Warehouse Reorders System

Arpino Nicola[1], Culpo Annamaria[2]

[1]nicola.arpino@studenti.unitn.it

[2]annamaria.culpo@studenti.unitn.it

https://github.com/nicolatosh/big-data

Abstract

The project develops an automating reorders system for the warehouses of a large supermarket chain, including a simulation framework which provides a realistic scenario to analyze such a system. Starting from the transactions of the customers, the quantities of the products in the warehouses of the retail shops decrease. When the quantity of an item in a warehouse goes below a certain level (called *Rop* - reorder point), a reorder message is sent to the coordinator of the entire supermarket chain, referred to as the "supplier". It provides restock for retailers, decreasing also its warehouse stock of the same quantities of the products sent. At the end of the day, the supplier orders from its provider(s) the items whose stock level has fallen under the *Rop*. Finally a *Rop* recalculation and a batch analysis take place.

Keywords

Kafka, spark, message queue, reorder point (rop), simulation

I. Introduction

The project assignment requires to develop a big data system for automating reorders for the warehouses of a large-scale supermarket chain. The simulation involves customers, shops and the supplier. Those entities have been created starting from real data. Unfortunately, the data regarding the products stored by supermarkets are private. To overcome this obstacle, we have chosen a well-known Italian supermarket chain, Esselunga, which is present in almost every province. We have created the list of its products parsing (*scraping*) its online shop, starting from an existing GitHub script.[1] The data about products has been collected and used to create a list of all products. These products have been employed to populate the inventories of the retail shops. Retailers have been realized parsing an online website[2] which provides information about the city, location, telephone number etc. Customers instead, have been generated from scratch using lists of common names. Each of them has a unique identifier and optionally a fidelity card.
In this way, we have created some entities' datasets from the ground-up, overcoming the initial constraint of private data. The implementation of the system is based upon this built-ad-hoc Esselunga scenario, but it could be easily generalized and adapted to any other supermarket data warehouse.

II. System Model

A. *System architecture*

Description of the pipeline (attached on the bottom of the report):

- The first stage of the pipeline is the ingestion of the data regarding the products sold by Esselunga through its online shop and the locations of the retails per province. Moreover there is the generation of different pools of customers that will be later used in the creation of transactions.
- All the successive stages of the pipeline are encapsulated into Docker containers
- The data files obtained from scraping Esselunga and Puntivendita (e.g *.htm* and *.har*) websites have been collected, processed and then saved as JSON files into a MongoDB database. For example, from the .har files coming from the Esselunga online shop, a JSON list of items has been created.
- A message-queue architecture has been employed using Kafka to simulate realistic transactions and interactions between the players involved:
    - The customers of every retail shop represent Kafka producers. They send to the Kafka broker transactions corresponding to the lists of products they have bought.
    - The readers of the transactions, which run as Kafka consumers, are the retail shops (e.g., Esselunga di via Milano BS, Esselunga di Roma Prenestino etc.). They use the valid transactions to decrease the quantities in their warehouses and they discard the failed ones.
    - Valid transactions are stored in the Redis in-memory database by a transaction-processor component for later analysis.

- Retailers act also as producers: they send an order request message as soon as the stock level of an item goes below the *Rop* threshold.
- At the end of the simulation trial, the supplier (thought as the coordination centre of the entire supermarket chain) acts as a Kafka consumer and reads all the order request messages coming from the retailers. Also the supplier acts as a producer, sending the restock messages to the retailers.
- The transactions processor, on behalf of the supply chain, performs the batch processing of the data, computing some useful KPIs and calculating the Rop (reorder point) through the use of Apache Spark. The results are stored in JSON format in the statistics collection of MongoDB. Rop values are later updated by the chain itself.
- The order request messages are used by the supplier to decrease the quantities of the corresponding products from the general warehouse. If the remaining quantity of a product (stock level) is lower than the *Rop*, the supplier orders it from its provider and performs the restock. The communication between the supplier and its providers is not shown in the pipeline because it has not been implemented with realistic behaviour, since we have preferred to focus on the management of retailers-supplier communications.

B. ***Technologies***

The core technology used to implement the system is Docker, the software platform that packages other softwares into standardized units called containers[3]. In this way we can take advantage of isolation, portability and well-known community supported images. In particular we created a stack made up of three sub-stacks using docker-compose technology:

1. docker-compose-multinode
2. docker-compose-mongo
3. docker-compose-redis

The *multinode* file was used to create the Kafka cluster which includes two brokers and two Zookeeper nodes. Kafka is one of the most popular message queues software. We have used it to manage the interactions between all the parties involved e.g customers, retails and the supplier. Zookeper is necessary to run properly a Kafka architecture because it 'keeps track of status of the Kafka cluster nodes and it also keeps track of Kafka topics, partitions etc.'[4]. We have chosen such a message queue technology for the following reasons:

- Capability to manage complex and heavy messages (we used the serialization/deserialization of JSON messages)
- Horizontal scalability
- Possibility to use Pub/Sub pattern with multiple consumers and multiple producers

The *mongo* compose file allows the creation of a MongoDB cluster made of several ReplicaSets properly sharded. The cluster has been mainly used for storing inventories for each retail store. Setup can be configured so that each retail store has at least one dedicated shard. Since the original data we manage (e.g items present in the Esselunga online shop), are in Json format and the query we planned to develop were not too complex for a no-sql database we chose to adopt MongoDB. Moreover thanks to the multi-document transactions[5] with ACID guarantees, we found it suitable to manage bulk updates for the customer's transactions. Finally such a cluster can be horizontally scaled by simply adding more shards.

Last but not least, the *redis* stack provides a Redis replicated multinode database, which is a key-value, 'open source, in-memory data store'[6] that we have used to store the valid transactions coming from the customers. The choice fell on Redis because it provides the management of a high volume of data (e.g. transactions coming from all the retailers in parallel).

Another technology which has proved to be very useful for our project is Apache Spark, that is defined by its developers as 'a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.'[7] We have used it to do a batch processing of the transactions of all the retailers at the end of the stimulation trial. Then we have extracted information useful to elaborate some KPIs, like the product sold the most, the client who has spent the most etc.

III. Implementation

The simulation framework, as anticipated before, provides a realistic scenario in which customers, retailers and the suppliers can exchange messages using Kafka message queue system in a pub/sub fashion.

I. *Inventory*

The inventory has been implemented taking inspiration from a realistic management system for supermarkets. In short, the data analysis of the inventory needs to provide insights about how to avoid overstocking and understocking[8]. To do so, the *Rop* calculation gives tangible help. Let's have a look at the figure below.

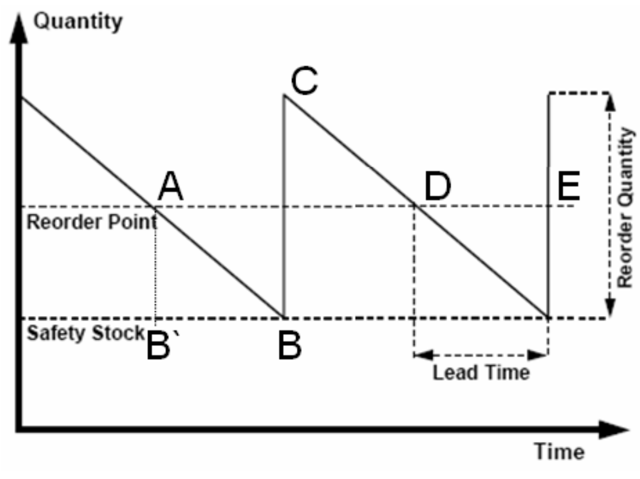


Fig. 1 Reorder Point (ROP) and input parameters for its calculation

The **reorder quantity**, also called **stock level**, is the quantity of an item that is present in the warehouse when none has bought a piece of it yet. The **reorder point** is a threshold that points out the necessity of a reorder when the stock level becomes lower than its value. The quantity that the retailer must order is exactly the reorder quantity, because from the moment of the reorder to the restocking (lead time) the stock level will be decreased up to the safety stock. The **safety stock** is the quantity that should never be eroded, because it is "the amount of extra inventory you keep just in case you don't receive a new shipment within the specified lead time".[9] In our project, the safety stock has been set to zero for simplicity. In the *Rop* formula another parameter used is the **daily sales velocity**, which "measures how many items you sell each day". Either for the retailers' warehouses or for the supplier's one, the *Rop* is calculated as follows:

$$ROP = (Daily\ sales\ velocity\ *\ Lead\ time) + safety\ stock$$

The inventory is a collection of products, for every product there are:

- ***Description***: brand and type (e.g: 'Esselunga, vitello fegato a fette')
- ***Category***: it can be 'carne', 'Confezionati alimentari', 'frutta_e_verdura', 'gastronomia', 'latticini_e_formaggi', 'pane_e_pasticceria', 'pesce_e_sushi', 'surgelati_e_gelati, 'vegetali'
- ***Price***: paid by the final customer for a single unit of product
- ***Qty***: quantity of product in a single unit (in kg or g)
- ***Price_per_kg***: price for a kilogram of product

II. *Customers*

Once a customer identity has been generated using *customers_generetor.py,* a Kafka **producer** can be instantiated through *customer_kafka_producer.py*. Such a script provides a wrapper to the *kafka-pyhton*[10] library and implements a Class that models the "retail client" behaviour. Each client is a Python Thread that can create transactions (shopping list of items that resembles a realistic receipt), and send those transactions to a retail store using a specific *kafka-topic* e.g *"city.shopid"*. The most relevant feature is the capability to activate a stream of transactions. Since we wanted to create a realistic scenario we provided the framework with a "***Turnout function***" e.g a mathematical function used to simulate the clients affluence in the shops:

$$affluence(t) = sin(t - 2) + 2 \quad t \in \{8, 20\}$$

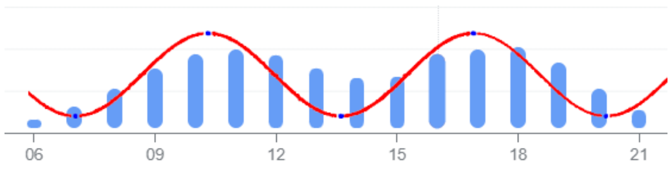


Fig. 2 Google representation of the average daily affluence of an Esselunga retail shop

The function above has been chosen starting from a visual analysis of the average affluence of some Esselunga retail shops. The blue bars in the graph represent the number of customers in the shop at every hour from 8:00 to 20:00, while the red line is the function we have used to model it.

Combining the client's transaction stream with a customizable turnout function it is possible to simulate a variable amount of transactions given a time slot.

III. Retailers

From the Kafka perspective the retail is both a **consumer** and a **producer**. The script *retail_kafka_consumer.py* contains the logic for the creation of the "retail shop" entity. Each entity is a Python Process, so multiprocessing has been enforced. The retail shop, once instantiated, performs the following tasks:

1. Listens for messages (transactions) coming from customers at the topic *"city.shopid"*
2. When a transaction arrives, it is validated and applied against the shop database.
3. Valid transactions are then published in a separate topic "*city.shopid.validtxns"* so that another component can receive them. The *transaction_batch_processor* saves the valid transactions in storage using Redis.

Validating a transaction means *enforcing an ACID update to the transaction items*. Since transactions are generated with a custom function in the customer producer script (customizable and tunable via params), some transactions might decrease the quantity of an item by a quantity which is larger than the actual stock level. Note that this situation typically does not happen in a real supermarket because customers need to physically "pick" an item and it must be available on the shelves!

Another job the retail has to do is to manage the "reorder messages". It has to send a reorder message (which contains the item universal product code *"upc"* and its quantity) to the supply chain by the topic "*shopid.requestorder"* and then listen for "restock messages" at the topic "*shopid.receiveorder"*. Orders are then "applied" towards the database as the same for transactions.

IV. Supplier

The supplier can be instantiated using the script *supply_chain.py.* The supplier is similar to a single retail store, the main difference is that the supplier has a much larger inventory. The principal task it has to accomplish is to manage all the orders coming from all the retail shops. To do so both a Kafka consumer and producer are adopted. Receiving an order means to decrease the stock level for the item involved; in this way the decreased stock is reserved for the retailer who issued the order request.
Once a simulation trial has finished, the supplier can *send_orders* to retailers and trigger the execution of a batch processing stage of the pipeline implemented with Spark. Finally once the batch processing is ended, it can update the *Rop* values of its own database.

V. Simulator

The framework can be started using the *simulator.py* script (details about how to run the simulation are available at the repository docs). It allows instantiating all the entities/components required to start the pipeline. The simulation can run in an interactive mode or in batch: the interactive mode creates different Gui applications per each process created. It means that producers and consumers are enabled to print to stdout in a dedicated console the exchanged messages, in this way they make visible the flow of messages.
There are some configuration parameters to tune the simulation, they are both documented in the code and on the project repository. Generally speaking, there are other components/scripts used to make the whole framework running, such as CRUD managers for databases and utilities. They are not as relevant as other components we discussed here; their documentation is available online.

VI. Batch processing

In the file "*batch_transactions_processor.py*" we have used the library Pyspark to perform Spark transformations and actions on all the transactions at the end of each trial. The batch processing starts from the creation of a Spark dataframe, which has the transaction fields as attributes. Thanks to transformations like *"groupBy()"*, *"sum()"*, *"sort()"* etc. we have computed some useful KPIs (Key Performance Indicators), which have been returned as Spark dataframes. In order to manage them during the simulation, we have converted them into tuples with *"reduceByKey()"* transformations.

IV. Results

We successfully deployed the big data pipeline thanks to the technologies mentioned before. The whole system runs in Docker containers so it is isolated and portable. The demo works as expected and it is capable of simulating multiple days of workload involving all the parties. We tested different configurations with the default city of *Como*, which has four shops. The system correctly manages up to millions of transactions per trial. Performances are in any case affected by the Python libraries and the hardware capability of the running machine. One of the main challenges of this project has been the proper implementation and configuration of the multinode Kafka architecture.

V. Conclusions

We have deployed a big data system for automatic reorders of the products in the warehouses of a supermarket chain of big dimensions, taking Esselunga as an example. The core of the system is the management of the message exchange between the players involved: the customers, the retailers and the supermarket chain coordinator (which we referred to as the "supplier").

A possible improvement of the project concerns the batch processing performed at the end of the day. In fact, the KPIs calculated using the transactions of all the retailers can be extended according to the necessity of the supermarket chain managers. Moreover, the ones computed in this report regard just a period of one day, but many useful indicators need data collected in mid or long periods of time (e.g.: year over year growth, inventory turnover etc.).[11] Also some forecasts about Rop, KPIs and demand indicators could be calculated to avoid overstocking and to plan efficiently the use of the staff.[12]

Another possible direction of improvement is the management of different lead times for each product. In this case, we should properly schedule the restock of the items for every retailer. Moreover, it could be useful for a supermarket chain to keep track of damaged goods, returns and expired items in order to maintain an inventory always properly updated.[13]

Finally, for a complete lambda architecture implementation, a Spark Streaming use-case could be designed to provide near-real time data processing.

Despite the acknowledgement of the presence of further implementations and some limitations, we are very proud of the results achieved with this project and we hope it could turn out to be useful to somebody else. This project has been a great challenge, but as Roy T. Bennett said, "Challenge and adversity are meant to help you know who you are. Storms hit your weakness, but unlock your true strength."

References

[1] M. Ferri. (2019) Esselunga scrape. [Online] Available: https://github.com/limi7break/esselunga-scrape

[2] Esselunga retail shops. [Online] Available: https://www.punti-vendita.com/esselunga.htm

[3] Amazon Web Service website. [Online] Available: https://aws.amazon.com/it/

[4] Cloud Kafka. [Online] Available: https://www.cloudkarafka.com/

[5] MongoDB transactions website. [Online] Available: https://www.mongodb.com/blog/post/mongodb-multi-document-acid-transactions-general-availability

[6] Redis website. [Online] Available: https://redis.io/

[7] Apache Spark website. [Online] Available: https://spark.apache.org/

[8] K. Jenkins. (2020) What is Overstocking? [Online] Available: https://www.shopify.com/ph/retail/overstocking-causes-and-prevention

[9] Reorder point and stock parameters. [Online] Available: https://www.shopify.com/ph/retail/reorder-point

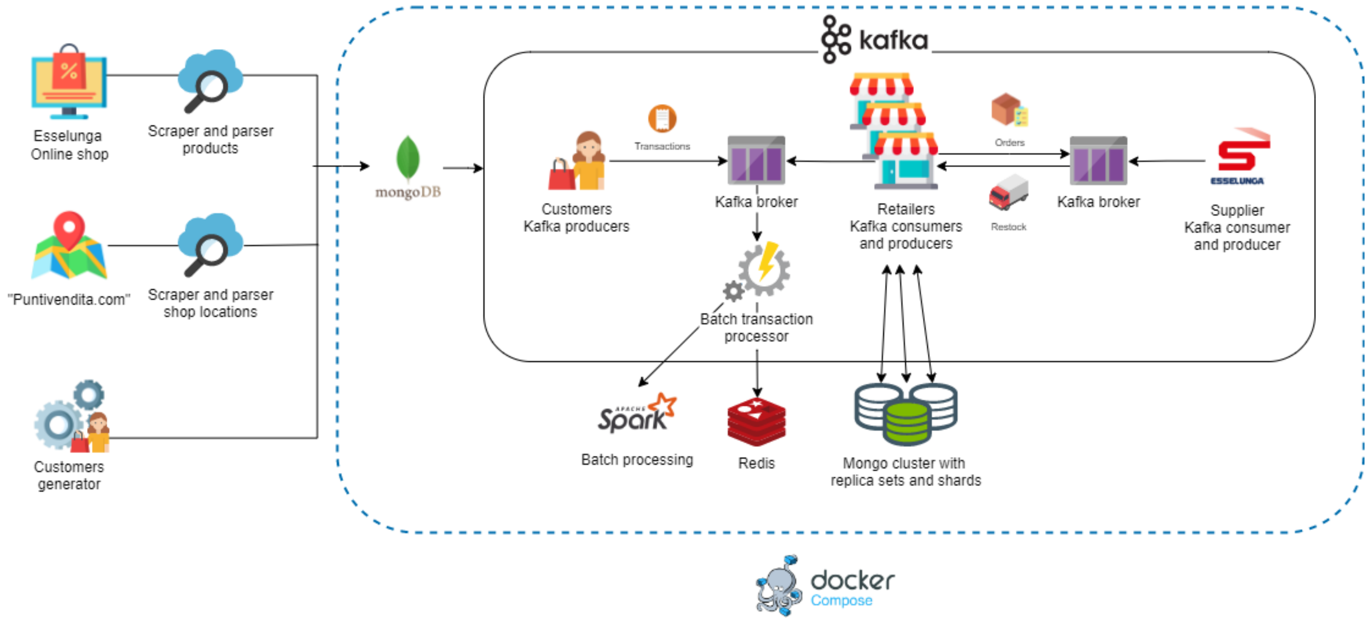
[10] Python library for implementing Kafka architecture. [Online] Available: https://kafka-python.readthedocs.io/en/master/

[11] Retail metrics and KPIs. [Online] Available: https://www.vendhq.com/blog/retail-metrics-and-kpis/

[12] M. Kay. (2021) How to forecast demand for your retail store. [Online] Available: https://www.shopify.com/ph/retail/demand-forecasting

[13] J. Campbell. How do Grocery Stores keep track of inventory? [Online] Available: https://thegrocerystoreguy.com/how-do-grocery-stores-keep-track-of-inventory/

PIPELINE


Esselunga
Online shop
Scraper and parser
products
"Puntivendita.com"
Scraper and parser
shop locations
Customers
generator
mongoDB
kafka
Transactions
Customers
Kafka producers
Kafka broker
Retailers
Kafka consumers
and producers
Orders
Restock
Kafka broker
ESSELUNGA
Supplier
Kafka consumer
and producer
Batch transaction
processor
Spark
Batch processing
Redis
Mongo cluster with
replica sets and shards
docker
Compose