Technical Report: Real-Time Economic Analytics Platform

Architecture
Backend Engineering Team

December 25, 2025

Contents

1 Project Overview

The objective of this project is to develop a digital platform composed of two main interfaces: **Accountant (Comptable)** and **Administrator**. The system is designed to handle high-volume transactional data, process it in real-time using Big Data technologies, and provide advanced economic insights (Product Winners/Lossers, Client Retention) to decision-makers.

2 System Architecture

The system follows a **Lambda Architecture** approach, handling both batch processing (uploading files) and stream processing (real-time analytics).

2.1 Workflow Description

1. **Data Ingestion (Interface Comptable):** The accountant uploads data (Stock, Sales, Clients) via CSV/Excel files.
2. **Parsing & Queuing:** The backend parses the files and pushes raw data events into **Apache Kafka** topics.
3. **Stream Processing (Apache Spark):** Spark Structured Streaming consumes Kafka topics to perform economic analysis (Profit calculation, Dead stock detection).
4. **Storage Layer:**
   - **Hot Storage (MongoDB):** Stores processed KPIs for the Admin Dashboard.
   - **Cold Storage (HDFS):** Stores raw and processed data for historical auditing.
5. **Visualization (Interface Admin):** The Admin dashboard subscribes to real-time updates to display ”Winner Products”, ”Loss Risks”, and ”Eligible Reductions”.

3 Data Modeling & Conception

The data structure is designed to support analytical queries.

3.1 Table Definitions

3.1.1 1. Products Table (Master Data)

Stores static and pricing information about items.

- `product_id` (UUID): Unique Identifier.
- `name` (String): Product Name.
- `category` (String): Classification.
- `buy_price` (Decimal): Cost of goods sold.
- `sell_price` (Decimal): Retail price.
- `min_margin_threshold` (Float): Minimum acceptable profit margin.

3.1.2 2. Stock Table (Inventory)

Used to track inventory health and expiry.

- `stock_id` (UUID): Unique Identifier.
- `product_id` (FK): Reference to Product.
- `quantity` (Integer): Current count.
- `batch_no` (String): Traceability batch number.
- `expiry_date` (Date): Critical for detecting ”Loss” products.
- `last_movement` (Timestamp): Date of last activity.

3.1.3 3. Sales/Facteur Table (Transactions)

The primary source for ”Winner” analysis.

- `invoice_id` (UUID): Unique Identifier.
- `client_id` (FK): Reference to Client.
- `items` (Array¡Object¿): List of products sold in this invoice.
- `total_amount` (Decimal): Final invoice value.
- `timestamp` (DateTime): Exact time of purchase.

3.1.4 4. Clients Table

Used for loyalty and reduction algorithms.

- `client_id` (UUID): Unique Identifier.
- `name` (String): Client Name.
- `total_revenue_generated` (Decimal): Lifetime Value (LTV).
- `loyalty_tier` (String): Standard, Silver, Gold.

4 Backend Logic & Functions (Spark & Kafka)

This section details the core processing logic implemented in the backend.

4.1 Apache Kafka Configuration

The system utilizes specific topics to segregate data flow:

- `topic-raw-sales`: Incoming sales data from CSV uploads.
- `topic-inventory-updates`: Stock level changes.
- `topic-analytics-results`: Output of Spark processing (consumed by Admin Dashboard).

4.2 Spark Streaming Functions

The following logic is implemented using PySpark/Scala Spark.

4.2.1 Function 1: calculate_product_winner()

Goal: Identify top-performing products over a specific time window (Weekly/Monthly).

```
def calculate_product_winner(sales_df, product_df):
    # Join Sales with Product to get Margin
    # Margin = (Sell_Price - Buy_Price) * Quantity_Sold

    result = sales_df \
        .join(product_df, "product_id") \
        .withColumn("profit", (col("sell_price") - col("buy_price")) * col("qty")) \
        .groupBy("product_id") \
        .agg(sum("profit").alias("total_profit")) \
        .orderBy(desc("total_profit"))

    return result.limit(10) # Top 10 Winners
```

Listing 1: Pseudo-code for Winner Logic

4.2.2 Function 2: detect_loss_products()

Goal: Identify products that are costing money (Dead stock or Expiring soon).

```
def detect_loss_products(stock_df):
    current_date = now()

    # Condition 1: Expiry date is within 7 days
    # Condition 2: No movement (sales) in 30 days

    loss_list = stock_df.filter(
        (col("expiry_date") < date_add(current_date, 7)) |
        (col("last_movement") < date_sub(current_date, 30))
    )

    return loss_list
```

Listing 2: Pseudo-code for Loss Logic

4.2.3 Function 3: apply_client_reduction()

Goal: Automatically flag clients eligible for discounts.

```
def apply_client_reduction(client_df, current_sales):
    # Threshold for discount
    REVENUE_THRESHOLD = 10000

    updated_clients = client_df \
        .withColumn("new_total", col("total_revenue_generated") + current_sales) \
        .withColumn("eligible_promo",
            when(col("new_total") > REVENUE_THRESHOLD, True)
            .otherwise(False)
        )

    return updated_clients
```

Listing 3: Pseudo-code for Reduction Logic

5 Technology Stack

Layer	**Technology**	**Role**
Frontend	React.js / Angular	User Interfaces (Comptable & Admin)
Backend API	Node.js / Spring Boot	CSV Parsing, Kafka Producer, API
Message Broker	**Apache Kafka**	Real-time data decoupling
Processing	**Apache Spark**	Analytics, Aggregation, Logic
NoSQL DB	**MongoDB**	Storing JSON results for Dashboards
Data Lake	**HDFS**	Archiving raw CSVs and Parquet files

Table 1: Project Technology Stack

6 Conclusion

This architecture ensures that the ”Comptable” can work with familiar tools (Excel/CSV) while the ”Admin” receives high-level, real-time strategic data processed by a robust Big Data pipeline. The separation of concerns via Kafka ensures the system is scalable and fault-tolerant.