FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Event-Driven Real-Time Streaming Approach for Big Data, applied to an End-to-End Supply Chain

Inês Teixeira

DISSERTATION

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Mendes Moreira, PhD

Co-supervisor: Luís Roque

July 25, 2019

Event-Driven Real-Time Streaming Approach for Big Data, applied to an End-to-End Supply Chain

Inês Teixeira

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Doctor Gabriel de Sousa Torcato David

External Examiner: Doctor João Vinagre

Supervisor: Doctor João Mendes Moreira

July 25, 2019

Abstract

Big Data has been a recent global transformation of smart technology paving the way to just about everything around daily lives. Although this concept has been around for a few years, nowadays, most organizations have come to understand the significant value they get from capturing and applying analytics to all the data that streams into their businesses.

Big data is really just large amounts of data near real-time or real-time from various sources. And "big" in fact can mean, not the actual size of the data, but in fact, can be relative to the size of a company, meaning more data than the company can actually handle.

It is in this context, that the dissertation was defined and developed. HUUB, a small start-up, born in 2015, providing an integrated ecosystem for fashions brands manage all the supply chain, end-to-end, realized how much advantage could take from experiencing Big Data and use to improve daily business and decisions.

The company offers a Software as a Service, SaaS, named *Spoke*, and it is interested in improving the client experience while making the best decision to gain new clients and grow its competitive advantage.

Taking advantage of the recent transition of the company from a monolithic architecture towards microservices, and using Event Sourcing as a way for communicating between services through a Kafka Message Broker, the main goal of this thesis was to study all these technologies and provide for a solution that would allow the company to ingest relevant data in real-time, with high throughput, capable to handle Big Data.

A few steps within the company had already been made, but had took more that one hour, with low throughput, and was still dependent from legacy data. At the end of the dissertation, the solution allowed for ingestion of events, around, 100 times higher in the beginning. Going from a couple of events consumed per second, and available for reporting one hour later, to hundreds of events being consumed per second, and made available in the data warehouse within a few minutes. This was made possible through a process of micro-batching, that not only allowed for availability of data in large amounts available only in a few minutes.

To conclude, this dissertation opens up a new way for real-time data ingestion and reporting, being this subject so poorly reviewed in the literature. Describing a possible architecture, discussing technologies and experiments for an improved way for consuming data in less time, and more throughput.

Resumo

Recentemente, *Big Data* tem sido considerada uma transformação global da tecnologia inteligente, abrindo o caminho para solucionar variados problemas do quotiadiano. Embora esse conceito já exista há alguns anos, hoje em dia, a maioria das organizações tem compreendido melhor o valor que esta tecnologia pode trazer na captura e aplicação de análises a todos os fluxos de dados do negócio.

Big data são grandes quantidades de dados, quase em tempo real ou em tempo real, vindo de várias fontes. E na verdade, "*Big Data*", pode significar, não o tamanho real dos dados, mas a relação entre o tamanho de uma empresa e os dados que esta pretende manipular.

É nesse contexto que a dissertação foi definida e desenvolvida. A HUUB, uma pequena empresa nascida em 2015, que fornece um ecossistema integrado para marcas de moda que gerenciam toda a cadeia de abastecimento, percebeu a vantagem que traria lidar com esta tecnologia, em tempo real, para melhorar na decisão diária dos fluxos e de métricas de negócio.

A empresa oferece um Software como Serviço, SaaS, chamado textit Spoke, e está interessado em melhorar a experiência do cliente, enquanto toma a melhor decisão para conquistar novos clientes e aumentar sua vantagem competitiva. Por todas as razões mencionadas, o objetivo principal desta dissertação é melhorar a forma como a empresa lida com dados em tempo real e alta taxa de transferência.

Aproveitando a recente transição da empresa de uma arquitetura monolítica para microsserviços, e utilizando *Event Sourcing* como forma de comunicação entre serviços através de *Kafka Message Broker*, o principal objetivo desta tese foi estudar todas essas tecnologias e fornecer uma solução que permita à empresa ingerir dados relevantes em tempo real, com alta taxa de transferência, a custos reduzidos.

A empresa já tinha feito alguns avanços dentro desta tema, mas os dados representam uma latência com mais de uma hora, com baixo rendimento, que dependiam de dados antigos. No final da dissertação, a solução permitiu a ingestão de eventos, cerca de 100 vezes maior que no início. Sendo que no início, apenas um ou dois eventos eram capazes de ser consumidos por segundo, e estando disponível para análise uma hora depois, para centenas de eventos a serem consumidos por segundo e disponibilizados no armazém de dados, em poucos minutos.

Tudo isto foi possível por meio de um processo de microtatching, que não apenas permitiu a disponibilidade de dados com alta taxa de transferência, sem comprometer a latência, e a custos reduzidos para a empresa, uma vez que os dados são armazeandos num armazém de dados na nuvem.

Para concluir, esta dissertação abre um novo caminho para a ingestão e comunicação de dados em tempo real, sendo este tema está fracamente representado na literatura, sendo tão recente. É ainda descrito uma possível arquitetura de dados para o caso tratado, discutodas as diferentes tecnologias e são feitas ainda experiências de forma a comprovar a nova forma proposta de consumir eventos, usando um Algoritmo de *Batching* com alta taxa de transferência e baixa latência.

Acknowledgements

First, I would like to thank my supervisor, Prof. João Moreira, for accompanying me throughout the development of the dissertation and for always showing concern and interest in my work, as well as being always available for discussions.

Secondly, I would like to thank my co-supervisor, Luís Roque, from HUUB, for creating a welcoming environment for me to work and develop the dissertation, for showing interest and always value in all my opinions and ideas.

Additionally, I would like to thank everyone who works at HUUB, for being always so pleasant and friendly, and create a great work environment. However, I would like to thanks in particular to Guilherme Gomes and Diogo Basto for helping me with all my questions and guide me through the initial process of getting to know all the technologies and process at HUUB.

I would also like to thanks Beatriz Guerner and Vasco Faria for tackling this challenge with me, as master thesis interns at HUUB, and all the coffees and breaks full of words of encouragement in the hardest times. I became a better arrows player due to you.

Moreover, I would like to thank all my friends at FEUP for struggling with me during these months of dissertation making. In addition to it, I thank the 5 years I spent with you all, with all the straight nights working and all the parties, where the best memories were created.

Finally, I would like to thank my family. My parents and my brother, for all the support they offered me my entire life. For always letting me be what I wanted, never forcing me to follow a certain path and believing in the savvy adult I was trying to become. All due to the amazing examples you always showed me.

Inês Teixeira

“Any time scientists disagree, it’s because we have insufficient data.
Then we can agree on what kind of data to get; we get the data;
and the data solves the problem. Either I’m right, or you’re right,
or we’re both wrong. And we move on.”

Neil deGrasse Tyson

Contents

List of Figures

LIST OF FIGURES

List of Tables

Abbreviations

AI	Artificial Intelligence
BI	Business Intelligence
CDC	Change Data Capture
DB	Database
DPFlowR	Data Processing Flow Regulator
DW	Data Warehouse
ETL	Extract-Transform-Load
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
ID	Identifier
IT	Information Technology
OLAP	Online Analytical Processing
RDBMS	Relational Database Management System
RTDW	Real-Time Data Warehouse
SQL	Structured Query Language
VDF	Vertical Distribution of Functionalities
SaaS	Software as a Service
SFlowR	Data Source Flow Regulator
SK	Surrogate Key
WFlowR	Warehouse Flow Regulator

Chapter 1

Introduction

This dissertation was conducted at HUUB, a Portuguese start-up, founded in 2015, focused on logistics and supply chain management, for distinctive fashion brands. HUUB is responsible for handling all stages of the supply chain in all sales channels: wholesale and e-commerce.

Being responsible for the supply chain of dozens of brands, and accountable for delivering the best business solutions regarding costs of transportation amount of stock available and forecasting methods, has a tremendous component of reporting and analysis, that must be met as real-time as possible.

1.1 Context

With the explosion of data, new ways of structure it, store it and analyze it has to be defined. In the industry is tightly connected to controlling complex technological processes, facilities or business decisions [DDM14].

The topic is often associated with Data Analytics [DDM14], since Big Data is often desired by companies in order to create new knowledge and help with decision making instead of using human expertise and *intuition* [MB12].

According to a governmental study, conducted by Kaisler et al. (2013) [KAEM13], there are three big issues when dealing with such amounts data, being them the storage, the management, and processing issues. Concluding that to process 1K petabytes of data would require a total processing time of roughly 635 years. Thus, effective processing of exabytes of data will require extensive parallel processing and new analytics algorithms in order to provide timely and actionable information.

Analogous to Big Data, new architectures have, recently, started to appear, in the field of development of big systems, known as microservices. As defined by Sam Newman, Microservices are small, autonomous services that work together [NMMA16], and further explored in the State of the Art of this dissertation, in chapter 2.1.

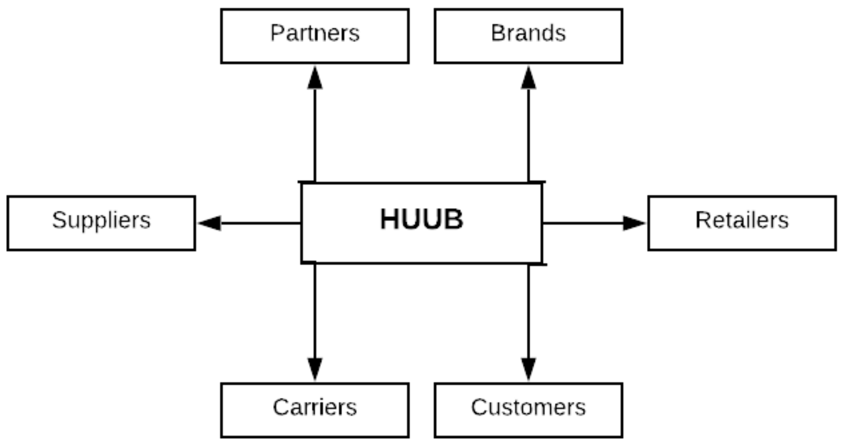

Figure 1.1: HUUB's posititioning relative to its stakeholders.

With the increasing number of microservices, and the number of distributed systems a new architecture started to be adopted, Event Sourcing, where entities keep their entire history in an event log, and are usually asynchronous, message-based interaction of loosely coupled, modular components and, when highly distributed, event-driven architectures represents a significant performance bottleneck and also a single point of failure [EMH+17], and will be discussed in more detail further in the dissertation.

These recent technologies are adopted by big enterprises, such as Netflix, Spotify [NMMA16] and LinkedIn. Additionally, these technologies have been, recently, is being adopted by the Portuguese logistics start-up HUUB.

The company provides logistics services by integrating all entities and steps of the supply chain into one dynamic ecosystem: customers, end users, suppliers, alliance partners, through a web platform, called *Spoke*.

This platform grants full visibility and control to all end users, being brand owners, warehouse workers, or analysts. Bringing together, in just one platform, as seen in Fig. 1.1, a business-oriented layer, by connecting brands and sales channels, but also supply chain management layer, integrating vendors, inbound transport, warehouse network, outbound transport, and delivery.

HUUB is Product Driven, being SPOKE a big part of their business model, it is crucial for the company to present a well designed and user-friendly application, hence a constant pursuit for state of the art technologies and architectures is a big concern for the Tech Team. Additionally, scalability is key for continuing to provide the best service possible, while the company continued to grow. To ensure this, recently, the company has taken some decisions regarding the architecture of the application, moving towards a microservice architecture. However, a more deep explanation of this transition will be presented in the next section.

Secondly, it is Data Driven, since all data collected is important, considering that it can improve business performance by supporting decision making. *Spoke* intents are to provide actionable feedback and business insights based on knowledge, rather than instinct. To achieve this, HUUB has a strong Business Intelligence team, where data is treated to provide historical, current and predictive views of business operations.

Finally, it is AI Driven, focusing on guaranteeing optimal efficiency through Artificial Intelligence and Data Science, managing every decision in the supply chain, through forecasting, machine learning, and deep learning models.

Spoke, HUUB's software started as a monolith meaning all components are combined into one single tiered platform. This was the best solution, in the beginning, since the start-up was still in an early stage, having only a few clients and a small set of features available on the application.

However, the company started to grow and incorporate more features into the application, which started to create some problems in the web platform, whilst for a company like HUUB, in a stage of exponential growth, it is important to deliver the best software possible to make it easier for end-users and improve everyone's business. With the increase of overall features and data transactions, the monolith started to perform below the expectations and the business needs. Considering this, a new approach had to be studied. Following current trends mentioned above, the monolith was set to be broken into small microservices, where each service would be built around a business capability and could be deployed independently.

This new decision brought a lot of changes to the architecture of the application, as well as how data flows would be ingested in order to continue to provide valuable metrics for decision making, at a faster pace, creating new data sources and new ways of data ingestion, through Event Sourcing and Event-Driven ETL, which will be discussed further.

1.2 Motivation

As mentioned before, HUUB holds a lot of responsibility regarding business decisions not internally, but for the entire supply chain of their clients.

Competitiveness of the 21st century, means, not only big companies, but also small to medium enterprises, struggle to incorporate into their core business technologies, that allow them to handle and create competitive advantages, such as Big Data, analytics and reporting, recommendation systems, and others.

Consequently, being HUUB a start-up in the most important years to succeed, it has to deliver the best service possible. Being in the process of microservices migration, some interesting optimizations could be done at the data ingestion level. Allowing the company to perform reporting and analysis much faster than was performing.

1.3 Goals of the Dissertation

The aim of the company is to provide the best logistic service for its brands, controlling the entire supply chain, while increasing profit and automating processes.

Taking into consideration the current changes in the system architecture of the company, and the need for consuming data as fast as possible for analysis and reporting, the goals of the dissertation are:

- Study all the technologies regarding this new architectures, underlying advantages and disadvantages for each technology.
- Observe what technologies are being applied at HUUB and how studying all the process as-is of the company.
- Design a valid architecture for ingestion of events, by streaming, to a data warehouse, applied for Big Data.
- Implement the solution designed at the company and see how the process of ingestion of events improves in the company, experimenting and validating the approach designed.

1.4 Structure of the Dissertation

Additionally to the Introduction, this chapter contains more 6 chapters.

In chapter 2, the author lays out background information, several concepts, and definitions about the subject matter.

In chapter 3, there is an analysis of the state of the art in terms of real-time data warehouses, new approaches for data ingestion and new Extract, Transform and Load tools.

In chapter 4, describes the architecture of the company of the dissertation development. Later, exposes the problem statement and set of achievements meant to accomplish.

In chapter 5, describes the implementation of the problems defined in the chapter before, at HUUB.

In chapter 6, it is described as a set of experiments, with a range variety of cases, realized at HUUB, showing the goals achieved.

Finally, in chapter 7, it is concluded the difficulties during the development of the dissertation, the contributions of the work realized and details possible future work.

Chapter 2

Background

In this chapter, it will be presented a few background knowledge of subjects regarding the thesis development, that are not commonly known.

2.1 Monolithic and Microservices Architecture

Microservices are the opposite of the monolithic architecture, where applications are usually deployed as a single package on a web container [TLPJ17]. The server-side of the application handles HTTP requests, executes domain logic, retrieves and updates data from the database and selects and populates HTML views to be sent to the browser. This design approach can rapidly become a bottleneck to fast continuous build, test, and deployment [SML18].

The modification of one single feature, in a monolithic system, involves building and deploying a new version of the server-side application, which also means scaling has to be made through the entire application, which takes greater resources [Lew14].

Within the last few years, a new architectural trend has been emerging and becoming established for a huge set of applications . This architectural trend is Microservice architecture and big companies like Netflix, Spotify, Twitter, and Amazon are already delivering their core business through microservice-based-solutions [STV18].

This architecture trend was first, formerly, introduced by Lewis and Fowler in their blog, in 2014, defining Microservices as "an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API" [Lew14]. Microservices are loosely coupled architectural elements that can be independently deployed by fully automated machinery, often achieved by exploiting lightweight, container-based platforms [STV18], like Docker [1]. This results in two different architectures as seen in Fig. 2.1

Each microservice should be an isolated business functionality, allowing the service to be independent and autonomous and be easily replaceable when no longer fits the business core, without crashing other services important for the application [HB16].

[1]https://www.docker.com/resources/what-container

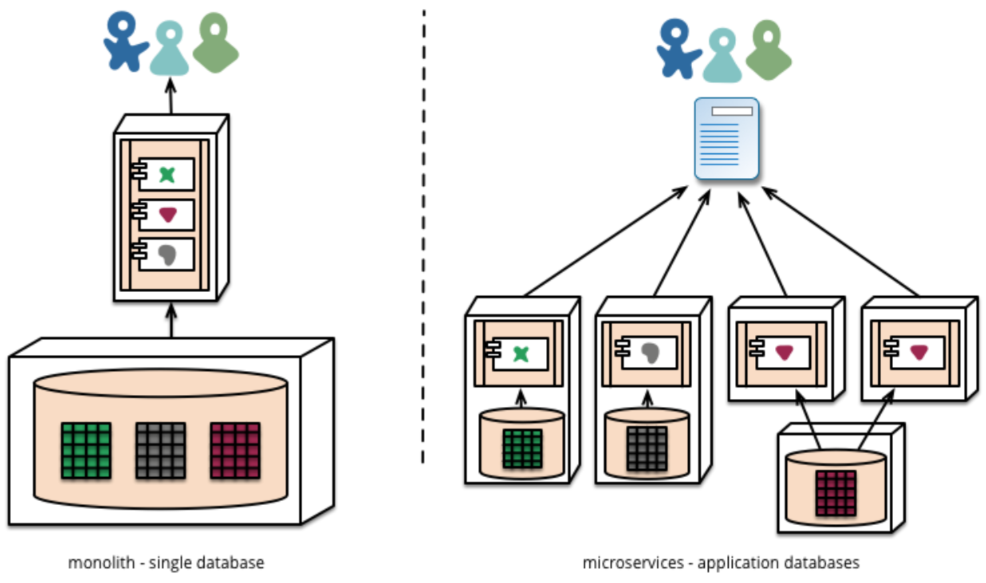
monolith - single database
microservices - application databases

Figure 2.1: Comparison between a monolithic and microservices architecture.
Extracted from [Lew14]

Regarding the migration of a monolithic architecture to a microservice architecture, in a survey [TLPJ17], conducted during the 17th International Conference on Agile Software Development (XP), among developers experienced in microservices, whose goal was to analyze the motivations as well as the pros and cons of migrating from one architecture to another, unveiled the need for extra machinery, imposing substantial costs and effort in the beginning, when transitioning to a microservice architecture.

Moreover, some issues in the adoption of the new architecture were also outlined. Regarding some technical issues, related to the fact this migration requires experienced developers. Also some economic issues, due to the fact of the aforementioned technical issues and the need for a DevOps infrastructure. Finally, some psychological issues, since changing completely the whole system is risky and depends on developers with an open-minded and inquisitive mindset [TLPJ17].

Nonetheless, the main benefits in the adoption of the architecture were confirmed by the participants in the survey, who in the beginning, when considering the issues claimed, thought the Return On the Investment (ROI) would be perceived as an issue, but on the contrary. In the long run, the need for maintenance is smaller, than in a monolithic architecture, and the scalability easier, thus the extra effort is compensated after a period of between two years (33% of the interviewees) and three years (66% of the interviewees) [TLPJ17].

2.2 Event Sourcing

Event sourcing refers to all changes of states in an application are stored as a sequence of events, allowing to not only query events, as well as, cluster all events building an event log, and reconstruct past states [Fow05].

The basic premise is to guarantee every change in the state of a application is captured as an event object, and these events are stored in an ordered sequence.

This architecture is more straightforward and less consuming than querying databases to find the current state of an application entity. Moreover, to build a historical change of states, it is necessary a persistent database, additional to the main database. Whenever a new query update happens in the main database, a new row is added to the historical database. With Event sourcing, for each change in the application, an event is produced and stored immeadiately in a pipeline of ordered events [NMMA16].

There are a few more advantages to event sourcing [Fow05]:

- Do a complete rebuild, discarding the current state of the application and rebuild it by re-running all events stored in the event log.
- Determine the state of the application at a specific time, by re-running the events until that time.
- Replay events in case of past events have been incorrect or computed wrong.

2.2.1 Apache Kafka

”Kafka was created to address the data pipeline problem at LinkedIn. It was designed to provide a high-performance messaging system that can handle many types of data and provide clean, structured data about user activity and system metrics in real time. Data really powers everything that we do.”
—Jeff Weiner, CEO of LinkedIn

Apache Kafka is commonly designated as a "distributed messaging system" or, more specifically, a "distributed streaming platform" [Neh10]. A *publish/subscribe* messaging system is when a piece of data follows a pattern, defined by the publisher (sender), classifying the type of message. Afterward, the receiver subscribes to receive certain types of messages, instead of the publisher directly send the message to the receiver. To facilitate this process, there is a central message broker.

A *topic* can be compared to a table, in relational databases, where each topic has a type of message. A Kafka *message* is an array of bytes and a message can have a specific format, referred to as *key*. Kafka is described as "distributed" because a topic can hold one or more *partitions*, and the keys are used to differentiate how messages are organized through the partitions, in a more controlled way.

Messages are stored according to the time they were created and sent to the topic, however, if a topic contains more than one partition, Kafka can provide total order over records only within a partition and not between different partitions in a topic, as described in Fig. 2.2

This data distribution also provides additional protection against failures, as well as significant opportunities for scaling performance, since each partition can be hosted on different servers

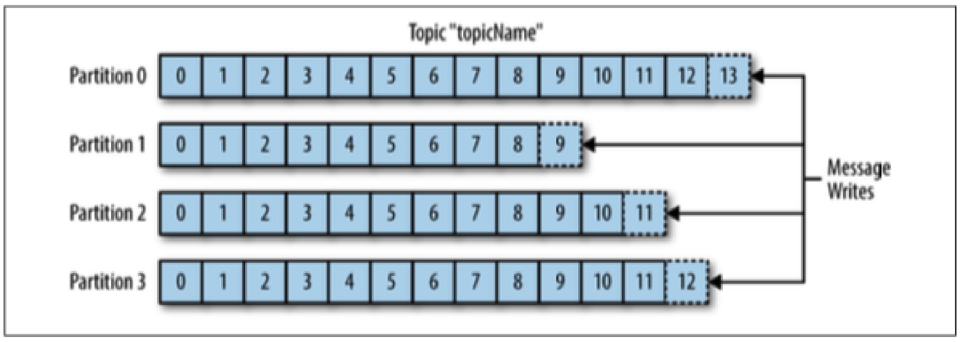

Figure 2.2: Representation of a topic with multiple partitions.
Extracted from [Neh10]

and thus a topic can be scaled horizontally and throughout multiple servers, improving overall performance.

Users of the system are designated as *Kafka clients*, and there are only two types: *producers* or *consumers*.

Producers are responsible for creating new messages and publish these messages to specific topics, negligent of how many, or which, consumers will it reach. As mentioned above, messages can be written to specific partitions if a key is defined and messages with the same key will be stored in the same partition.

Consumers read messages. Consumers label themselves with a consumer group and subscribe to one or more topics and messages are read according to the order they were produced. While messages are read, a few metadata stores the position of the last message consumed, called the *offset*. Storing this offset allows consumers to stop and restart, within a partition, without losing track of the last message it consumed. In Fig. 2.3, it can be observed how a consumer group, with three different consumers, are distributed between four different partitions. Consumer 1 is responsible for consuming messages from two different partitions. Furthermore, if a consumer fails, the rest of the consumers in the group rebalance and partitions affected are taken over by other consumer members.

In conclusion, a Kafka server is a *broker*, which receives messages from producers. An offset is assigned to each producer and messages are stored on disk. Consumers connect to this central broker and read messages, by order, from partitions in a topic. A set of brokers is called a *cluster*. A partition is owned by a broker in the cluster and it is the leader of the partition. A partition can be replicated through multiple brokers, providing redundancy in case of failure. Messages from a specific partition always have to be written to the partition leader, but a consumer can be read from a replication.

Kafka holds a lot of advantages when comparing to other message brokers systems. Some of these advantages are the retention factor, which means messages can be stored for a long period of time and can be changed at any time. It allows for scalable solutions, by having multiple clusters, which hold multiple brokers, and also distribute messages through partitions inside the same topic, being flexible and easy to handle any amount of data.

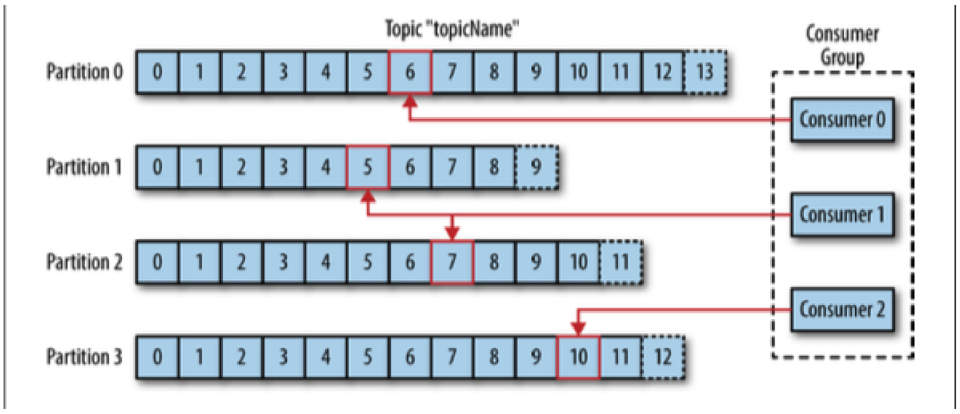

Figure 2.3: A consumer group reading from a topic.
Extracted from [Neh10]

2.3 Data Warehousing

Data warehousing was defined by Bill Inmon in 1990, “A warehouse is a subject oriented, integrated, time variant and nonvolatile collection of data in support of management’s decision making process” [Inm02].

Data warehouses are Subject Oriented since a DW can focus on one single business. For example, inside a company, it is possible to analyze data from sales specifically. It is Integrated, due to the fact data in DWs comes from all different sources and must put this data into a consistent format, resolving naming conflicts and other inconsistencies. They are also nonvolatile, which means that once data is inside the data warehouse, it must not change. And, finally, Time-Variant, which means that, in order to discover new trends and find new solutions and patterns, a large amount of data is needed and it has to change over time [Inm02].

When designing a DW, the most important aspect is the issue of granularity.
"Granularity refers to the level of detail or summarization of the units of data in the data warehouse" [SA12], so the more detail there is, the lower the level of granularity. Therefore, it is one of the biggest issues in the DW environment, since it influences the amount of data it exists and, consequently, the type of query that can address it. However, granular data can be really useful, after all, within a corporation, different departments require different data and different levels of granularity. No data goes completely to waste.

Additionally, DWs can take enormous volumes of data from independent heterogeneous operational data-sources and create a single view of the organization. When comparing to traditional relational databases, the costs of building this architecture and maintaining the interface is mitigated over time, it is easily expandable and new subject areas can be always added [SA12].

However, the new business requirements demand up-to-second updated information, and more complex and enormous amounts of data, resulting in the need to come up with new ways of architect more complex Data Warehouses [BNG17].

When researching about warehousing, data modeling is essential, since it provides users to see how the data relate to one another. Mainly, there are two different and predominant designs, being the Relational Model and the Dimensional Model [SCT11].

The main difference between them is in the structure of the logical schema, while relational models represent themselves by the traditional relational schema, dimensional models are represented by using Star Schema [KR02]. However, there are other alternatives for dimensional models, the Snowflake Schema and the Star Cluster Schema [MK00].

- The **Flat Schema Model** is the simplest schema model for data warehousing. This solution minimizes the number of tables and usages of joins when querying data. Information is all normalized, ending up with one table for each minimal entity. However, there is a lot of redundancy, since entries are repeated across multiple tables. Despite the fact the number of tables is minimized, the complexity of each table increases given that the number of attributes also increases [SCT11] [KR02].

- The **Star Schema Model** is represented by one central table, the fact table, and smaller tables, called dimension tables, that *radiate* from the fact table, like a star. The fact table contains measurements, or metrics, interesting for the business, like amounts of sales [MK00]. Each dimension is aggregated to the fact table and, each dimension only has a one-dimensional table and cannot have sub-dimension tables, being represented in the fact table has a foreign/surrogate key. It represents a much simpler representation than traditional relational schemas. Dimensional models are most common when dealing with data warehousing, and described as "the only viable technique for designing end-user delivery databases" [SCT11]. The main reason to use star schema is the fact that it reduces the number of tables in the database, the number of relationships between tables and the number of joins that has to be used in queries [MK00], decreasing the level of complexity in the overall modeling.

- The **Snowflake Schema** is a star schema, but with dimensions expanded out, normalized. So, each dimension table can contain multiple independent hierarchies, and the child tables inherit all relationships to component entities and key attributes from the parent entity. However, some can argue that this schema is undesirable, since by adding new hierarchies it adds complexity to the schema, being more difficult to design and understand, and requires extra joins when querying. So the Star Schema and Snowflake Schema do not fulfill all needs to design the model, since on has fully collapsed hierarchies and it is too simple, the other has full expanded hierarchies and it is too complex [SCT11].

- The **Star Cluster Schema** is represented has the schema with the "minimal number of tables while avoiding overlap"[MK00]. It is a balance between the two last schemes and selectively separates hierarchical sub-dimensions. For each transaction, the entity is formed a fact table and hierarchies should be collapsed down until it reaches a component entity, then a sub-dimension should be formed. After sub-dimension is formed, collapsing starts again and when it reaches a component entity again, a dimension table should be formed

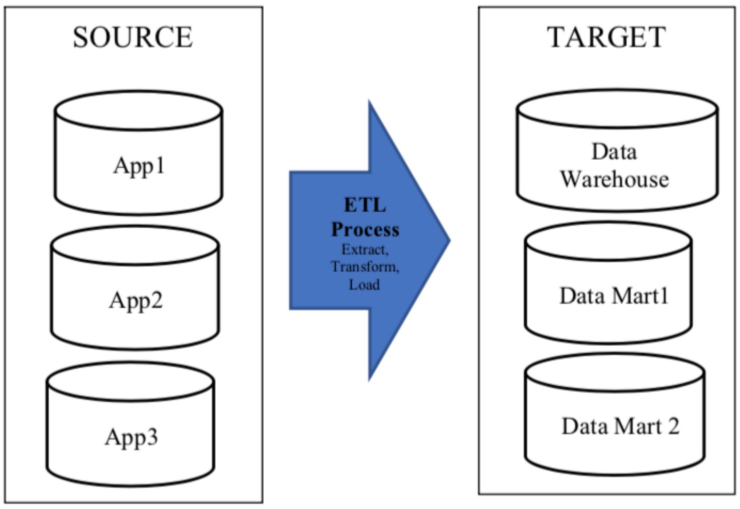

Figure 2.4: ETL System Flow in a Data Warehouse Environment
Extracted [SAS+17]

and so on [MK00]. This schema is the perfect balance between star schema and snowflake schema. Being simple while contributing to consistency between dimensions [SCT11].

2.4 ETL: Extract, Transform and Load

”ETL is a tool for technicians work in the area of business intelligence, which facilitated the consolidation of information in a central repository called data warehouse.”, described in the paper *Agile ETL*, by Xavier Cristiano [XM14].

Extract, Transform and Load (ETL) System is a broad presentation of data movement and transactional processes from extraction of multiple application sources, transforming data into dimensional fact table and conformed format to feed data into data warehouse environment such as data marts or other target systems, [SABTU] but also provides data from different sources to be analyzed and structured together [XM14].

It involves cleaning, integrating and staging the data before transporting and channeling the transformed data to target systems. Proved to be a choice standard for managing and sustaining the movement and transactional process of the valued big data assets [SAS+17]

ETL system needs to have the capacity to address change of directions and to support new data demand which is becoming more huge volume, endless streaming, a wider variety of types and sources, request for real-time user requirements and new analytics, extreme integration, availability of new technologies and more powerful tools [SAS+17].

Chapter 3

Literature Review

The topic of real-time data and event-driven systems are still poorly unexplored subjects in the literature, being mainly investigated by companies and IT individuals. As a result, the state of the art will be reviewing the work developed by companies, such as Netflix, Spotify, and LinkedIn, who are the great pioneers and examples to follow when stepping towards these architectures.

First, will be briefly described the evolution from Traditional Data Warehouses to Real-Time Data Warehouses, where some approaches to handling ETL will also be researched in the literature. Lastly, recent approaches to address Big Data and streaming of messages will be discussed, considering the recent trend of Event Sourcing.

3.1 Real Time Data Warehouse

Traditional Data Warehouses have a major drawback since data content is not updated according to business needs and bad decisions can be made. These situations do not correspond to the new business requirements which require up-to-second updated information to perform better-informed decisions and resulting in new ways to architect more complex Data Warehouses, such as Real Time Data Warehousing[BNG17].

According to the literature, there is no official definition regarding real-time data warehouses. However, it can be defined, in a more generic way, and according to the business's needs, as a system that represents the characteristics and the actual situation of the organization, so RTDWs are meant to contain current data, thus, the refreshing frequency plays a predominant role, but also the structure of the system needs some changing [BNG17].

3.1.1 Architecture of Vassiliadis

One of the first real-time architectures was proposed by Vassiliadis[VS08], in 2009.

In this architecture, there is a continuous loading as opposed to the periodic batches in a traditional approach. It has three main parts. The Data Source Area, the Data Processing Area, where data is processed and transformed and is added to the Data Warehouse, the third part of this architecture[VS08], the full architecture can be seen at Fig, 3.1.

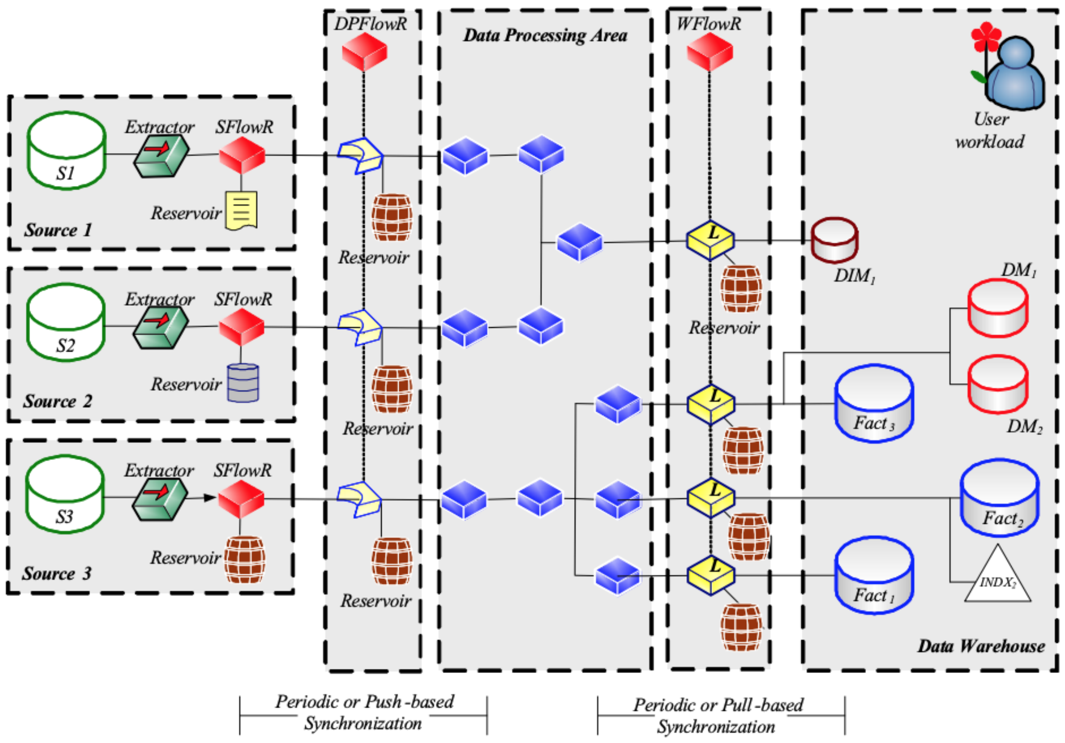

Figure 3.1: Real-time Data Warehouse Architecture proposed by Vassiliadis.
Extractred from [VS08]

What distinguishes this architecture from traditional architectures is the fact that each part hosts a Flow Regulator Module, that has different functionalities for each part of the process. The Data Source Flow Regulator (SFlowR) is in charge of the identification of relevant changes and extracts data at periodic or convenient intervals, depending on the policy chosen by the administrators of this system. The Data Processing Flow Regulator (DPFlowR) is responsible for deciding which source is ready to transmit data, but in fact has a lot of tasks, depending on the needs of the system, but mainly acts as an intermediary between the Data Source Area and the Data Warehouse, and data is prepared to go through the process of ETL (Extraction, Transform and Load), and regulates the traffic. Once the ETL process is over, the Warehouse Flow Regulator (WFlowR) is responsible to orchestrate data into the Data Warehouse, based on the current workload[VS08].

3.1.2 Architecture of Obali

Obali proposed an RTDW that assures near real-time data from different data sources. The components of this architecture are Web Service Client, Web Service Provider, Metadata, ETL, Real Time Partition, Data Warehouse and Real-Time Data Integration [ODEG13]. The full architecture can be seen in Fig. 3.2. The Web Service Client is responsible for getting data changes from raw

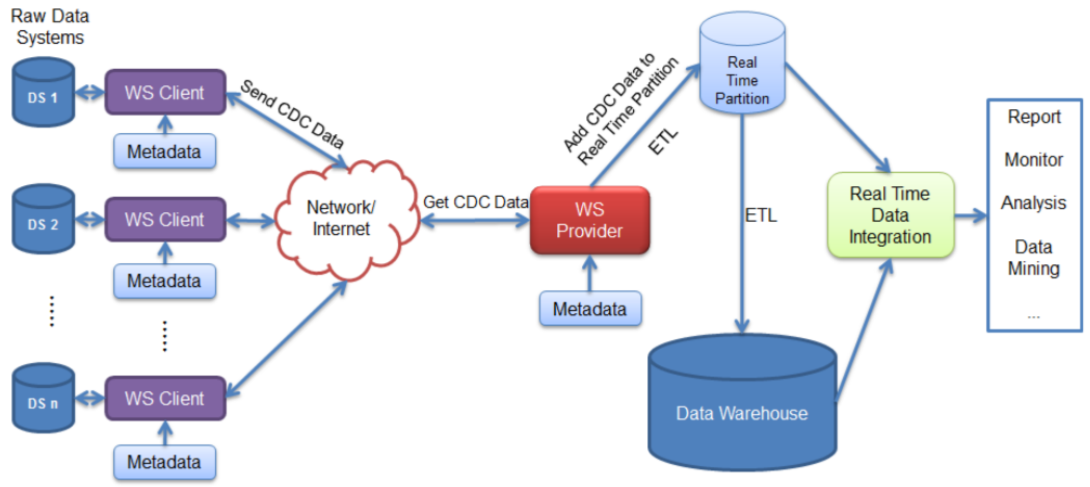

Figure 3.2: Web Services based Real Time Data Warehouse Arquitecture.
Extracted from [ODEG13]

data in data sources and sends it to the Web Service Provider and adds it to the Real Time Partition. This component gets Data Transfer Object which is sent by RTDW Web Service Client and decomposed into two parts: data and metadata. RTDW Web Service uses metadata to generate SQL via SQL-Generator for inserting data to RTDW log tables then executes this generated SQL on RTDW database and inserts data[ODEG13]

The component Real-Time Data Integration is used to integrate data both in Real Time Partition and the Data Warehouse, thus, if a query only needs historical data, then this component sends the query to Data Warehouse; if query wants both historical and instant data, then this component rewrites the query to get and integrate data[ODEG13][BNG17].

3.1.3 A Generic Architecture, by Cardinale

A Big Data analytical approach architecture, proposed by Cardinale[CGR18], is composed of four main components:

- **Data Sources** referring to heterogeneous sources of data.
- **Data Ingestion Module** are responsible for moving source data into a system where it can be stored and analyzed. Data can be ingested asynchronously, continuously, in real-time, batched or both, real-time and batched, referred as *Lambda Architecture*.
- **Analytic Processing Module** can have three different layers:
  - The **Data Storage Layer**, describing where the data is going to be stored. These can be persisted locally, placed in the cloud or cached in memory, such as a file system (Hadoop[1]), a NoSQL repository, a parallel RDBMS or a Graph Database.

[1]More information about Hadoop at https://hadoop.apache.org/docs/r1.0.4/

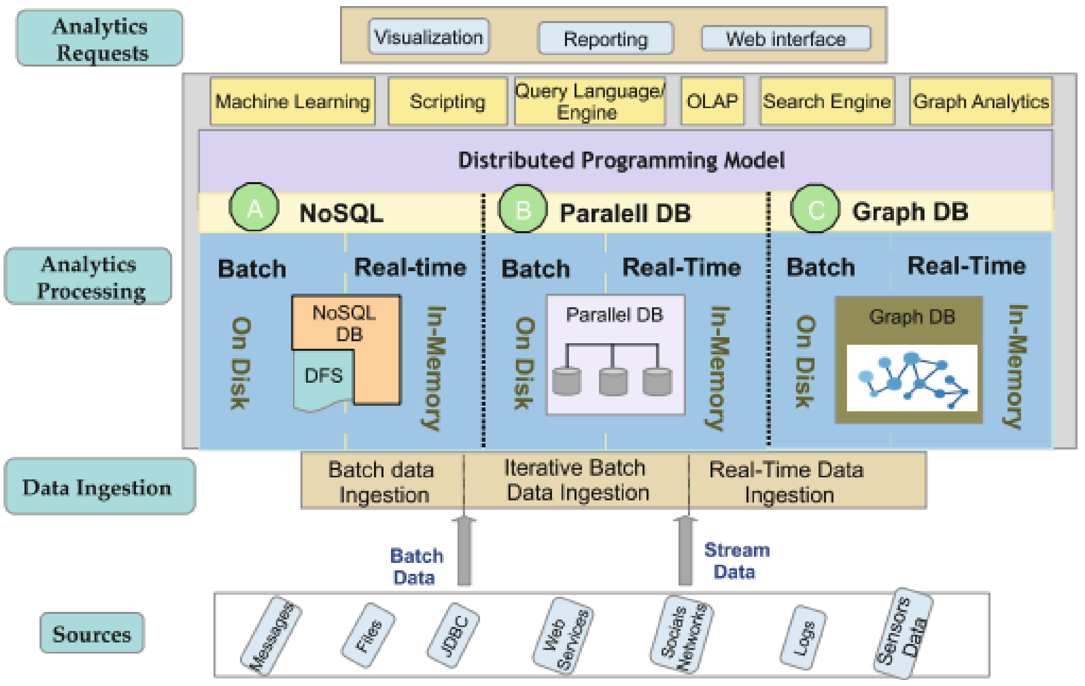

Figure 3.3: Generic architecture for a Big Data Analytical approach. Extractred from [CGR18]

  – The **Programming Model Layer**, where programming models are set (for example, MapReduce[2]) and corresponding frameworks (Hadoop, Storm[3], Spark[4], Kafka, etc.). The choice of the framework will depend on the priorities regarding latency, throughput, along with the type of analytics solution, or other needs.

  – The **Data Analysis Layer**. containing one or more analytics tools (ML, Query Language/Engine, OLAP, Search Engine, Graph analytics).

- **Analytical Requests**, related to Visualization, Reporting, BI, Dashboards, etc.

The architecture defined above can be visualized in fig. 3.3.

3.2 Real Time Big Data Systems

Nowadays, companies struggle to remain valuable and important in the market, with an increasing number of competitors. The market has been overrun by this idea of data-driven business models. With the increasing availability of Big Data technologies, the more information a company has the more competitive advantage it holds [Rag18].

[2]More Information about MapReduce at: https://hadoop.apache.org/docs/r1.0.4/mapred$_tutorial.html$
[3]Learn more about Storm at: http://storm.apache.org/releases/current/index.html
[4]More information on Spark can be found here: https://spark.apache.org/docs/latest/

However, to take advantage of Big Data, it is important to build a robust data management architecture strong enough to support and deliver on-demand, real-time business insights to the company analysts [GCLZ18].

According to Cuzzocrea, data stored in these scenarios have four specific characteristics, being the first how to deal with*large-scale data*, the size and the distribution of data repositories, the second one being the scalability issues, which refers to the capacity if the systems to run on a large and growing-in-size scale, the third one is the necessity of more advanced Extract, Transformation and Loading (ETL) approaches to somehow present more structured information and the last one the need to design and develop more advanced and easy analytics repositories to extract useful knowledge from Big Data [CSU13].

Some of the Big Data tools first created were tools such as Hadoop and MapReduce. The Hadoop Distributed File System, HDFS, is the core element of Hadoop, it is highly fault-tolerant and is designed more for batch processing, providing high throughput access to applications with large data sets, rather than for low latency of data access [SM13]. MapReduce is a framework for processing large data sets in parallel on a cluster, data has loaded a part of the ETL process, and the second phase consists in a *mapping* data to worker nodes and *reducing*, and giving a final output. It is a cloud-based, easy scalable framework. Cloud-based computing provides a number of benefits for implementing Big Data since are easily accessible and have low up-front costs with high capacity for scaling [SM13]. Although these technologies represent a major step towards Big Data, it does not perform well for real-time processing [RSA+18], that requires low latency messaging and event processing, as close to real-time as possible [PLA18], where popular technologies choices are Apache Kafka, Spark Streaming, Apache Flume, Amazon Kineses, and so on.

Big companies like Netflix, Twitter, Facebook, and LinkedIn have made huge improvements in delivering a real-time system, capable of handling Big Data analytics [PP15].

For, Twitter, an application with millions of users and millions of interactions per second, it is of great importance to have a Big Data platform capable of handling millions of data coming in real-time. In 2013, the company had already observed that the typical Hadoop approach would never meet the needs of the latency requirements of the application [MDL+13].

3.3 ETL in a Real Time Environment

In a traditional Data Warehouse, data integration, usually, occurs with the system offline, at dead hours or overnight, and a very large and heterogeneous set of data goes through the whole process of being extracted, transformed and loaded, known as ETL processing [FMF13][SAS+17].

However, in recent past, many industries are starting to require the most current information possible in their data warehouses and up-to-date information for analysis and reporting[FMF13].

The key features to achieve Near Real-Time ETL are High Availability, Low Latency of intervals between transactions and Horizontal Scalability for performance improvement[SAS+17]:

- **High Availability** is important since, in a near real-time environment, data is key to perform decisions and should always be updated and made available. Therefore, no data should be lost and to ensure this, distribution and replication are important factors to take into account, so, when data is lost, it can always be recovered. Unlike batch data which are distributed periodically, change of fresh data is less frequent and the same batch data can be used repeatedly.

- **Low Latency** means delivering fresh data as fast as possible, and it is the time between data arrival and data made available. Hence, smaller the time, smaller the latency and faster the access to new and fresh data. Unlike batch data which are distributed periodically, change of fresh data is less frequent and the same batch data can be used repeatedly. Traditional ETL was designed to accommodate batch processing where data refreshment can be done during off-peak hours of which operational activities can be temporarily suspended: could not produce an accurate result for near real-time analysis where stream data flow continuously.

- **Horizontal Scalability** is an important factor when discussing the improvement of performance and scalability. It mitigates the risk of interruption of seamless data flow, by adding independent servers to a cluster of servers, handling different tasks.

A few near real-time ETL frameworks were studied in order to understand how traditional ETL could be improved to handle real-time streaming.

3.3.1 Big Data ETL

Big ETL is a parallel/distributed ETL approach for Big Data. This framework consists of MapReduce ETL techniques, where each ETL process is distributed on a cluster of computers[BBA15].

Data coming from data sources is partitioned and an ETL process is run, in parallel, for each one. The novelty of this framework is the *Vertical Distribution of Functionalities, VDF*, which consists of the distribution of ETL tasks, of one particular ETL process. In other words, for an ETL instance, such as creating the SK, to insert in two tuples. being CUSTOMER and PRODUCT, this instance will be running in one cluster, but the ETL task of creating the SK for CUSTOMER and the SK for PRODUCT will be running in different nodes, in parallel, as can be seen in Fig. 3.4. The framework showed improvements of, roughly, five minutes saved, when testing for more than two billions of tuples and with five to six tasks running in parallel.

3.3.2 Event-Driven ETL

The first event-driven framework studied was **NMStream**, a scalable event-driven ETL framework for processing heterogenous streaming data [XLWW18].

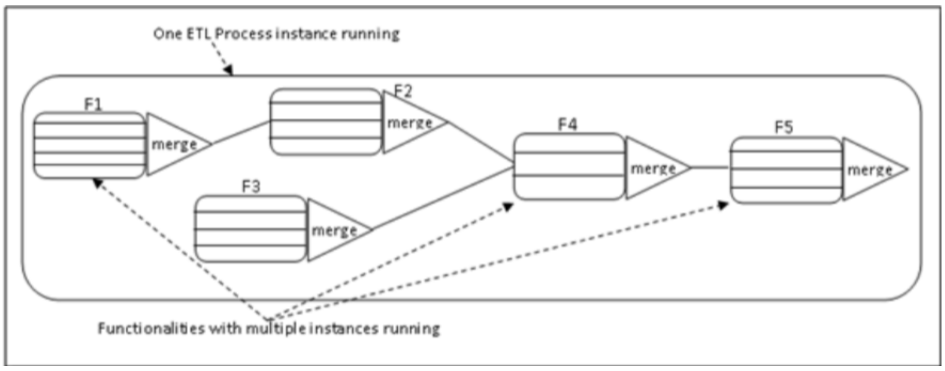
One ETL Process instance running
F1
merge
F2
merge
F3
merge
F4
merge
F5
merge
Functionalities with multiple instances running

Figure 3.4: Vertical Distribution approach. One step from the ETL process, with multiple nodes for 5 different contexts.
Extracted from [BBA15]

This application follows an event-driven architecture and addresses heterogeneous and distributed data by integrating Apache Flume[5], responsible for handling ETL processes, and Cassandra DB[6]. And is described as an effective manner to feed traditional and real-time data warehouses or data analytical tools.

Apache Flume was built as a distributed and reliable service, robust and fault tolerant, and responsible for gathering, aggregating and moving various types of streaming data into HDFS. Agents form Flume workflow and messages between two agents are called *events*, and the agent responsible for deliver events is called *Channel* and for filtering events uses *Inceptors* agents.

Cassandra is a NoSQL fully distributed and highly scalable database, and clusters can run on different commodity servers and across multiple data centers.

Each agent can be linked with one or more interceptors, which are used to transform or filter an event, and can form a data flow and after being filters and transformed can be written in the database, instantly.

Lastly, the last framework studied was **Striim**, an end-to-end distributed platform that enables rapid development and deployment of streaming platforms [PKS$^+$18]. It holds built-in adapters to extract and load data in real-time from legacy, can transform events using a SQL-based extension inject custom logic and has built-in data validation to make sure data arrives at the destination.

Striim has a distributed scale-out architecture running on several clusters, horizontally for scalability, and can, continuously, ingest massive amounts of data and then process it using a transformation engine with high throughput and low-latency. ETL tool runs in a scalable and fault-tolerance cluster of nodes, extracting data from sources and applying transformations necessary and results are loaded, or streamed, into the targets. The overall architecture and organization of the platform can be seen in Fig. 3.5.

To extract real-time transactional data from databases it uses the CDC technique, capturing transactional changes and sends data in the form of an event stream, using Apache Kafka as a

[5]More information about Apache Flume at https://flume.apache.org/documentation.html
[6]More information about Cassandra DB at: http://cassandra.apache.org/doc/latest/

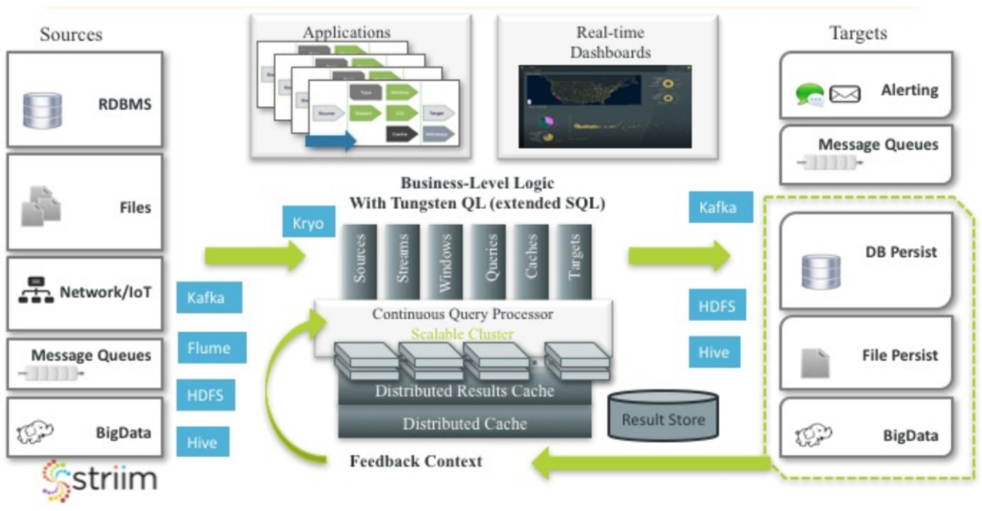

Figure 3.5: Striim Architecture.
Extracted from [PKS+18]

messaging system. However, it uses a lot of dependencies from internally developed languages and is not open-sourced.

3.4 Real Time Distributed Messaging System at LinkedIn

Taking advantage of Kafka distributed, low-latency and high throughput qualities, a recent trend has been emerging where activity data is used in the form of a log system or event messages to capture user and server activity. This data is the core of many recommendation services, as well as fulfilling its traditional role in analytics in reporting, that is becoming more real-time demanding.

Being Kafka a framework developed by LinkedIn, a social network with millions of users, and billions of interactions per day, a deep study was developed in how this company handled the number of events produced in the system and how fast could they ingest this data for reporting and analytics.

The company did not always use this distributed publish/subscribe messaging system. Several years ago, it had two separate systems, a batch-oriented system, designed purely to load data into the data warehouse, and a second system, handling server metrics and logging but only for the monitoring system. These systems were point-to-point data pipelines, with no integration between. The system was too complex and had to high latency, not responding to the business needs [GKK12].

It started a quest for integrating a real-time infrastructure, but no solution seemed to target the requirements for high-volume scale-out and low latency deployment. Kafka was then built.

LinkedIn handles more than 10 billion messages writes per day, resulting, on average, 172,000 messages per second. Holding dozens of subscribing systems and delivering more than 55 billion messages to these consumers, each day. There are a total of 40 real-time consumers that consume one or more topics. The average end-to-end latency through the pipeline is 10 seconds, and in the worst case, takes 32 seconds [GKK12].

Kafka uses three techniques to improve efficiency:

- Partitioning up data, so producers, broker, and consumers are all handled by clusters of machines, that can be easily scaled as load increases.
- Batching messages, sending larger chunks at once.
- Shrinking data to make the total data sent smaller in size, without data loss.

Batching fundamental technique is to induce a small amount of latency to a small group of messages to improve throughput. This latency is a precise timeout and message count, guaranteeing that a batch never has more than a predefined number of messages, and data is never held for more than then the time desired by the user. When tested, showed high efficiency for high-volume and low-volume Kafka Topics, by a factor of 3.2 for producers, a 51.7 factor, for brokers and 20.1 factors for consumers, considering throughput [GKK12]. The algorithm used is similar to Nagle's algorithm, used in TCP implementation. The pseudocode can be seen below, in table 3.1.

Table 3.1: Pseudo-code for Nagle's Algorithm

```
    IF there is new data to send
    IF the window size >= MAXIMUM_SEGMENT_SIZE and available data is >=
        MAXIMUM_SEGMENT_SIZE
      send complete MAXIMUM_SEGMENT_SIZE segment now
    ELSE
      IF there is unconfirmed data still in the pipe
        enqueue data in the buffer until an acknowledge is received
      ELSE
        send data immediately
      END IF
    END IF
  END if
```

Shrinking Data helped LinkedIn to handle with bottlenecks related to bandwidth between data center facilities. There are two approaches for shrinking data:

- Exploiting explicit structure in message serialization
- Exploiting implicit structure using compression

Kafka allows for batch compression of multiple messages into a single intricate message. The compression algorithm used by LinkedIn is GZIP[7] and results in a compression ratio of 3 times and a 30% improvement in throughput [GKK12].

[7]More information about GZIP at: https://www.gnu.org/software/gzip/manual/gzip.html

3.5 Conclusions

Due to the nature of this thesis, where it shall be improved the data ingestion for higher throughput and lower latency, providing a scalable solution capable of handling Big Data, there is no doubt the paper from LinkedIn, in how they build there real-time activity data pipeline, brought an interesting vision in how much advantages could be taken away from Kafka especially the micro-batching algorithm, due to the improvement factor it showed for amount of data consumed, without causing too much latency.

Chapter 4

Problem Definition

In this chapter, it shall be presented the context and the definition of the problem studied during the period of the development of the dissertation.

The dissertation was developed on a Portuguese logistics company, HUUB, therefore a small description of the company technologies and architecture will be presented, along with how the company positions itself on the state of the art and pinpoint some architectural problems studied during the first weeks of the internship that helped to define the problem.

4.1 SPOKE Architecture

As mentioned before, HUUB is currently on the process of transitioning from a monolithic system to microservices. At the moment, the platform presents the architecture described in Fig. 4.1.

Most business processes still depend a lot from the legacy system, nonetheless, there are three independent services deployed. The services correspond to Quality Control Service, Tracking Service, and Delivery Service.

Quality Control Service was the first independent deployed service. Registers all entries from the quality control team in warehouses about quality and state of the products.

Tracking Service is related to the tracking experience launched by the company. This service allows clients to trace orders from purchase to delivery. It displays information, such as the expected date of arrival, and sent out email notifications.

Delivery Service carries a lot of important information useful for reporting and analytic teams. After an order is confirmed, there are some pending requirements to allow its process, delivery will fulfill all those requirements and calculates when the delivery must be processed.

An important aspect when considering the transition from a monolithic architecture to microservices is how would these new services communicate. The approach followed was to have an event-driven lightweight communication mechanism, HUUB chose the message broker Apache Kafka.

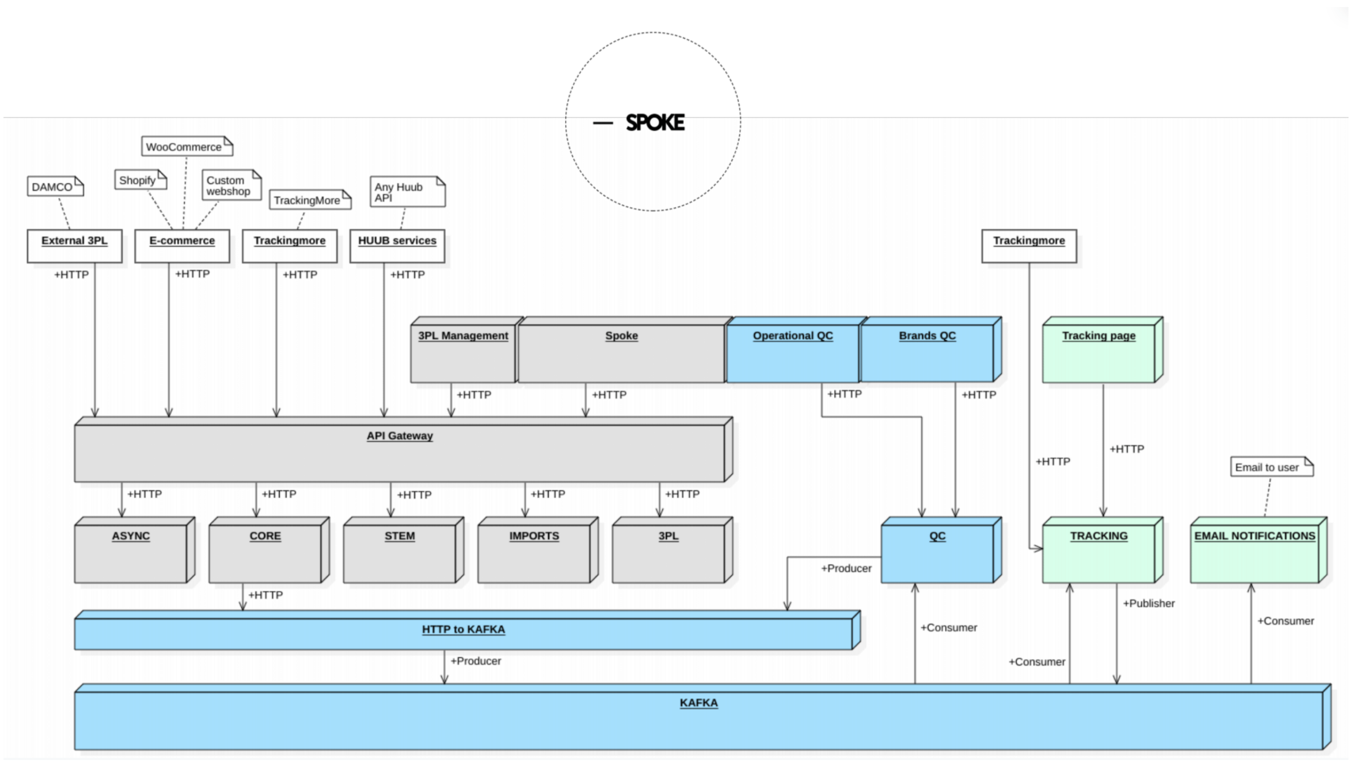

Figure 4.1: Spoke Architecture: Legacy(in grey), Microservices(blue and green) and Kafka Message Broker(in blue)

There are 4 Kafka Brokers, as seen in fig. 4.2 distributed in two different clusters, one in the Staging Environment[1], and three brokers in the Production Environment. For every type of event, there is one topic where messages are published, and, there are, currently, 40 topics in production, allowing communication between all three services built up to now.

Each Topic is divided, by default, in 16 Partitions, and each message carries a header. Messages that share the same header are published in the same partition, guaranteeing consistency. Additionally, every topic as, by default, a replication factor of three, having one leader and two non-leader replicas, to guarantee consistency and partition tolerance.

Nonetheless, HUUB still has a long way before completely discard the monolith, meaning there is still a high dependency on business capabilities from legacy, and only a small set of features are independently deployed as a small set of services.

Legacy was built around out-dated and not very scalable technologies. Front-end uses HTML and CSS, PostgreSQL is used as the database and all back-end was build using Node.js, Web2Py, PHP and Laravel. The new architecture uses Angular for front-end, Python and Django for back-end and PostgreSQL as database.

[1]Staging Environment in HUUB is a development stage similar to the Production Environment, used for software testing. Ensuring quality under a production-like environment before deployment.

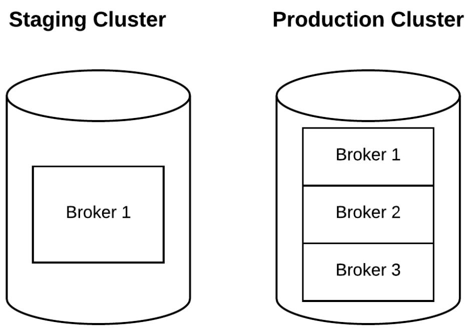

Figure 4.2: Kafka Clusters at HUUB

4.2 Data Platform Architecture

Being Data and AI Driven, the company had to design a solution to ingest all data in a way that could be easily accessible and readable.

The current architecture of data capturing, at HUUB, can be separated in two different categorizations: Iterative Batch from legacy data and Streaming data from Event Sourcing. The schema for each type of data ingestion are synthesized below, in fig 4.3 and 4.5, respectively.

The most valuable data comes from Spoke. Being the core of the company to handle the whole supply chain, from end-to-end, and improve clients business, HUUB uses Business Intelligence, to organize and structure data in a Data Warehouse, separating each business capability into a Data Mart. Despite the fact that data is already available on Spoke, traditional operational databases only represent the current state of business, while data warehouses are designed to give a historical, long-range view of data and provide analytics which represents more of a comprehensive view of company's history.

The DW used, by the company, is Google BigQuery[2], which is a column-based, structured data store on the cloud. It allows streaming 1000 rows of data per second, to the same table, being a great tool for Big Data.

Even though data extracted comes from Spoke, there are two types of sources. The vast majority of data handled still comes from the legacy system. As it is usual for monolithic systems, the application has the main DB, which can be called Spoke Main DB. This DB holds the current state of the application, being a non-persistent DB. Regardless, for every entry inserted, data is also added to a persistent DB, containing all history of every transaction in the application. This is the first process of ETL, and a more comprehensive description of this process will be described below.

[2]More information about Google BigQuery at: https://cloud.google.com/bigquery/docs/reference/

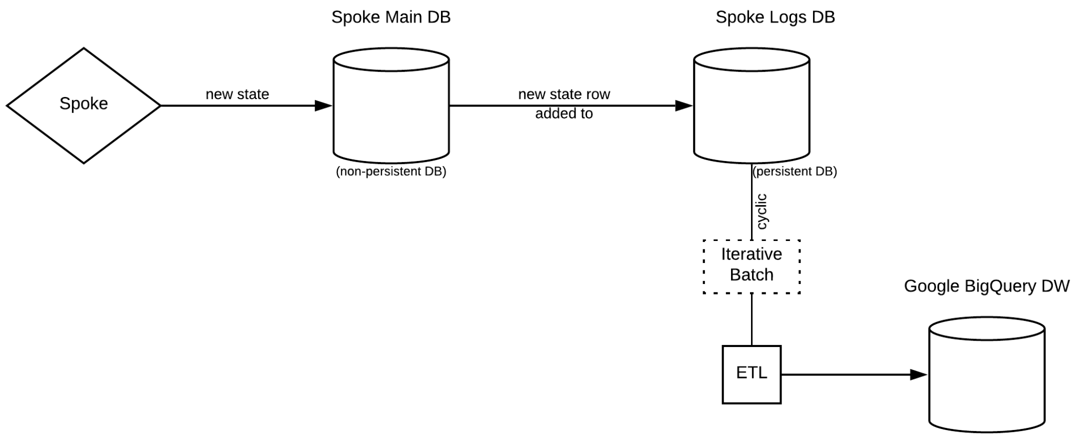

Figure 4.3: Architecture for legacy data, coming from iterative batch

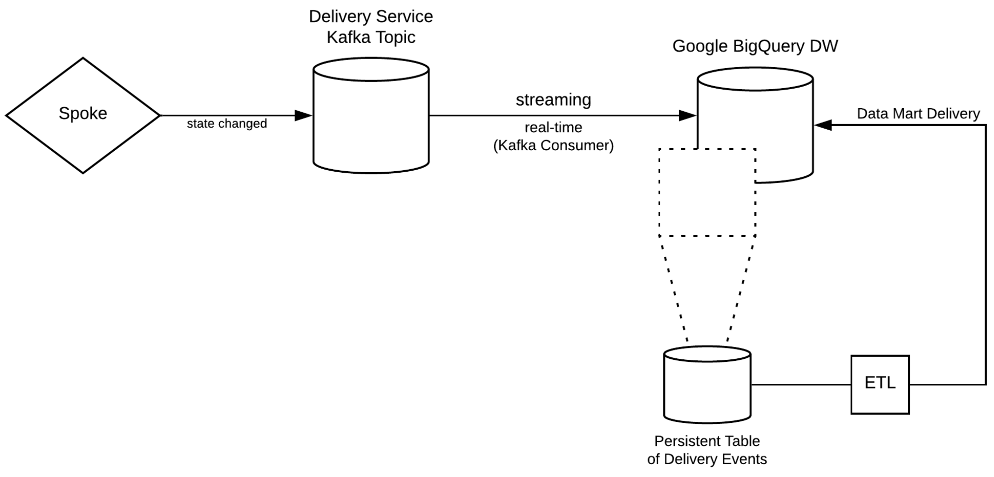

Figure 4.4: Architecture for event sourcing data, coming from streaming

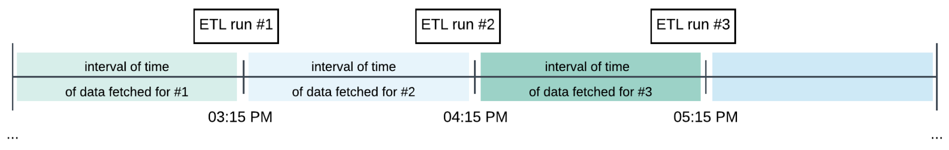
ETL run #1
ETL run #2
ETL run #3
interval of time of data fetched for #1
interval of time of data fetched for #2
interval of time of data fetched for #3
03:15 PM
04:15 PM
05:15 PM
...
...

Figure 4.5: ETL timeline: data collected by orchestrator to go through ETL

4.2.1 ETL Orchestrator

The process of extracting, transforming and loading data to a data warehouse, as discussed in chapter 2, is ETL.

BI team developed its own ETL framework, having a Main Orchestrator. The Orchestrator can handle different types of data sources and it is easily adaptable and scalable. It was developed using Python and runs on an automation server, Jenkins[3].

The orchestrator has 4 stages. The first stage queries a configuration table, that has all table names that will go through the process of ETL. It carries all information needed, such as source table name, destination table name, if it is run weekly, daily, hourly or sequentially, and references for SELECT, UPDATE and AUDIT SQL files, for each different source of data.

The orchestrator ran every hour, to treat all data from Spoke. It started by querying data coming in batches from Spoke persistent DB and adding it to the respective staging table (e.g. *pre_tablename*), according to the business capability it represents. However, to guarantee data consistency, when querying data to add to the staging area, it will only fetch data that was created in an interval between the current time, of the run of the orchestrator, and an interval of time defined. In this case, is of sixty minutes.

The third step consists of querying data from staging to dimensions in the star schema, and, at last, into facts.

The 4 stages of the ETL take around fifteen minutes, which means overall as the latency of one hour and fifteen minutes, on average.

Regarding data ingestion of data coming from microservices systems, only a few moves had been made then, but a more profound explanation will be portrayed in the next section.

4.2.2 Event-Driven ETL

Due to the high relevance of Delivery Service events, the company started to make progress in how could they take advantage of event sourcing to populate the Data Warehouse of the company as near to real-time as possible.

[3]More information about Jenkins can be found at: https://jenkins.io/doc/

Data from new independent services is slightly different, each service has a topic, a pipeline where messages produced are stored, for as long as the user wants. Each message holds information about a record that was created, if it did not exist, or suffered an update. As read in the background chapter in 2.2.1, an advantage of using Kafka is the fact any system, can consume events produced in real-time, being another microservice, or any other independent system, such as an ETL framework.

In other words, data can be extracted and used to reporting way faster than traditional transactional databases, given the fact in a traditional database data must be queried into a batch, and it consumes a lot of resources.

To consume messages from a topic, it only has to be connected to a Kafka consumer to it, and events are consumed as soon as made available. This was implemented using a Kafka library, Confluent Kafka Consumer[4], and was automatically built and ran using Docker Compose[5], which handles and restarts the Consumer in case it goes down.

Each event consumed is serialized to a JSON format, containing all information about a specific Delivery. A more thorough description of how information is organized will be displayed in the Implementation section.

4.3 Identified Problems

Studying the process as-is of HUUB allowed to understand what were a few of the problems that can be encountered when transitioning from a traditional Data Warehouse, dependant of batch processing and rudimentary ETL tools.

Overall, the architecture that HUUB is trying to achieve proves to follow recent trends in the state of the art, as seen during the literature review, but only mentioned by big social networks or streaming media services companies. However, the latency and throughput HUUB achieves in event sourcing for data ingestion and reporting, does not fit the current needs if the company and it does not prove to fulfill future business needs.

As seen during the Literature Review, at section 3.3, *Latency* is the time it takes for data to be consumed and made available since it has been produced. Therefore, the minimum latency, in this proof of concept, is the minimum amount of time it takes for data produced at Spoke to be Extracted, Transformed and Loaded into the DW, and made available for reporting. Whilst, the maximum latency is the maximum amount of time HUUB desires the process mentioned above to last.

The first problem identified was the fact HUUB was using a simple, Kafka Consumer, where only one consumer was ingesting events from the entire Delivery Topic, and only *polling* one event at a time.

[4]More information about Confluent Kafka Consumer can be found at: https://docs.confluent.io/5.0.0/clients/confluent-kafka-python/index.htmlconsumer

[5]More information about Docker Compose can be found at: https://docs.docker.com/compose/

For example, to build the Data Mart for Delivery, it was necessary to consume events from Kafka, as seen above. However, the solution was not optimized and went against the vision of scalability and Big Data the company was trying to set to itself. This conclusion was made by making a few calculations, at the beginning of the dissertation. By only having one consumer *polling* one event at a time and sending it to the DW. Polling an event and send it to the DW took approximately 0.755 milliseconds, meaning to consume 1 million of events, which is not even approximately to Big Data, would take almost 8 days.

Secondly, although these events were being consumed near to real-time (for the number of events being produced at the time, which was, on average, 10.000 events per day), to insert data into the Star Schema, it would have to wait and carry the same latency as data coming from legacy, due to the fact it holds dependencies on instances that should be initialized before the addition of a delivery.

4.4 Proposed Solution

The goal of the company is to provide the best logistics solution for their clients. Being so dependent from data and investing in the new approach of services and event sourcing, the problem to tackle became obvious, and a better solution for ingesting events, at a higher throughput and lower latency had to be defined.

Kafka holds a lot of advantages, some of them, were not being used by the company approach for faster data ingestion. To this matter, the solutions proposed for the improvement of this technology was:

- First, take advantage of some features from Kafka Confluent library and change the consumer class, from fetching one event at a time, and return a list of events already available in the partition.
- Another step was to take advantage of multi-partitioning and create multiple consumers and see how could this affect the throughput and latency of events ingestion.
- To finish, in order to handle Big Data, implement a Batching Algorithm, with a maximum latency, creating small batches of events in order to send more events at a time for the DW, without compromising latency.

When continuing to study the process at HUUB, as mentioned before, one of the biggest problems was to handle legacy data and event sourcing data. Such as the coherence and dependency of context. To solve this problem, not only for data legacy but future event sourcing data coming every second, an ID reservation had to be done to every context that an event held from other legacy entity.

Chapter 5

Implementation

After studying all processes at HUUB related to event sourcing with Kafka and ETL orchestration, it was easy to see how poorly the solution was optimized to handle with high throughput real-time data ingestion, coming from event sourcing.

There were two problems to tackle. First, how could the current real-time Kafka Consumer be scaled in a way it could handle millions of events per day, i.e., increasing throughput without neglecting latency. Secondly, due to the fact, HUUB still depends much on legacy data, how events could be added to the data warehouse despite these dependencies.

5.1 Setup

To run Kafka Server it is used a Docker Compose[1], defining the database, Zookeeper, Cassandra DB, and other specifications. For the thesis development, it was created a Kafka server, hosted locally in the machine. When Docker is *up*, it is possible to list all topics that exist locally in the machine, using the following command shell, in table :list$_k afka_t opics$.

Table 5.1: List all Kafka Topics

```
        # kafka-topics --zookeeper my_kafka_zookeeper:[NUM_PORT] --list
```

Afterward, it was created a topic where events produced could be stored, without compromising flows in the company. The default topic is created with only one partition and a replication factor of one. The code necessary to create a topic is listed in table 5.2.

[1]More about Docker Compose at: https://docs.docker.com/compose/overview/

Table 5.2: Create Kafka Topic

```
        # kafka-topics --create --topic my_topic --partitions 1 --replication-
            factor 1 --if-not-exists --zookeeper my_kafka_zookeeper:[NUM_PORT]
```

Partitions can be added to a topic and this can be listed using the commands registered in table 5.3.

Table 5.3: Alter number of partitions in a topic and list all partitions

```
        # kafka-topics --alter --zookeeper my_kafka_zookeeper:[NUM_PORT] --topic
            my_topic --partitions [NUM_PARTITIONS]
        # kafka-topics --describe --topic my_topic --zookeeper my_kafka_zookeeper:[
            NUM_PORT]
```

5.2 Building Kafka Producer

This implementation could not be performed at the Production level, and, even though Delivery Service was already up and used as an experiment to start building HUUB's real-time activity data pipeline, it did not produce the necessary amounts of data to test a Big Data approach.

To overcome this problem, it was designed and implemented, locally, a Producer of *dummy events*, similar to Delivery. This was adapted from previously created producers built in the system, using ConfluentKafkaProducer[2]. The code used to create this producer is available in appendix F.

To create *dummy* delivery events it was built a python script, that initialized the producer class and continuously creates new events. The code for this implementation can be seen at appendix **??**.

5.3 Improving Kafka Consumer Class

The consumer built for consuming events from Delivery, to the Data Warehouse was, as mentioned before, poorly optimized. This first consumer was built using the same library from *Confluent Kafka Consumer*. This consumer used the method *poll* which returns one message at a time. It also used the method *subscribe*, which told the consumer the list of topics to subscribe to.

To take advantage of Kafka partitioning, it was built a new Consumer class, that inherited most of the methods from the superclass, but with a few modifications:

- When defining the attributes for the class, the consumer would not be *subscribed* to a list of topics, but instead, it would be *assigned* to a list with one or more partitions, for a given

[2]ConfluentKafkaProducer

topic. This was implemented using the method assign, with the parameter *LIST_PARTITIONS*, telling which partitions shall the consumer connect too.

- Secondly, after a consumer is initialized, instead of using *poll*, which tells the consumer to only fetch one event, it calls the method *consume*, where it passes the maximum number of messages it fetches, returning a list of messages, or empty in case of timeout.

- This last change, meant a different return type, so a few alterations had to be done to the function of deserializing, previously done.

The code for this part of the implementation can be seen in appendix A.

5.4 Consumer Parallelization through Multiprocessing

In the beginning, it was thought that the best way for consuming as many events as possible from a topic, it was by initializing as many consumers as the number of partitions that existed. However, as seen in the State of the Art, especially in the paper from LinkedIn[GKK12], despite having a lot of topics and consuming millions of messages, it only holds a few dozens of consumers.

For each consumer initialized, more resources are consumed. Therefore, the approach designed was to initialized consumers depending on the number given by the user. Meaning, the user can monitor flows and create less or more consumers depending on the needs of the system.

To automate, using *chron* jobs, and use some custom commands developed, previously, in the old Kafka Consumer, it is used Django Custom Management Commands. The code is available in appendix B. The main *class Command* instantiates all configurations necessary to create Kafka Consumers and communicate using a signal dispatcher that helps to communicate between decoupled operations.

5.4.1 Distributed/Parallel Consumers

When a consumer is created, through the new class developed, it passes a list of partitions. The list of partitions is defined depending on the number of partitions a topic has and how many consumers the user wishes to initiate. So the list of partitions per consumer is calculated using the following expression:

$$PARTITIONS\_PER\_CONSUMER = CEILING(\frac{\varepsilon PARTITIONS}{\varepsilon CONSUMERS})$$

Partitions are identified using only the partition number, therefore, after calculating the number of partitions, it is defined the list with the partitions that each consumer will be connected to. Which can be observed at table 5.4.

Table 5.4: Function responsible for creating the list of partitions to be consumed for each Kafka Consumer

```python
    def get_partitions_by_consumer(list_partitions, PARTITIONS_PER_CONSUMER):
        for i in range(0, len(list_partitions), n):
            yield list_partitions[i: i + n]
```

If a there are 8 partitions and only 3 consumers, the final list would be:

list_partitions = [[Partition 0, Partition 1, Partition 2], [Partition 3, Partition 4, Partition 5], [Partition 6, Partition 7]]

After the list of partitions is set, consumers must be initiated. Since Kafka only provides total order over records within a partition, consumers can be run concurrently, maximizing number of events read per time frame.

To run this process in parallel it was used the library, from Python, Multiprocessing, more specifically, the Process package. This allowed launching parallel consumers, taking into consideration the number of consumers and the partitions it had to read from. The code for the Multiprocessing is described in table 5.5

Table 5.5: Code for Parallel Kafka Consumers

```python
    for i in range(0, len(list_partitions)):
        p = Process(target=consumer_init, args=(self, list_partitions[i], i,
            options,))
        logger.info("LAUNCHING {} CONSUMER(s) FOR PARTITION(s): {}".format(
            NUM_CONSUMERS, list_partitions[i]))
        processes.append(p)
        p.start()

    for process in processes:
        process.join()
```

When a process is started, the first step is to instantiate the consumer, using the Confluent Kafka Consumer class designed, previously. When the consumer is created and partitions are assigned, it is possible to start reading from the topic. This is done by calling consumer method *consume()*, with the parameters *NUM_MAX_ROWS* and *POLL_TIMEOUT*.

5.5 Micro Batch Implementation

As discussed before, the Kafka consumer can return a list of messages, instead of consuming one event at a time. Also, Google BigQuery, the DW used by the company, can stream 1,000 rows per second. One message equals to one row. Considering these features from both technologies, a micro batch, with low latency, inspired by Nagle's Algorithm [MSMV07], can be designed in a way of improving throughput and make HUUB capable of consuming hundreds of millions of events per day.

The restrictions applied to this algorithm were:

- For each loop, the maximum latency is, currently, 1.5 seconds, but can easily be changed depending on the number of events being produced.

- The loop does not end while the maximum latency has been reached or the list of events is less than 100000 rows since, for each stream, BigQuery can only handle 100000 events.

When one loop ends, events are denormalized and sent to BigQuery, by using the API library streaming method. The code for the micro-batch implementation and sending data to BigQuery is available in appendix D.

5.6 ETL Improvement

As discussed before, data coming from event sourcing can cause problems with data coming from legacy (and in the future, events coming from other entity). If new events arrive every second, some instances present in those events will be out of context, due to dependencies coming from legacy.

To solve this problem, there was a study of how data goes through the orchestrator. When a new row is added to a table in the star schema, it holds an ID, created in Spoke. This ID, when added to the dimension tables is inserted under the column DD_COD. However, And an incremental SK (Surrogate Key) is also created, in order to keep a certain order in the DW.

Delivery holds data from legacy. For example, Sales Order, Shipping Address, Customer and Customer Address, and others. This data was previously created in Spoke, but since is legacy it suffers a bigger latency in ETL. To solve this problem, when a Delivery is added from staging to a dimension table, a verification is made. If a DD_COD, from dependency from legacy, does not exist in the dimension table correspondent, an ID reservation is made, where the DD_COD from the entity is added, along with an SK. Later, when ETL for legacy data runs, that row is updated and the rest of the data is filled in.

Being Delivery the only service producing events ready to be consumed and holding important information, all implementation was done to this entity. However, this solution is capable of being reproduced in all contexts.

To the insert file on dimension delivery, it was added the piece of SQL code where it is verified if all dependencies are already instantiated or not, as seen below in table 5.6.

Table 5.6: Instantiate dependencies from legacy in INSERT_DIM_DELIVERY

```sql
    INSERT `star_schema.dim_sales_order` (
    SK_SALES_ORDER
  ,COD_SALES_ORDER
a)
SELECT
    source.SK_SALES_ORDER
  ,source.COD_SALES_ORDER
FROM (
    SELECT
        ROW_NUMBER() OVER (PARTITION BY innerQ.serial_generator ORDER BY innerQ.
            COD_SALES_ORDER) + (SELECT CASE WHEN MAX(SK_SALES_ORDER) IS NOT NULL
            THEN MAX(SK_SALES_ORDER) ELSE 0 END AS max_sales_order FROM `
            star_schema.dim_sales_order`) AS SK_SALES_ORDER
      ,innerQ.COD_SALES_ORDER
      FROM (
      SELECT
        1 AS serial_generator
        ,so.delivery_order_id AS COD_SALES_ORDER
    FROM (SELECT distinct delivery_order_id FROM `staging.pre_delivery` AS d) so
      WHERE so.delivery_order_id NOT IN (SELECT DISTINCT COD_SALES_ORDER FROM `
          star_schema.dim_sales_order`)
    ORDER BY so.delivery_order_id
    ) AS innerQ
) source
ORDER BY source.SK_SALES_ORDER
```

This only for one dependency, in appendix E it is possible to see the whole INSERT file, as well a few other small changes made to the orchestrator to handle these new features.

5.7 Conclusions

All the features added were tested in the Staging Environment of the company, showing great improvement and responding well. The goals were all answered, the new framework of Kafka Consumer proved to run effectively and consuming a lot more events, much faster, without compromising ETL processing, due to the deep study to the company orchestrator and improvements made.

However, a few experiments were set in order to provide a better comparison and prove the goals achieved during the dissertation, in the next chapter.

Chapter 6

Experiments and Results

6.1 Experiments Setup

The experiments took place on a processor of 2,9 GHz Intel Core i5 machine, with 16 GB of RAM, in a macOS Mojave operating system.

The intents of the experiment were to compare the solution presented before and after the implementation of the subject of the dissertation. Being the whole goal of this project to lower the latency of data availability and throughput, the final objective was to set different cases of a number of events, number of partitions, number of consumers and different times of maximum latency in the batch algorithm, throughout different types of consumers.

The experiment has 3 different types of consumers:

- **Old Consumer:** the consumer applied in the company before the dissertation. It has only one partition and one consumer, and no batching algorithm.
  It also uses the method *poll* to consume events, which means it fetches one event at a time, from the Kafka pipeline, and since it has no batching sends one event at a time for the DW. This consumer is compared to a consumer, holding the same conditions, but using the method *subscribe* from Kafka Confluent library, opposite to *poll*, meaning it is retrieved more than one event at a time, sending a list of events to the DW, but it does not go through the batching algorithm.

- **Multi Consumer With No Batching Algorithm:** For this experiment, the number of partitions and consumers can vary, however, it does not hold any batching algorithm. This consumer allows to experiments with the basic features of Kafka, such as taking advantage of multi partitions and consumers. Holds two types for fetching of events, it compares multi partitioning and multi consumers with the methods *poll* and *subscribe*, retrieving and sending to the DW, one event or a list of events, respectively.

- **New Consumer:** Tests times for the new consumer proposed in this solution. Using multi partitioning and multiple consumers, using only the method *subscribe* from the Kafka Confluent library, as well as, the Batching Algorithm proposed, with different maximum latency times depending on how much events are being produced.

Each set of experiments will focus on each type of consumer and for two different amounts of data: 1.000, 10.000 and 100.000 events.

Each experiment, for each type of consumer, was run three times and the result is an average of the latency experienced.

6.2 Experiments for 1 Partition and 1 Consumer Latency

The first experiments to took place were for the case of only having one Kafka partition for each topic and only one consumer assigned to the topic. The goal was to record how much longer it would take the different consumers to ingest 1.000 and 100.000 events.

The experiment was divided into two:

– First, not using the Batching Algorithm developed during the implementation.
  The consumer was run using the original *poll* method, prior to the dissertation, fetching one event at a time.
  However, a second experiment was done, running the consumer using the *subscribe* Kafka method talked during implementation, added in the custom Kafka Consumer, where a list of events was fetched, depending on the events in the Kafka Partition.
  The latency for this experiments can be seen in the table 6.2.

– Secondly, using not only the custom Kafka Consumer Class, by using the method *subscribe*, as well as, the Batching Algorithm, for 3 different maximum latencies (MAX_LATENCY): 0.5 seconds, 1.0 seconds and 1.5 seconds.
  The results for this experiments can be seen in table 6.2. This values refer to the amount of time the MicroBatching Algorithm waits while consuming new events, stored in the same frame.

Table 6.1: Latency, in milliseconds, for 1 partition and 1 consumer, when producing 1.000, 10.000 and 100.000 events, using previously used Kafka Consumer class and customized Kafka Consumer class

NUM EVENTS	LATENCY w/ poll()	LATENCY w/ subscribe()
1.000	00:11:35,500	00:00:06,408
10.000	01:45:00,500	00:00:45,285
100.000	16:15:00,500	00:09:50,468

6.2.1 Multiple Partitions and Consumers

The second set of experiments was conducted by adding more partitions in the Kafka topic and by assigning more consumers. Similar to the experiences above, the experiments were

Table 6.2: Latency, in milliseconds, for 1 partition and 1 consumer, when producing 1.000, 10.000 and 100.000 events, using customized Kafka Consumer class and Batching Algorithm

NUM EVENTS	MAX LATENCY: 0.5 sec	MAX LATENCY: 1.0 sec	MAX LATENCY: 1.5 sec
1000	00:00:04,154	00:00:04,738	00:00:03,974
10.000	00:00:35,468	00:00:32,161	00:00:29,442
100.000	00:04:16,731	00:04:08,147	00:03:49,320

tested by seeing how much time it would take the consumers to ingest 1.000 and 100.000 events, for both cases, of using, and not using, the Batching Algorithm.

The results for the multiple consumers without using the Batching Algorithm are in table 6.2.1. The results for the multiple consumers using the Batch Algorithm are in table 6.2.1.

Table 6.3: Latency, in milliseconds, for multipartitioning and multiple consumers, when producing 1.000, 10.000 and 100.000 events, using customized Kafka Consumer class

NUM PARTITIONS	NUM CONSUMERS	NUM EVENTS	W/O BATCHING ALGTH
2	2	1.000	00:00:05,150
2	2	10.000	00:00:38,180
2	2	100.000	00:07:57,234
4	2	100.000	00:06:28,163
4	4	100.000	00:06:31,227

Table 6.4: Latency, in milliseconds, for multipartitioning and multiple consumers, when producing 1.000, 10.000 and 100.000 events, using customized Kafka Consumer class and Batching Algorithm

NUM PARTITIONS	NUM CONSUMERS	NUM EVENTS	MAX LATENCY: 0.5 sec	MAX LATENCY: 1.0 sec	MAX LATENCY: 1.5 sec
2	2	1.000	00:00:07,979	00:00:05,944	00:00:06,794
2	2	10.000	00:00:31,936	00:00:31,198	00:00:32,103
2	2	100.000	00:01:31,654	00:01:16,316	00:00:51,938
4	2	100.000	00:03:57,507	00:03:23,187	00:03:01,654
4	4	100.000	00:03:37,199	00:03:32,380	00:03:27,515

6.3 Conclusions

From the results exposed, it is possible to identify a few conclusions and improvements.

When using the*subscribe* method from Kafka Confluent class, it is possible to verify a drastic decrease in latency, with improvement around 100 times less time. After analyzing latencies between maximum latencies for the same quantity of events, or the same number of partitions and events, the difference is really small, but in a scenario of Big Data, and higher throughput, a few milliseconds, can make a difference.

Using the Batching Algorithm and the *subscribe* method, brings advantages, allowing for more throughput in less time. In some cases of multi-partitioning and multiple consumers is not so necessary. At least not when dealing with Big Data. Since the resources spent on the distribution system do not compensate in the end. However, when comparing latencies of the usage of the custom Kafka Consumer class without the Batching Algorithm, it is possible to see an increase in times. When ingesting 100.000 events, it is possible to see a decrease of 50% of latency when having 1 Partition and 1 Consumer, or even when having multiple partitions and consumers.

In conclusion, the time it takes data to be produced, consumed and made available for reporting, went from hours, as seen in table 6.2, to last only a few minutes, even when consuming 10 or 100 times more events.

Chapter 7

Conclusions

This chapter discusses the entire work developed during the dissertation, what were the goals to be achieved and the final motivation. Moreover, the difficulties of implementation, paths for improvement and the goals attained.

7.1 Difficulties

One of the biggest difficulties around this topic was how little literature there was to search. Microservices and Event Sourcing are subjects with few scientific contributions. Most of the material found on these subjects come from approaches explored and developed by enterprises. Additionally, the topic of Data Warehouses and ETL tools are, too, subjects of a more business character.

Another difficulty was having to learn such specific technologies so fast in order to, not only contribute with a novelty to the scientific community, but also create a solution that would bring the most value for HUUB.

7.2 Contributions

As mentioned before, being this a topic with hardly any documentation, this document does not only presents a real-time ingestion data platform, in the cloud but on top of that, how can a message broker be used to retrieve important information that can be used for report and analytics.

Although developing the solution provided in a company, restricted the technologies that were going to be used, such as Apache Kafka and BigQuetry. Those proved to be state of the art technologies capable of handling Big Data and good approaches to follow. Additionally, the Batching Algorithm can be used in fact in other message brokers who do not serve the same features as Kafka, increasing throughput without damaging latency and real-time data ingestion.

This work also provides explicitly the times of ingestion that can be used later for reference or benchmarking.

7.3 Future Work

In terms of future work, it would be interesting to analyze a Non-Relational Database, since these are better for large amounts of data, or even explore Data Lakes.

Moreover, it would be interesting to test a few more use cases and experiment with the algorithm to follow the scalability and resilience of it when entering an era of Big Data in the company.

Another possible future work could be to automate the number of partitions, consumers and maximum latency depending on the number of events being produced per second, adapting to the needs of the platform. As well as, a tool to monitor all topics and consumers assigned.

7.4 Conclusions

Altogether, this dissertation allowed to gain new knowledge about new technological trends, that are gaining every day more importance in the scientific community and in the market, due to the expansion of microservices and distributed systems. Lastly, the author believes the dissertation brings new value when considering new ways of improving business intelligence approaches to increase competitive advantage.

References

[BBA15] Mahfoud Bala, Omar Boussaid, and Zaia Alimazighi. Big-ETL: Extracting-Transforming-Loading Approach for Big Data. *International Conference on Parallel and Disrtibuted Processing Techniques and Applications*, 2015.

[BNG17] Senda Bouaziz, Ahlem Nabli, and Faiez Gargouri. From traditional data warehouse to real time data warehouse. In *Advances in Intelligent Systems and Computing*, 2017.

[CGR18] Yudith Cardinale, Sonia Guehis, and Marta Rukoz. Classifying big data analytic approaches: A generic architecture. In *Communications in Computer and Information Science*, 2018.

[CSU13] Alfredo Cuzzocrea, Domenico Saccà, and J.D. Jeffrey D Ullman. Big Data : A Research Agenda. *ACM International Conference Proceeding Series*, 2013.

[DDM14] Yuri Demchenko, Cees De Laat, and Peter Membrey. Defining architecture components of the Big Data Ecosystem. In *2014 International Conference on Collaboration Technologies and Systems, CTS 2014*, 2014.

[EMH$^+$17] Benjamin Erb, Dominik Meißner, Gerhard Habiger, Jakob Pietron, and Frank Kargl. Consistent retrospective snapshots in distributed event-sourced systems. In *2017 International Conference on Networked Systems, NetSys 2017*, 2017.

[FMF13] Nickerson Ferreira, Pedro Martins, and Pedro Furtado. Near real-time with traditional data warehouse architectures. 2013.

[Fow05] Martin Fowler. Event Sourcing, 2005.

[GCLZ18] Varun Grover, Roger H.L. Chiang, Ting Peng Liang, and Dongsong Zhang. Creating Strategic Business Value from Big Data Analytics: A Research Framework. *Journal of Management Information Systems*, 2018.

[GKK12] Ken Goodhope, Joel Koshy, and Jay Kreps. Building LinkedIn's Real-time Activity Data Pipeline. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2012.

[HB16] Sara Hassan and Rami Bahsoon. Microservices and their design trade-offs: A self-adaptive roadmap. In *Proceedings - 2016 IEEE International Conference on Services Computing, SCC 2016*, 2016.

[Inm02] W. H. Inmon. *AM Building the*. 2002.

[KAEM13] Stephen Kaisler, Frank Armour, J. Alberto Espinosa, and William Money. Big data: Issues and challenges moving forward. In *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2013.

[KR02] Ralph Kimball and Margy Ross. The data warehouse toolkit: the complete guide to dimensional modelling. *Career: Data and Analytics*, 2002.

[Lew14] M. Lewis, J. and Fowler. No Title, 2014.

[MB12] Andrew McAfee and Erik Brynjolfsson. Big data: The management revolution. *Harvard Business Review*, 2012.

[MDL$^+$13] Gilad Mishne, Jeff Dalton, Zhenghua Li, Aneesh Sharma, and Jimmy Lin. Fast data in the era of big data. 2013.

[MK00] Daniel L Moody and Mark A R Kortink. From Enterprise Models to Dimensional Models: A Methodology for Data Warehouse and Data Mart Design. *DMDW*, 2000.

[MSMV07] Greg Minshall, Yasushi Saito, Jeffrey C. Mogul, and Ben Verghese. Application performance pitfalls and TCP's Nagle algorithm. *ACM SIGMETRICS Performance Evaluation Review*, 2007.

[Neh10] Gwen Shapira & Todd Palino Neha Narkhede. *Kafka the Definitive Guide*. 2010.

[NMMA16] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. 2016.

[ODEG13] M. Obali, B. Dursun, Z. Erdem, and A. K. Gorur. A real time data warehouse approach for data processing. 2013.

[PKS$^+$18] Alok Pareek, Bhushan Khaladkar, Rajkumar Sen, Basar Onat, Vijay Nadimpalli, and Mahadevan Lakshminarayanan. Real-time ETL in Striim. 2018.

[PLA18] Gautam Pal, Gangmin Li, and Katie Atkinson. Big Data Real Time Ingestion and Machine Learning. In *Proceedings of the 2018 IEEE 2nd International Conference on Data Stream Mining and Processing, DSMP 2018*, 2018.

[PP15] Pekka Pääkkönen and Daniel Pakkala. Reference Architecture and Classification of Technologies, Products and Services for Big Data Systems, 2015.

[Rag18] Elisabetta Raguseo. Big data technologies: An empirical investigation on their adoption, benefits and risks for companies. *International Journal of Information Management*, 2018.

[RSA$^+$18] M. Mazhar Rathore, Hojae Son, Awais Ahmad, Anand Paul, and Gwanggil Jeon. Real-Time Big Data Stream Processing Using GPU with Spark Over Hadoop Ecosystem. *International Journal of Parallel Programming*, 2018.

[SA12] AAR El Sheikh and Mouhib Alnoukari. *Business Intelligence and Agile Methodologies for Knowledge-Based Organizations: Cross-Disciplinary Applications*. 2012.

[SAS$^+$17] Adilah Sabtu, Nurulhuda Firdaus Mohd Azmi, Nilam Nur Amir Sjarif, Saiful Adli Ismail, Othman Mohd Yusop, Haslina Sarkan, and Suriayati Chuprat. The challenges of Extract, Transform and Loading (ETL) system implementation for near real-time environment. In *International Conference on Research and Innovation in Information Systems, ICRIIS*, 2017.

[SCT11] David Schuff, Karen Corral, and Ozgur Turetken. Comparing the understandability of alternative data warehouse schemas: An empirical study. *Decision Support Systems*, 2011.

[SM13] Rainer Schmidt and Michael Möhring. Strategic alignment of cloud-based architectures for big data. In *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, 2013.

[SML18] Robert Stricker, Daniel Müssig, and Jörg Lässig. Microservices for Redevelopment of Enterprise Information Systems and Business Processes Optimization. 2018.

[STV18] Jacopo Soldani, Damian Andrew Tamburri, and Willem Jan Van Den Heuvel. The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software*, 2018.

[TLPJ17] Davide Taibi, Valentina Lenarduzzi, Claus Pahl, and Andrea Janes. Microservices in Agile Software Development: a Workshop-Based Study into Issues, Advantages, and Disadvantages. In *Proceedings of the XP2017 Scientific Workshops on - XP '17*, 2017.

[VS08] Panos Vassiliadis and Alkis Simitsis. Near Real Time ETL. 2008.

[XLWW18] F. Xiao, C. Li, Z. Wu, and Y. Wu. NMSTREAM: A scalable event-driven ETL framework for processing heterogeneous streaming data. In *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 2018.

[XM14] Cristiano Xavier and Fernando Moreira. Agile ETL. *Procedia Technology*, 2014.

REFERENCES

Appendix A

Kakfa Consumer Class

```python
from confluent_kafka.cimpl import Consumer, TopicPartition, KafkaException
from io import BytesIO
from messaging.consumer import ConfluentKafkaConsumer, ConsumerException

class ConfluentKafkaConsumerCustom(ConfluentKafkaConsumer):
    def __init__(self, conf, topics, partitions, logger: (Logger, None)):
        super(ConfluentKafkaConsumerCustom, self).__init__(conf, topics, logger)
        self.partitions = partitions

    def __enter__(self):
        self.consumer = Consumer(self.conf)
        if isinstance(self.partitions, list):
            list_topic_partitions = []
            for partition in self.partitions:
                list_topic_partitions.append(TopicPartition(self.topics[0].name(), partition))
            self.consumer.assign(list_topic_partitions)
        else:
            self.consumer.assign([TopicPartition(self.topics[0].name(), self.partitions)])
        self.__consumer_assignment_report(self.topics[0].name(), self.partitions)

        return self

    def consume(self, num_max_messages, timeout):
        messages=self.consumer.consume(num_messages=num_max_messages, timeout=timeout)
        if messages is None:
                return None
            return self.deserialize_messages_values(messages)

    def consume_messages(self, num_max_messages, timeout):
        messages = self.consumer.consume(num_messages=num_max_messages, timeout=timeout)
        if messages is None:
            return None
        if messages.error():
            if messages.error().fatal():
                raise ConsumerException() from KafkaException(messages.error())
            return None
        return messages

    def deserialize_messages_values(self, messages):
        list_msgs_deserialized = []
        for msg in messages:
            topic = next(x for x in self.topics if x.name() == msg.topic())
            item_type_name = dict(msg.headers())['item_type_name'].decode('utf-8')
            with BytesIO(msg.value()) as buff:
                list_msgs_deserialized.append(topic.deserialize(buff, item_type_name))
        return list_msgs_deserialized

    def __consumer_assignment_report(self, topic, partitions):
        self.__log_info_msg('Topic: {} | Partitions assigned: {}'.format(topic, partitions))

    def __log_info_msg(self, message):
        if self.logger is None:
            print(message)
        else:
            self.logger.info(message)
```

Appendix B

Initialization of Main Process

```
class Command(BaseCommand):
    must_exit = Event()

    def __init__(self):
        super().__init__()
        self.help = 'Launch a Kafka consumer'
        self.config = settings.KAFKA['consumer']
        self.poll_timeout = settings.KAFKA['consumer_service']['poll.timeout'
            ]
        self.topics = settings.CONSUMER_TOPICS
        signal.signal(signal.SIGINT, Command._handle_SIGINT)

    @staticmethod
    def _handle_SIGINT(signal, frame):
        Command.must_exit.set()

    def add_arguments(self, parser):
        parser.add_argument(
            '--env',
            action='store',
            dest='env',
            default='uat',
            choices=['prod', 'uat', 'dev'],
            help='Specifies whether the environment is production, user
                acceptance testing or development.'
        )
```

Initialization of Main Process

Appendix C

Parallel Kafka Consumers

```
def handle(self, *args, **options):
    start_time = time.time()
    processes = []
    list_partitions = list(range(0, NUM_PARTITIONS))

    partitions_per_consumer = math.ceil(NUM_PARTITIONS / NUM_CONSUMERS)

    list_partitions = list(get_partitions_by_consumer(list_partitions,
        partitions_per_consumer))
    logger.info(list_partitions)

    for i in range(0, len(list_partitions)):
        p = Process(target=consumer_init, args=(self, list_partitions[i], i,
            options,))
        logger.info("LAUNCHING {} CONSUMER(s) FOR PARTITION(s): {}".format(
            NUM_CONSUMERS, list_partitions[i]))
        processes.append(p)
        p.start()

    for process in processes:
        process.join()
```

Appendix D

Micro Batch Implementation

```python
def consumer_init(self, partitions, process_num, options):
    total_number_evts = 0
    loop = 1
    with ConfluentKafkaConsumerCustom(settings.KAFKA['consumer'], TOPICS,
        partitions, logger) as consumer:
        for _ in range(100000):
            list_evts = []
            end_time = datetime.now() + timedelta(seconds=MAX_LATENCY)
            sub_loop = 1
            while datetime.now() < end_time and len(list_evts) <
                BIG_QUERY_NUM_MAX_ROWS:  # max number or rows per stream:
                100.000
                try:
                    num_max_rows=BIG_QUERY_NUM_MAX_ROWS-len(list_evts)     #
                        to prevent errors
                    evts_returned = consumer.consume(num_max_rows,
                        POLL_TIMEOUT)

                    list_evts += evts_returned
                    total_number_evts += len(evts_returned)
                    sub_loop += 1

                except Exception as e:
                    logger.info(e)

            loop += 1
            logger.info(
                "PROCESS {} | PARTITION {} | LOOP {} | SUBLOOPS {} | NUM
                    EVENTS FOR LOOP: {} | "
                "TOTAL NUM EVENTS: {} | NOW: {}".format(process_num,
                    partitions, loop, sub_loop, len(list_evts),
                    total_number_evts, time.time()))

            if list_evts:
                logger.info("PROCESS {} | Sending to BigQuery...".format(process_num))
                signals.dummy_evts_signal.send(sender=self.__class__, list_evts=list_evts, process_num=process_num, env=options['env'])
                delay_time = random.randrange(500, 1500)
                time.sleep(delay_time/1000)

                logger.info("PROCESS {} | NOW IS: {}".format(process_num, time.time()))
            else:
                logger.info("PROCESS {} | Nothing to send to BigQuery".format(process_num))
```

Appendix E

ETL Orchestrator Improvement

```
    INSERT `star_schema.dim_delivery` (
      SK_DELIVERY,
      COD_DELIVERY,
      COD_DELIVERY_NUM,
      DSC_DELIVERY_STATUS,
      DSC_SHIPPING_SERVICE_LEVEL,
      DAT_BRAND_DELIVERY_DATE,
      DAT_EXPECTED_DELIVERY_DATE,
      BOOL_IS_HOME_DELIVERY,
      CREATED_AT,
      UPDATED_AT,
      COD_TIMESTAMP
    )
    SELECT(
        (...)
    )
    (...)
    ORDER BY source.SK_DELIVERY
--NEXT QUERY --
INSERT `star_schema.dim_sales_order` (
  SK_SALES_ORDER
  ,COD_SALES_ORDER
)
SELECT
  source.SK_SALES_ORDER
  ,source.COD_SALES_ORDER
FROM (
  SELECT
    ROW_NUMBER() OVER (PARTITION BY innerQ.serial_generator ORDER BY innerQ.
        COD_SALES_ORDER) + (SELECT CASE WHEN MAX(SK_SALES_ORDER) IS NOT NULL
        THEN MAX(SK_SALES_ORDER) ELSE 0 END AS max_sales_order FROM `
        star_schema.dim_sales_order`) AS SK_SALES_ORDER
    ,innerQ.COD_SALES_ORDER
    FROM (
    SELECT
        1 AS serial_generator
        ,so.delivery_order_id AS COD_SALES_ORDER
    FROM (SELECT distinct delivery_order_id FROM `staging.pre_delivery` AS d)
         so
    WHERE so.delivery_order_id NOT IN (SELECT DISTINCT COD_SALES_ORDER FROM `
        star_schema.dim_sales_order`)
    ORDER BY so.delivery_order_id
  ) AS innerQ
) source
ORDER BY source.SK_SALES_ORDER
--NEXT QUERY --
INSERT `star_schema.dim_customer` (
  SK_CUSTOMER
  ,COD_CUSTOMER
)
SELECT
  source.SK_CUSTOMER
  ,source.COD_CUSTOMER
FROM (
  SELECT
    ROW_NUMBER() OVER (PARTITION BY innerQ.serial_generator ORDER BY innerQ.
        COD_CUSTOMER) + (SELECT CASE WHEN MAX(SK_CUSTOMER) IS NOT NULL THEN
        MAX(SK_CUSTOMER) ELSE 0 END AS max_customer FROM `star_schema.
        dim_customer`) AS SK_CUSTOMER
    ,innerQ.COD_CUSTOMER
    FROM (
    SELECT
        1 AS serial_generator
        ,so.customer_id AS COD_CUSTOMER
    FROM (SELECT distinct customer_id FROM `staging.pre_delivery` AS d) so
      WHERE so.customer_id NOT IN (SELECT DISTINCT COD_CUSTOMER FROM `
          star_schema.dim_customer`)
    ORDER BY so.customer_id
  ) AS innerQ
) source
ORDER BY source.SK_CUSTOMER
--NEXT QUERY --
INSERT `star_schema.dim_customer_address` (
  SK_CUSTOMER_ADDRESS
  ,COD_CUSTOMER_ADDRESS
)
SELECT
  source.SK_CUSTOMER_ADDRESS
  ,source.COD_CUSTOMER_ADDRESS
FROM (
  SELECT
```

```sql
    ROW_NUMBER() OVER (PARTITION BY innerQ.serial_generator ORDER BY innerQ.
        COD_CUSTOMER_ADDRESS) + (SELECT CASE WHEN MAX(SK_CUSTOMER_ADDRESS) IS
         NOT NULL THEN MAX(SK_CUSTOMER_ADDRESS) ELSE 0 END AS
        max_customer_address FROM `star_schema.dim_customer_address`) AS
        SK_CUSTOMER_ADDRESS
    ,innerQ.COD_CUSTOMER_ADDRESS
    FROM (
    SELECT
        1 AS serial_generator
        ,so.customer_address_id AS COD_CUSTOMER_ADDRESS
    FROM (SELECT distinct customer_address_id FROM `staging.pre_delivery` AS
        d) so
      WHERE so.customer_address_id NOT IN (SELECT DISTINCT
          COD_CUSTOMER_ADDRESS FROM `star_schema.dim_customer_address`)
    ORDER BY so.customer_address_id
  ) AS innerQ
) source
ORDER BY source.SK_CUSTOMER_ADDRESS
--NEXT QUERY --
INSERT `star_schema.dim_customer_address` (
  SK_CUSTOMER_ADDRESS
  ,COD_CUSTOMER_ADDRESS
)
SELECT
  source.SK_CUSTOMER_ADDRESS
  ,source.COD_CUSTOMER_ADDRESS
FROM (
  SELECT
    ROW_NUMBER() OVER (PARTITION BY innerQ.serial_generator ORDER BY innerQ.
        COD_CUSTOMER_ADDRESS) + (SELECT CASE WHEN MAX(SK_CUSTOMER_ADDRESS) IS
         NOT NULL THEN MAX(SK_CUSTOMER_ADDRESS) ELSE 0 END AS
        max_shipping_address FROM `star_schema.dim_customer_address`) AS
        SK_CUSTOMER_ADDRESS
    ,innerQ.COD_CUSTOMER_ADDRESS
    FROM (
    SELECT
        1 AS serial_generator
        ,so.shipping_address_id AS COD_CUSTOMER_ADDRESS
    FROM (SELECT distinct shipping_address_id FROM `staging.pre_delivery` AS
        d) so
      WHERE so.shipping_address_id NOT IN (SELECT DISTINCT
          COD_CUSTOMER_ADDRESS FROM `star_schema.dim_customer_address`)
    ORDER BY so.shipping_address_id
  ) AS innerQ
) source
ORDER BY source.SK_CUSTOMER_ADDRESS
```

AUDIT files add also to be changed. In order to update all rows, since before it was updating only rows that carried a smaller ID than the one saved in the audits. The lines were commented.

```
        (...)
             FROM `staging.pre_sales_order` AS so
             --WHERE so.id <= (@#id)
             --AND so.cod_updated_at > (@#cod_date)
             ORDER BY so.id
           ) AS innerQ
        (...)
```

The Orchestrator also had to change. It was set to read a dmlFile with only one query. After adding the code to handle dependencies, this INSERT files has more than one INSERT query. So the file had to be split up and for each INSERT, and the orchestrator runs for each query.

```
    (...)
    if dmlFile is not None:
            # Get update query
            ss_all_queries = fileHelper.read_file(file_name=dmlFile, layer=
                package['DSC_TABLE_DESTINATION_DATASET'])

            ss_queries_list = ss_all_queries.split('--NEXT QUERY --')

            for ss_query in ss_queries_list:
                -- (...) starts replacing dmlFiles and insert in star schema
```

Appendix F

Producer Kafka Class

```

def produce_create_delivery(event_id):

    with ConfluentKafkaProducer(KAFKA['producer'], None) as p:
        for _ in range(NUMBER_OF_EVENTS):

            try:
                cmd = delivery_data(event_id, time.time_ns())
                payload = CreateDeliveryV2Command(**cmd)
                logger.info('Payload: {}'.format(payload))
                p.produce_sync(DeliveryCommandsV2Topic(), payload)
                ts = time.time()
                logger.info('EVENT {} produced at {} -  PUBLISHED'.format(
                    event_id, datetime.fromtimestamp(ts).strftime('%Y-%m-%d %
                    H:%M:%S,%f')))
                event_id += 1

            except ProducerException:
                logger.info('ERROR ON PRODUCER')
                raise

        logger.info("PRODUCTION OVER")
```