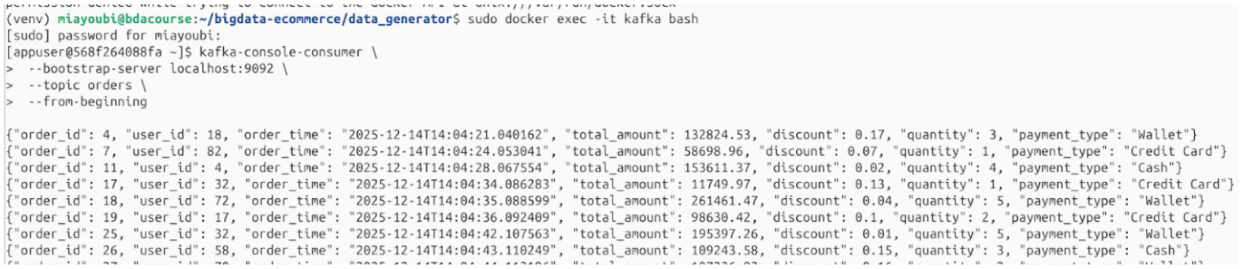
Why Kafka is Used

- Handles real-time streaming efficiently
- Decouples data generation from processing
- Supports scalability
- Ensures reliable message delivery

Generated Data to Kafka:

```
(venv) miayoubi@bdacourse:~/bigdata-ecommerce/data_generator$ python generate_realtime_data.py
🚀 Faker-based Kafka streaming started...
/home/miayoubi/bigdata-ecommerce/data_generator/generate_realtime_data.py:86: DeprecationWarning
atetime.now(datetime.UTC).
  order_time = datetime.utcnow()
✅ Order 1 pushed to Kafka
✅ Order 2 pushed to Kafka
✅ Order 3 pushed to Kafka
✅ Order 4 pushed to Kafka
✅ Order 5 pushed to Kafka
✅ Order 6 pushed to Kafka
✅ Order 7 pushed to Kafka
✅ Order 8 pushed to Kafka
```

Orders received by Kafka:



```
(venv) miayoubi@bdacourse:~/bigdata-ecommerce/data_generator$ sudo docker exec -it kafka bash
[sudo] password for miayoubi:
[appuser@568f264088fa ~]$ kafka-console-consumer \
>  --bootstrap-server localhost:9092 \
>  --topic orders \
>  --from-beginning

{"order_id": 4, "user_id": 18, "order_time": "2025-12-14T14:04:21.040162", "total_amount": 132824.53, "discount": 0.17, "quantity": 3, "payment_type": "Wallet"}
{"order_id": 7, "user_id": 82, "order_time": "2025-12-14T14:04:24.053041", "total_amount": 58698.96, "discount": 0.07, "quantity": 1, "payment_type": "Credit Card"}
{"order_id": 11, "user_id": 4, "order_time": "2025-12-14T14:04:28.067554", "total_amount": 153611.37, "discount": 0.02, "quantity": 4, "payment_type": "Cash"}
{"order_id": 17, "user_id": 32, "order_time": "2025-12-14T14:04:34.086283", "total_amount": 11749.97, "discount": 0.13, "quantity": 1, "payment_type": "Credit Card"}
{"order_id": 18, "user_id": 72, "order_time": "2025-12-14T14:04:35.088599", "total_amount": 261461.47, "discount": 0.04, "quantity": 5, "payment_type": "Wallet"}
{"order_id": 19, "user_id": 17, "order_time": "2025-12-14T14:04:36.092409", "total_amount": 98630.42, "discount": 0.1, "quantity": 2, "payment_type": "Credit Card"}
{"order_id": 25, "user_id": 32, "order_time": "2025-12-14T14:04:42.107563", "total_amount": 195397.26, "discount": 0.01, "quantity": 5, "payment_type": "Wallet"}
{"order_id": 26, "user_id": 58, "order_time": "2025-12-14T14:04:43.110249", "total_amount": 109243.58, "discount": 0.15, "quantity": 3, "payment_type": "Cash"}
```

Real-Time Processing

Kafka → Spark → MongoDB

After data is generated, it is streamed in real time using **Apache Kafka**.
Kafka acts as the messaging layer that receives continuous order data from the data generator.

Kafka Layer

- Each order is sent to a Kafka topic named `orders`
- Data is sent in JSON format
- Kafka ensures reliable and continuous streaming

Spark Streaming Layer

Apache Spark is used to process the Kafka stream in real time.

Spark performs the following tasks:

- Reads live data from the Kafka topic
- Converts JSON messages into structured format
- Extracts order, order items, and delivery data
- Processes data in micro-batches

Spark acts as the processing engine between Kafka and MongoDB.

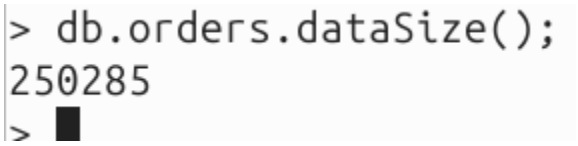
MongoDB Storage Layer

After processing, Spark writes the data to **MongoDB**.

- MongoDB stores real-time transactional data
- Data is saved in structured collections
- Used as the main database for analytics and dashboards
- Supports fast read operations for live visualization

Checkpointing is used to ensure fault tolerance and prevent data loss.

Data in MongoDB:



Archiving Data

MongoDB → HDFS

To manage the continuously growing real-time data in MongoDB and prevent it from exceeding storage limits, an **archiving process** was implemented using Spark.

The logic works as follows:

- Spark reads the **orders collection** from MongoDB and converts the order timestamps into a date column.
- Records older than a configured threshold (e.g., one day) are selected for archiving.
- These records are written to **HDFS in Parquet format**, partitioned by order date for efficient storage and query performance.
- After successful archiving, the corresponding records in MongoDB are deleted using Spark's MongoDB connector.
- The archived data in HDFS serves as a historical data store for analytical queries, dashboards, and further processing.

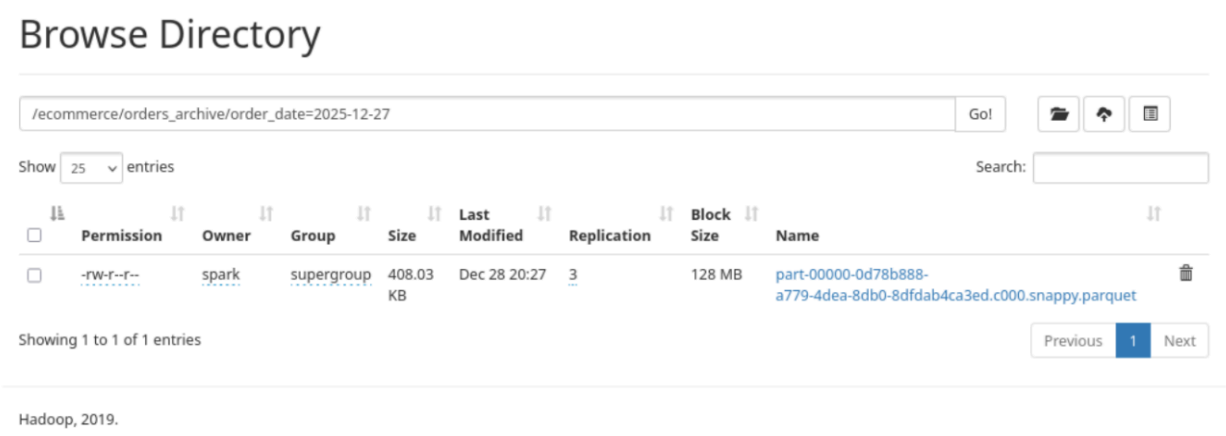
Key points:

- Archiving is triggered when MongoDB data exceeds a certain size (e.g., 300 MB).
- Partitioning in HDFS improves query efficiency and future Spark processing.
- This approach ensures MongoDB contains only the most recent real-time data, while historical data is safely stored in HDFS.
- Future enhancement includes adding a **flag per row** in MongoDB to track whether a record has been archived before deletion.

This strategy allows the system to maintain **real-time performance** in MongoDB while keeping a **durable historical record** in HDFS for reporting and analytics.

Data Archived:

```
25/12/28 21:22:40 INFO StandaloneAppClient$ClientEndpoint: Executor updat
Archiving 11768 records to HDFS... (DataFrame size: 7.91 MB)
Archived to HDFS at: hdfs://namenode:9000/ecommerce/orders_archive/
Marked 11768 records as archived in MongoDB.
Appended archive summary to MongoDB collection 'orders_archive_summary'.
Overwritten MongoDB collection 'orders' with updated archived flags.
miayoubi@bdacourse:~/bigdata-ecommerce$
```



Workflow Orchestration with Airflow

To automate the real-time pipeline and the archiving process, **Apache Airflow** is used as the orchestration layer. Airflow manages scheduling, dependencies, and monitoring for both Spark jobs.

Workflow Overview

1. **Spark Streaming Job (`spark_stream_orders.py`)**
    - Scheduled to run continuously via Airflow.
    - Reads real-time orders from **Kafka**, processes them using Spark, and writes structured data to **MongoDB**.
    - Handles live streaming of orders, order items, and delivery data.

2. **Spark Archiving Job (`mongo_to_hdfs.py`)**
    - Scheduled as a **daily or periodic batch job** in Airflow.
    - Reads historical data from MongoDB exceeding the size threshold or older than a set date.
    - Archives data to **HDFS in Parquet format** (partitioned by order date) and deletes the corresponding records from MongoDB.

Real-Time Dashboard

The final layer of the pipeline visualizes the processed data from MongoDB using a **real-time dashboard** built with **Streamlit**.

Dashboard Overview

- The dashboard connects to MongoDB and reads both **dimension tables** (users, products, warehouses, couriers, inventory) and **fact tables** (orders, order items, deliveries).
- Data from MongoDB is preprocessed using **Pandas**, including:
    - Flattening nested JSON data
    - Converting timestamps to datetime
    - Merging order items with product and warehouse information
- The dashboard automatically **refreshes every few seconds** to show the latest data.

Key Features & KPIs

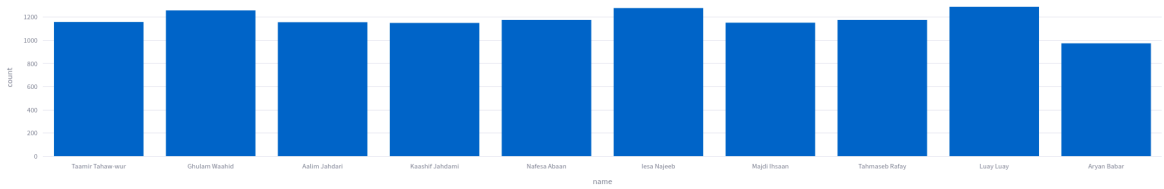
- **Total Orders**, **Total Revenue**, **Average Order Value**
- **Delivered vs Pending Orders**
- **Payment Type Distribution** (Pie chart)
- **Top 10 Products Sold** (Bar chart)
- **Delivery Status Over Time** (Line chart)
- **Orders per Courier** and **Orders per Warehouse** (Bar charts)

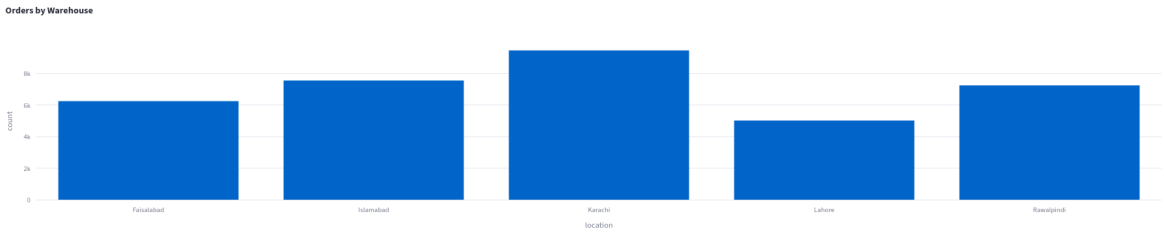
Data Processing Logic

- Orders are exploded to separate individual order items for detailed analytics.
- Merges with dimension tables enable queries like:
    - Total sales per product
    - Orders handled by each courier
    - Warehouse-level order analysis
- Aggregations are performed in Pandas and visualized using **Plotly** for interactive charts.

Dashboard Benefits

- Provides **live insights** into real-time e-commerce operations.
- Enables managers to monitor **sales, deliveries, and inventory** continuously.
- Integrates seamlessly with the upstream **MongoDB Spark pipeline**, ensuring the dashboard reflects current streaming data.





Work Division

Ibrahim:

- Implemented Kafka-Spark connection for real-time streaming
- Developed Spark jobs for processing orders and archiving data
- Managed MongoDB flushing and HDFS archival
- Configured Airflow DAGs for workflow scheduling and orchestration

Kisa:

- Developed data generation scripts using Faker
- Handled Kafka data ingestion and MongoDB setup
- Built the real-time dashboard with aggregations and joins
- Created Docker Compose setup for containerized deployment
- Prepared the project report and documentation