UNIVERSITY OF PIRAEUS

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGIES

DEPARTMENT OF DIGITAL SYSTEMS

POSTGRADUATE PROGRAM

INFORMATION SYSTEMS AND SERVICES

SPECIALIZATION

BIG DATA AND ANALYTICS

MASTER THESIS

TRAJECTORY ANALYSIS OF MOVING VEHICLES IN REAL TIME

by

KONSTANTINOS GIANNOPOULOS

SUPERVISOR: CHRISTOS DOULKERIDIS

PIRAEUS, 2022

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Christos Doulkeridis, for his invaluable assistance and guidance during this thesis project. He has always been there to offer me his helpful advice and his extensive knowledge. Without his contribution, this thesis would have never been fulfilled.

Furthermore, I am extremely grateful to Mr. Athanasios Koumparos, who offered me the dataset that was used throughout this thesis and for sharing his knowledge with me.

I would also like to express my absolute love and gratitude to my parents, Panagiotis and Georgia, and my sister, Marianna, for their tremendous encouragement and their trust in my capabilities. Their unconditional love, patience and support made this project possible.

Lastly, I would like to thank my beloved friend Isidora, who always believes in me. I wholeheartedly thank her for her support and her unlimited patience.This master thesis is dedicated to her and my family.

Abstract

The ever-increasing production of spatiotemporal data, the increasingly velocity of their production as well as the development of real-time streaming data analysis systems, have raised new challenges in the field of data analysis. The analysis of trajectory data of moving objects is crucial for companies and organizations, which possess, and therefore need to effectively manage, fleets of vehicles. Efficient fleet management is based on quick and effective decision making. Thus, real-time and effective analysis of GPS emitted data is of vital importance.

In this Diploma Thesis, we propose a system that processes the GPS emitted data in real-time and creates a concise and explanatory report of each vehicle's trips. One of the major challenges that real-time streaming data analysis systems are facing, is the effective processing of the delayed, out-of-order data. Our proposed implementation addresses this issue since it effectively detects and processes data that reaches the system in wrong chronological order.

The implementation of the proposed system is based on the use of scalable technologies. Apache Kafka was adopted as storage layer for the GPS emitted data. Processing of stored data is undertaken by Apache Flink, which is capable of distributed processing bounded and unbounded data streams, with high throughput, low latency and in a fault-tolerant way. The processed data are then stored into Elasticsearch, which allows fast search and retrieval of the required statistics. The last step of the proposed implementation is the visualization of the stored statistics. Kibana is used to create all the necessary dashboards, providing the end user with a high overview of the results.

This thesis is structured as follows: First, the theoretical foundations of big data analysis for both streaming and bounded data are presented. Then, the most popular real-time stream processing frameworks are presented and compared. Moreover, we explain in detail the proposed architecture, how our application is processing data that reach the system in chronological order and how it handles the delayed data. Finally, we present the performance of the implemented system.

Keywords: *Spatiotemporal Data, Big Data, Real-Time Data Analytics, Stream Processing Frameworks*

Περίληψη

Η συνεχώς αυξανόμενη παραγωγή χωροχρονικών δεδομένων, η ταχύτητα της παραγωγής τους καθώς και η ανάπτυξη των συστημάτων ανάλυσης ροών δεδομένων, έχουν οδηγήσει στη δημιουργία νέων προκλήσεων στον τομέα της ανάλυσης δεδομένων. Η ανάλυση δεδομένων τροχιάς είναι νευραλγικής σημασίας για εταιρείες και οργανισμούς που διαθέτουν στόλο οχημάτων, τον οποίο οφείλουν να διαχειρίζονται αποτελεσματικά. Η σωστή διαχείριση του στόλου στηρίζεται στη γρήγορη λήψη αποφάσεων. Είναι επομένως σημαντικό, η ανάλυση δεδομένων που εκπέμπονται από συσκευές εγκατεστημένες στα οχήματα, να γίνεται αποτελεσματικά και σε πραγματικό χρόνο.

Στη συγκεκριμένη διπλωματική εργασία, προτείνουμε ένα σύστημα το οποίο επεξεργάζεται τα δεδομένα που εκπέμπονται από τα οχήματα, σε πραγματικό χρόνο και δημιουργεί μία συνοπτική και επεξηγηματική αναφορά για τα ταξίδια που πραγματοποίησε το κάθε όχημα. Μία από τις σημαντικότερες δυσκολίες που αντιμετωπίζουν τα συστήματα ανάλυσης ροών δεδομένων σε πραγματικό χρόνο είναι η σωστή διαχείριση εκείνων των δεδομένων που καταφθάνουν αργοπορημένα στο σύστημα. Η υλοποίηση μας αντιμετωπίζει αυτό το πρόβλημα, καθώς εντοπίζει και επεξεργάζεται αποτελεσματικά τα δεδομένα που φτάνουν στο σύστημα σε λανθασμένη χρονολογική σειρά.

Για την υλοποίηση του παραπάνω συστήματος στόχος ήταν η χρησιμοποίηση κλιμακώσιμων τεχνολογιών. Για την αποθήκευση των δεδομένων που εκπέμπονται από τις εγκατεστημένες στα οχήματα συσκευές, χρησιμοποιήθηκε το Apache Kafka. Την επεξεργασία των αποθηκευμένων δεδομένων την αναλαμβάνει το Apache Flink, το οποίο είναι ικανό να επεξεργάζεται συνεχείς αλλά και οριοθετημένες ροές δεδομένων, κατανεμημένα, σε υψηλό ρυθμό, με ελάχιστη καθυστέρηση και με ανοχή στα σφάλματα. Στη συνέχεια, τα επεξεργασμένα δεδομένα, αποθηκεύονται με τη βοήθεια του Elasticsearch, το οποίο επιτρέπει τη γρήγορη αναζήτηση των στατιστικών που επιθυμεί ο χρήστης. Τέλος, το Kibana αναλαμβάνει την οπτικοποίηση των αποθηκευμένων δεδομένων.

Η δομή της παρούσας διπλωματικής εργασίας είναι η ακόλουθη: Αρχικά, παρουσιάζεται το θεωρητικό υπόβαθρο που αφορά την επεξεργασία μεγάλων δεδομένων, τόσο οριοθετημένων όσο και συνεχών ροών. Στη συνέχεια, γίνεται παρουσίαση και σύγκριση των διαφόρων συστημάτων επεξεργασίας ροών δεδομένων. Τέλος, εξηγείται εκτενώς η προτεινόμενη αρχιτεκτονική, παρουσιάζονται οι επιδόσεις του συστήματος, καθώς και ο τρόπος επεξεργασίας τόσο των δεδομένων που φτάνουν στη σωστή χρονολογική σειρά, όσο και των δεδομένων που φτάνουν στο σύστημα αργοπορημένα.

Λέξεις Κλειδιά: Χωροχρονικά Δεδομένα, Μεγάλα Δεδομένα, Ανάλυση Δεδομένων σε Πραγματικό Χρόνο, Συστήματα Επεξεργασίας Συνεχών Ροών Δεδομένων

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

In modern era, the rapid growth of GPS (Global Positioning System) enabled devices has led to the production of vast amount of trajectory data and has raised new challenges for the analysis of big mobility data. The need of efficient analysis of this kind of data applies to different domains of industry, such as urban, marine, and air-traffic management. There are several applications, where data generated by devices installed on vehicles, ships or aircraft needs to be processed with minimal latency. For example, a system that detects anomalies in trajectories of moving objects or dangerous driving behavior, should process data as soon as they reach the system. With that said, we can easily understand why research interest has focused on real-time analysis of data streams that are emitted by moving objects. This thesis focuses on real-time trajectory analysis of fleets of commercial vehicles, which constitutes an extremely important factor of European and global economy.

In this chapter, we provide the motivation, introduce the problem under consideration and present the objective and the challenges of this thesis. Finally, we provide the document's structure.

1.1. Motivation & Problem Statement

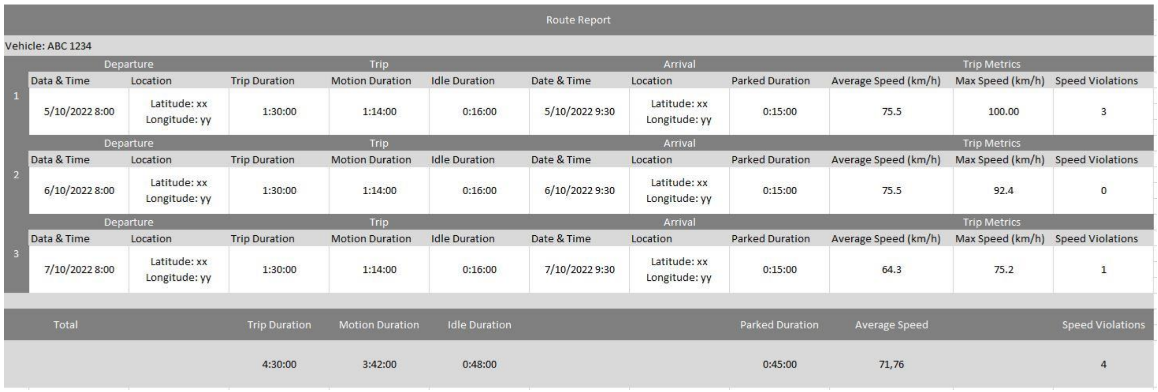
According to the European Automobile Manufacturers Association, in 2020, the European Union's passenger car fleet grew by 1,2% compared to 2019, with approximately 246 million cars on the road in total, whereas there are more than 6 million medium and heavy commercial vehicles on European Union's roads [14]. Thus, vehicle fleet management is of crucial importance for many companies. Installed devices on commercial vehicles emit enormous amounts of spatiotemporal and operational data. Collection and analysis of these data can provide meaningful insights into different aspects of fleet management, such as fleet maintenance, driver behavior, economical driving, fuel consumption, planning and managing of the transportation services. Data streams that are emitted from moving vehicles must be processed with low latency for better fleet monitoring and on time decision-making.

For a human to understand this amount of information, data aggregation needs to take place and transform the emitted GPS data stream into a condensed, meaningful, and clear piece of information. In this diploma thesis we assume that the emitted GPS records contain the following information:

- Id of the vehicle that the GPS device is installed on
- Timestamp of the emitted GPS record
- Engine Status of the vehicle
- Geographic coordinates
- Speed of the vehicle

Even though various analysis results can be generated, as mentioned above, the basis information is the vehicle's route report [Figure 1]. This report describes the trip that a vehicle performed since the engine started, up to the point the driver switched it off, offering a high level of easily understandable information by providing the following data within a single line:

- Date and time of trip start
- Coordinates of starting location
- Duration of the trip
- Temporary stop duration (speed is equal to 0 but engine is on)
- Parked duration (time between engine off and engine on states)
- Average speed
- Maximum speed
- Speed violations (based on a predefined threshold)
- Coordinates of ending location
- Date and time of trip end



Route Report

Vehicle: ABC 1234

	Departure		Trip			Arrival			Trip Metrics		
	Data & Time	Location	Trip Duration	Motion Duration	Idle Duration	Date & Time	Location	Parked Duration	Average Speed (km/h)	Max Speed (km/h)	Speed Violations
1	5/10/2022 8:00	Latitude: xx Longitude: yy	1:30:00	1:14:00	0:16:00	5/10/2022 9:30	Latitude: xx Longitude: yy	0:15:00	75.5	100.00	3
	Departure		Trip			Arrival			Trip Metrics		
	Data & Time	Location	Trip Duration	Motion Duration	Idle Duration	Date & Time	Location	Parked Duration	Average Speed (km/h)	Max Speed (km/h)	Speed Violations
2	6/10/2022 8:00	Latitude: xx Longitude: yy	1:30:00	1:14:00	0:16:00	6/10/2022 9:30	Latitude: xx Longitude: yy	0:15:00	75.5	92.4	0
	Departure		Trip			Arrival			Trip Metrics		
	Data & Time	Location	Trip Duration	Motion Duration	Idle Duration	Date & Time	Location	Parked Duration	Average Speed (km/h)	Max Speed (km/h)	Speed Violations
3	7/10/2022 8:00	Latitude: xx Longitude: yy	1:30:00	1:14:00	0:16:00	7/10/2022 9:30	Latitude: xx Longitude: yy	0:15:00	64.3	75.2	1
	Total		Trip Duration	Motion Duration	Idle Duration			Parked Duration	Average Speed		Speed Violations
			4:30:00	3:42:00	0:48:00			0:45:00	71,76		4

Figure 1: Example of route report

In the most common implementation, GPS emitted data are stored into a database and when the user requests the route report, data aggregations take place and statistics of each trip are calculated for the route report to be generated. This process takes significant amount of time to be completed and leads to poor customer experience. In this thesis we try to overcome these drawbacks, by proposing an architecture that will calculate the required statistics when data arrive at the system and not when user asks for the route report.

To be more specific, the problem that this diploma thesis aims to address is the following: Given that emitted GPS data contain the information that we described above, the route report [Figure 1] should be generated. For the route report to be generated, the computations of the required values of each trip should be triggered when the emitted GPS data arrive at the system, and not when the user requests the report. Thus, no raw GPS data will be stored, but only the already calculated values of a given trip.

A challenging part of this implementation is that for a plethora of reasons, for example low signal strength of the installed device, data are sometimes outdated. These data must be processed in an effective way in order not to lose any important information. Since the required values of the trip are updated every time a new GPS record arrives at the system, there is a chance that not all the corresponding data of the trip have been considered in the calculations of the trip's statistics. When the missing, outdated GPS record arrives at the application, the already computed values should be corrected. The efficient recalculations of the already computed values of a given trip is one of the most challenging parts of this thesis.

1.2. Objective of this Thesis

This thesis project aims to implement a highly efficient system that analyzes the trajectory data of moving vehicles in real-time. The proposed implementation needs to efficiently process the emitted GPS data as soon as they arrive at the system, with low latency, and produce the required statistics, for the route report [Figure 1] to be generated.

Since the calculations of the required values are performed in real-time, on each one of the emitted GPS data, another requirement of this thesis is to address and correctly handle data that may arrive at the system later than expected. In order to correctly calculate the desired statistics of a trip and not produce incorrect results, effective handling of the delayed, out-of-order data is of crucial importance for the implemented system.

The calculated statistics should also be stored in an efficient way so that they can be retrieved as quickly as possible when requested by the user. Finally, data should be effectively visualized to provide users with a high level of monitoring of each vehicle's trip.

Right below we summarize the main goals of this thesis:

Goal 1: Developing an application that will calculate in real-time the values needed to generate the route report [Figure 1]. It is of crucial importance that the designed application processes the delayed data efficiently, in order to correctly update the already computed values and provide correct and meaningful insights.

Goal 2: Storing the required values in an efficient way so that the end user can access them quickly. A visualization layer should be added to provide the end user with a high overview of the statistics of each trip.

1.3. Thesis Structure

The remaining of this thesis is structured as follows:

Chapter 2 presents the basic characteristics of big data, the basic concepts of big data technologies and big data processing, and the different architectures of stream processing. It addresses the main differences between batch and stream processing. Mainly, focus is given to the data stream processing, for which key concepts and architectures are presented.

Chapter 3 introduces the technical background for big data stream processing and big data storage. Initially we present the most common used technologies for stream processing. Apache Storm, Apache Spark Streaming, Apache Flink, Apache Samza and Kafka Streams are briefly presented. We also go through comparisons of these tools and we discuss their main differences, the advantages, and the disadvantages of each one. Finally, we present the different types of NoSQL databases, which are by their nature the most suitable storage option for big data.

Chapter 4 presents the overall architecture of the proposed implementation, the technologies and methodologies that have been used. Additionally, in this chapter we discuss the logic that we used to compute the required trip statistics, and how we handle the calculations on out-of-order data. Finally, we discuss how the processed data are stored, and we present the dashboards that visualize the calculated statistics.

Chapter 5 introduces the dataset that was used throughout this thesis and discusses the performance of our proposed solution.

Chapter 6 summarizes the results of this thesis project and indicates the directions for future work.

2. Theoretical Background

In this chapter, we provide an overview of the theoretical concepts related to this thesis. Initially we discuss definition and characteristics of big data. Subsequently, we present the different big data processing approaches, focusing on stream processing. Finally, we consider stream processing architectures and core concepts.

2.1. Big Data

Over the last years, the wide use of Internet, social medias, and the extremely fast development of IoT (Internet of Things) have led to exponential growth of produced data. Enhanced medical devices, wearables, GPS tracking devices, factory automation sensors are only a small number of an endless list of products which contribute to the creation of huge amounts of data. According to Statista, it is expected that by 2025, the digital information will grow up to 181 zettabytes [Figure 2]. Hence, parallel and distributed process is of vital importance for overcoming scale and timeline challenges.

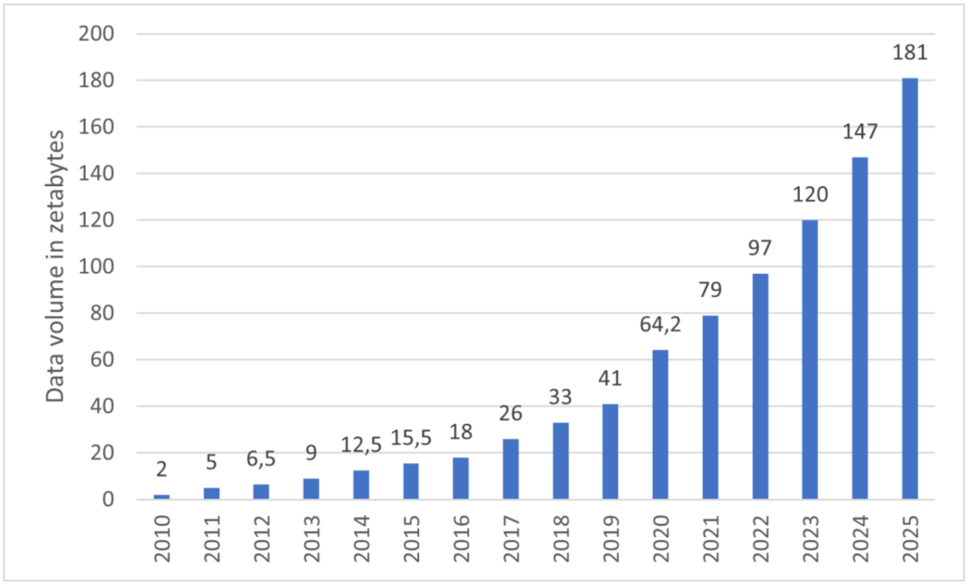


Figure 2: Data quantity in zettabytes [15]

2.1.1. Definition and Characteristics

Big data is a term that is used to describe huge and complex data that exceeds the capabilities of traditional database systems to process them. Big data has been characterized by Douglas Laney using 3 "Vs", namely Volume, Velocity and Variety [16]:

- *Volume*: refers to the huge data quantity that is generated by a variety of devices as we mentioned above.
- *Velocity:* refers to the rapidly increasing speed at which new data are generated, and the corresponding need for that data analyzed in near real-time.
- *Variety:* refers to the fact that big data can be produced in any type of format. Big data could be structured, such as numeric data in a traditional database, semi-structured, such as emails, or even unstructured, such as texts, videos, and images.

Later, IBM, in response to the quality issues that their clients began facing using big data solutions, introduced an additional "V", namely Veracity.

- *Veracity:* refers to the consistency, uncertainty, and trustworthiness of data.

In 2013, Yuri Demchenko proposed a 5 "Vs" big data definition. This new characteristic was Value.

- *Value:* refers to one of the most important aspects of big data, the potential insights that big data analysis can lead to.

Aiming at maximizing the business value, Microsoft extended the initial characterization of big data, added a 6 "V", Visibility.

- *Visibility:* refers to the need of having a full picture of data in order to make a meaningful decision.

The aforementioned characteristics are not the one and only used to explain the nature of big data. There is not a unified definition to describe big data, and characteristics that are good explanation for big data on one domain may fail to apply to other domains. For example, the data domain may be explained sufficiently by using the characteristics: volume, velocity, and variety, but for the statistics domain validity, veracity, and variability are more suitable explanation of big data.

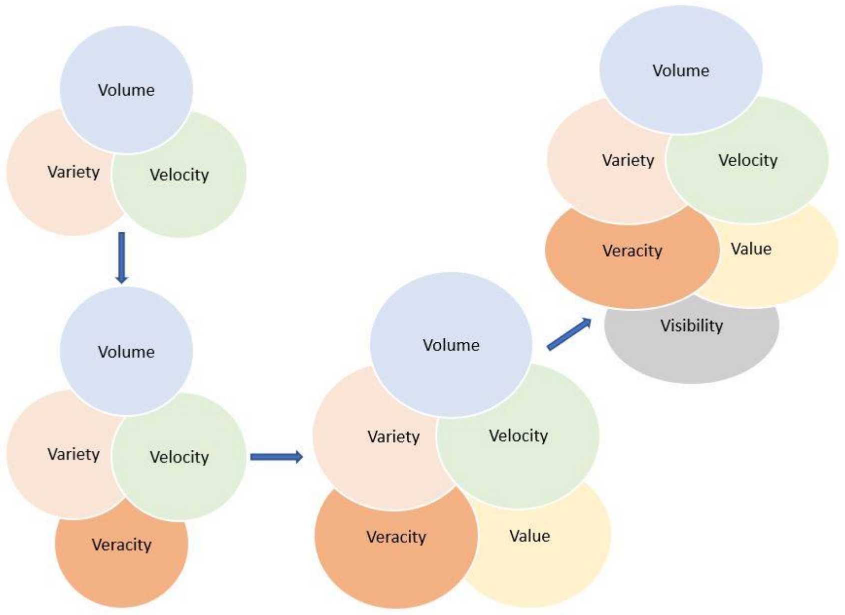


Figure 3: From Laney's 3Vs definition of Big Data to Microsoft's 6Vs [16]

2.2. Big Data Processing

As mentioned above, parallel, and distributed processing concepts of big data are of preeminent importance. Two different approaches are commonly used when building distributing big data processing systems: batch and stream processing.

2.2.1. Batch Processing

In *batch processing* approach, the system reads a large amount of data input, which is usually stored in a distributed file system, processes it in batches and produces output data. This processing method is used when we first store data and analytics are performed at a later stage. Batch jobs are usually scheduled to run periodically and automated without user's interaction with the system. Depending on the scheduling intervals, the size of data and computational capabilities of the system, it may take from hours to days until data processing is completed. MapReduce [4] is a fairly low-level batch processing programming model. For a MapReduce job to be created two functions needs to be developed:

Map function, which is called once for every input record and creates a set of intermediate key-value pair for each one. MapReduce framework groups together all the pairs with the same intermediate key and then passes them to the reducer.

Reduce function, which takes as input the key-value pairs iterates over this collection of values and produces output records.

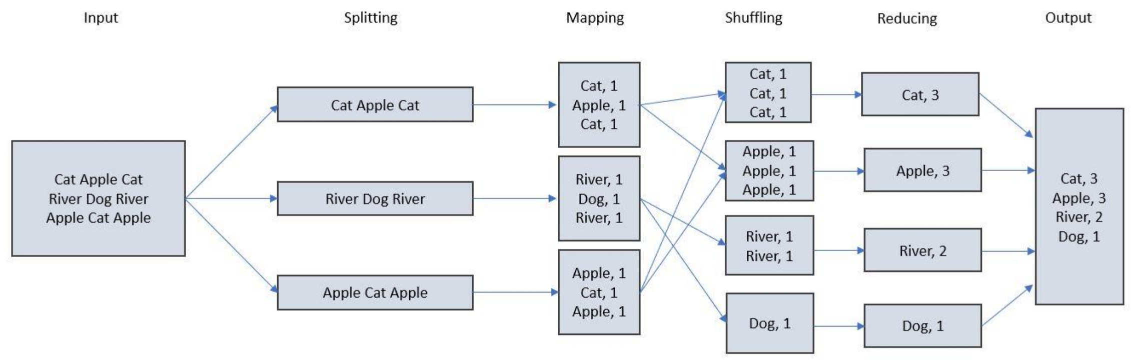


Figure 4: MapReduce word count process [43]

Hadoop was developed on top of the MapReduce paradigm and was designed as a distributed batch processing system. Data are stored in a distributed filesystem, called Hadoop Distributed File System (HDFS), divided into smaller chunks, to support MapReduce processing. Although Hadoop has facilitated distributed programming, has achieved good scalability and fault tolerance, it is unsuitable for real-time stream processing, since all data must be transformed by *map function* before the *reduce function* starts executing. Except from its lack of real-time processing there are several other weaknesses that comes along with MapReduce [5].

2.2.2. Stream Processing

As discussed in the previous paragraph, the batch processing model requires all the input to be stored before computation is triggered. In contrast to the batch processing model, a variety of modern applications receive data gradually over time, namely *data streams*, and require continuous, real-time processing. *Data streams* differ from stored data in several ways: data elements in the stream arrive online, the system has not control over the order of the data, data streams are potentially unbounded in size and processed data are discarded from the stream or archived [17]. Since these data could be unbounded, potentially infinite, and need to be processed in real-time with low latency and high throughput, the batch model lacks the capability to handle them efficiently, thus a different processing model should be adapted.

In *stream processing*, data are processed as soon as they arrive (on-the-fly), without need to be stored and retrieved in the future, so the output is produced in real-time.

2.3. Stream Processing Architectures

Developing a real-time stream processing application is inherently very challenging. Several design patterns have been proposed as a template for building real-time applications. Two of the most common architectures are lambda and kappa architecture.

The Lambda architecture [Figure 5], which was first proposed by Nathan Marz, is a generic, scalable and fault tolerant real-time data processing architecture [18]. To fulfill both batch and real-time processing, it comprises of three layers, named batch, serving and speed layer.

The Batch layer is responsible for storing the master copy of the dataset and precomputing the batch views on this dataset. This master dataset is immutable and ever growing, thus the batch layer should be able to store this kind of dataset and produce arbitrary views from it. When new input data arrive, the recomputing of batch views take place on the next batch iteration.

The Serving layer is a specialized distributed database in which the batch views are loaded and can be queried. The serving layer supports batch updates, so when new batch views are available, it automatically swaps the old ones out. Since the serving layer updates as soon as the batch layer finishes precomputing a batch view, data that arrived on the system while the precomputation was running are not available.

The Speed layer is responsible for ensuring that the more recent data will be available as quickly as needed by the application. Like the batch layer, the speed layer does also produce views, but with a key difference. It looks only at recent data, whereas batch layer looks at all data at once. Another difference is that, aiming at low latency, the speed layer does incremental computation instead of computation from scratch like batch layer. Thus, the implementation of the speed layer is the answer to the inability of batch and serving layers to present the latest data.

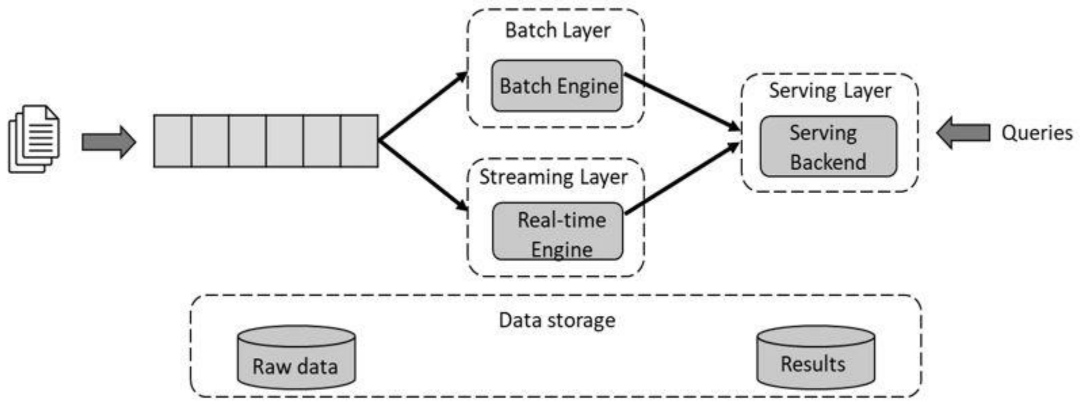


Figure 5: Lambda Architecture [41]

Even though the lambda architecture is fault-tolerant, scalable, reliable and a good choice for many use cases, such as large-scale machine learning models [19] still there are drawbacks. One major weakness is the code maintenance of two complex, distributed and heterogeneous systems. Hence, development related processes like debugging, implementation and synchronization of the different layers are exceedingly difficult to be handled.

In 2014, Jay Kreps, mentioning the existing disadvantages of lambda architecture, introduced an alternative approach for real-time processing applications, which is easier to implement and maintain, since it does not require implementation of two systems that works alongside [20]. This architectural proposal, called *Kappa architecture* [Figure 6], tries to simplify lambda architecture. In contrast to lambda architecture, kappa architecture lacks batch layer and consists only of real-time layer and serving layer. The basic idea is to handle both real-time data processing and data reprocessing using a single streaming system. One of the key challenges of stream processing is data reprocessing. Kappa architecture supports data reprocessing on code change. To achieve this behavior in an efficient way, when reprocessing needs to be executed, a second instance of stream processing job is triggered and starts processing the retained data. The output result of this process is stored into a new table. When the second job instance finishes the process, application queries the new table and the outdated job and table are deleted. The trade-off that comes with simplicity is that the absence of batch layer might result in errors during data processing or while updating the database.

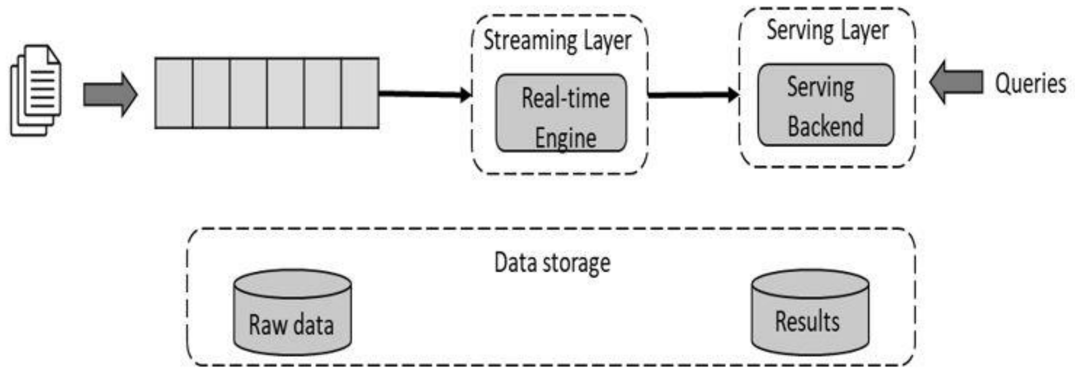


Figure 6: Kappa Architecture [41]

To conclude, the choice between lambda and kappa architecture is not obvious and depends on the specific characteristics of the application to be developed. If the main goal of the application is the simplicity, then Kappa architecture seems to be the best choice. On the other hand, if the reliability of the application is the priority, developers should consider the Lambda architecture.

2.4. *Stream Processing Concepts*

To better understand *stream processing,* we should dive into some of its key concepts. Stream processing execution can be categorized under two different models:

1. Stream Dataflow Approach, where an application is specified and given to the system for execution as a dataflow graph, which consists of tasks and data dependencies between them. Tasks encapsulate the logic of predefined operators, such as filter, window, aggregate, join, or even user defined operators. A master node receives a dataflow graph, which in most cases is a Directed Acyclic Graph (DAG), and schedules tasks among cluster.

2. Micro-Batch Approach, which handles a streaming computation as a sequence of transformations on bounded sets by discretizing a distributed data stream into batches, and then scheduling these batches sequentially in a cluster of worker nodes.

Another significant characteristic of stream processing methods is the so-called *state management*. Stream processing could be divided into two major groups: *stateful* and *stateless* [Figure 7]. A stateless operation creates an output, processing each event individually, while stateful operation creates output based on saved states of previous events, meaning that multiple events are taken into consideration. For example, if we want to develop an application which receives continuously speed data of a vehicle and our objective is to compute the average speed, we should consider a stateful implementation. Contrariwise, if our objective is to raise an alert every time the speed exceeds a predefined speed limit, the stateless implementation is more suitable.

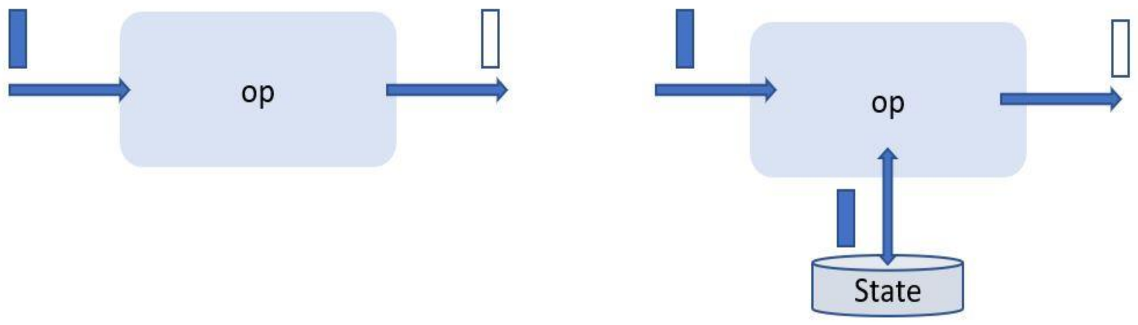


Figure 7: Representation of stateless and stateful stream processing [22]

Delivery guarantee is also an important factor of stream processing. It is a different term of "level of correctness" of the output produced after failure and successful system recovery, compared to the produced results, supposing that no failure has occurred. Stream processing is distinguished between the following different levels of consistency [22]:

- *At most once*, which implies that if a message was not successfully processed, no attempts will be made to deliver it. This is the least fault-tolerance approach, thus the least desirable.
- *At least once*, which implies that a message will be delivered at least once, until an acknowledgment of delivery is received. With this approach, if a failure occurs, messages will be redelivered, ensuring that the processing of the message was completed. Even though this approach provides greater fault tolerance, it can lead to additional cost for repeated processing. Also, for operations such as counting, at-least-once delivery guarantee can cause incorrect results.
- *Exactly once*, which guarantees that a message will be processed precisely once. It follows the same failure detection mechanism as the at-least-once model, but provides more acknowledgements checks to prevent the duplication of the same message. This is the most desirable mode.

In stream processing, time plays a crucial role. Generally, time is divided under two domains of interests:

- *Event time*, which indicates the time that an event actually happens on the real world. Usually, each generated event contains a timestamp attribute, which is part of the emitted data record itself.
- *Processing time*, which indicates the time that the system processed the input record. It is just the current time according to the system clock.

To better understand the difference between event and processing time, Figure 8 presents an illustrative example.

Efficient handling of time is of great interest in stream processing applications. In the real world, due to a variety of reasons, for example network delays, data do not arrive at the system in perfect order, as a result event time and processing time differs, and the ordering based on the event time is often not the same as the ordering based on the processing time. Both notions of time have their place in real-time applications. Some applications need results as fast as possible and they accept slightly inaccurate results, while other applications prioritizing correctness over speed. Finally, there is a chance that some applications need both speed and accuracy. For example, an application which counts the times an anomaly is observed but also provides an alert if an anomaly is detected, needs both the correctness of the counted numbers, but also the speed of the alert mechanism.

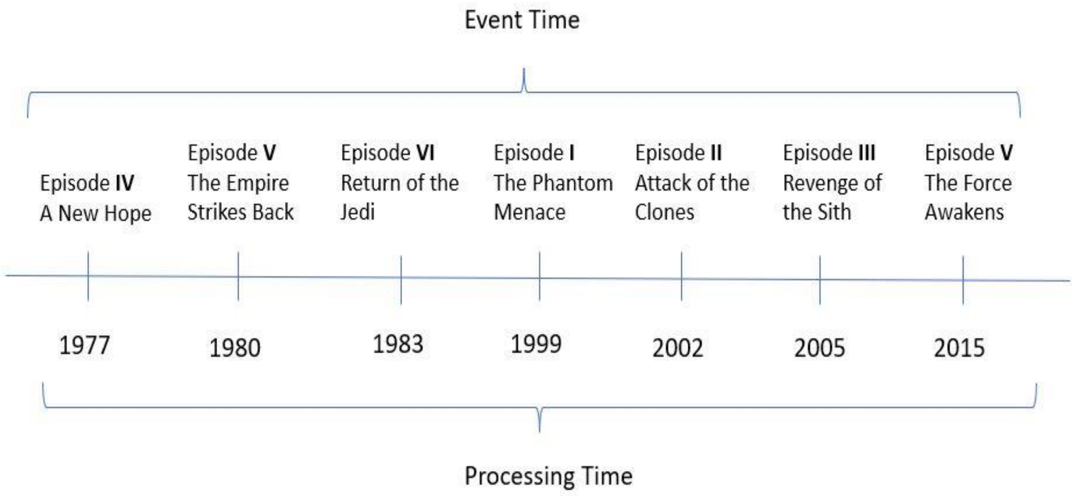


Figure 8: Difference between event and processing time [22]

3. Technical Background

In the era of big data and IoT, new challenges were raised for real-time data analysis. Stream processing frameworks are an ongoing process of great interest for researchers and technology companies. In this section, we briefly present some of the most common stream processing engines, we discuss their core concepts and provide comparisons between them. Finally, we discuss efficient storage techniques for big data applications, presenting the different types of NoSQL databases.

3.1. Stream Processing Frameworks

Nowadays, there is an increasing need for big data analysis to transition from offline (batch) to online (streaming) systems. To serve this need, various streaming algorithms and streaming platforms have been developed [41]. In this thesis we are going to focus on the most widely used open-source streaming frameworks: Apache Storm, Apache Samza, Apache Flink, Kafka Streams and Spark Streaming

3.1.1. Apache Storm

Apache Storm [23] is a scalable, distributed and fault-tolerant real-time processing engine, which was created by Nathan Marz at BackType and was open sourced after BackType was acquired by Twitter.

It implements the dataflow model by representing the entire stream processing pipeline as a DAG called *topology*. *Topology* is a graph network of computation, in which each node contains processing logic, while each edge represents the data pass through mechanism. The core abstraction of Strom is the *stream*, an unbounded sequence of tuples, that is processed in a parallel and distributed way. A tuple is simply a named list of values. By default, tuples can contain integers, longs, shorts, bytes, strings, doubles, floats, booleans, byte arrays, but also custom types defined by user. Every stream is given an id when declared. Vertices of topology are of two different kinds: *spouts* and *bolts.*

Spout is the source of streams in topology. Typically, a spout reads tuples from an external queuing broker, like Kafka or RabbitMQ, and emits them into the topology. There are two different types of spouts, reliable and unreliable. A reliable spout is capable of replaying a tuple if Storm fails to process it, whereas unreliable spout cannot replay any tuple as soon as it is emitted. More than one tuple can be emitted by a spout.

Emitted streams are consumed by *bolts.* They are responsible for processing streams and can do any computation or transformation, such as filtering, aggregations, joins, connecting to databases and more. Bolts take any number of streams as input and produce any number of streams as output making simple or more complex stream transformations. For more complex transformations, multiple bolts are required. Figure 9 presents an example of a Storm topology.

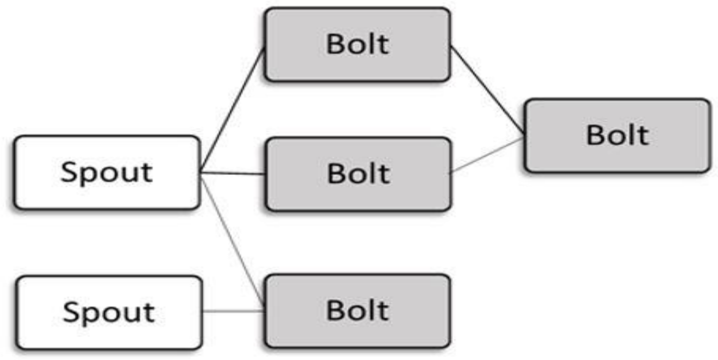


Figure 9: An example of Storm topology [24]

Each instance of a bolt or spout is called *task*. Tasks are inherently parallel, just like map and reduce tasks are inherently parallel in MapReduce. Tasks are spread among the different workers of a cluster

Part of the implementation of a Storm topology is specifying for each bolt which streams should be received as input. *Stream grouping* defines how the stream should be partitioned among the bolt's tasks. Although custom stream grouping can be implemented, there are eight built-in stream grouping techniques.

1. *Shuffle grouping*, which distributes tuples using a random round-robin algorithm, ensuring that each bolt get an equal number of tuples.
2. *Fields grouping*, which distributes tuples by fields specified in the grouping. For example, if the stream is grouped by the "user-id" field, tuples with the same "user-id" will always go to the same task, while tuples with different "user-id" may go to different tasks.
3. *Partial Key grouping*, which works like fields grouping with the difference that stream load is balanced between two downstream bolts, which provides better utilization of resources when the incoming data are skewed.
4. *All grouping*, which replicates the stream across all the bolt's tasks.
5. *Global grouping*, which sends the entire stream to the bolt's task with the lowest id.
6. *None grouping*, which indicates that the user does not care how the stream is grouped. Currently, none grouping is equivalent to shuffle grouping.

Eventually though, Storm will push down bolts with none grouping to execute in the same thread as the bolt or the spout they subscribed to.

7. *Direct grouping,* which lets the producer of the tuple to decide which task of the consumer will receive this tuple. Direct grouping can only be declared on streams that have been declared as direct streams.
8. *Local or shuffle grouping*. In this case, if the target bolt has one or more tasks in the same worker process, shuffles tuples to just those in-process tasks. Otherwise, it acts like a normal shuffle grouping.

By default, Apache Storm provides at-least-once delivery guarantee. To achieve this, it tracks the tree of tuples triggered by every spout tuple and verifies that tree of tuples has been successfully completed. If Storm fails to detect that a spout tuple has been completed within a timeout, which is associated with every topology, it replays the tuple later. This mechanism may falsely classify several records as non-acknowledged. Hence, these data will have to be reprocessed leading to low throughput. In some use cases, missing some records is not a problem, so Storm offers to the user the option to disable this fault tolerance mechanism. To provide higher throughput, micro-batch processing with exactly once delivery guarantee, *Trident* was developed on top of Storm infrastructure.

3.1.2. Apache Flink

Apache Flink [25] is a distributed, unified system for processing both streaming and batch data. Originally, it was developed under the project name “Stratosphere”, by researchers in Germany, but later was donated to Apache Software Foundation.

It was developed from the beginning as a distributed processing framework for stateful operations over unbounded and bounded data streams, by running two core APIs on the same distributed streaming execution:

- *DataSet API,* which is used for batch processing.
- *DataStream API*, which is used for event stream processing.

Using Flink, programs can be written from Java, Scala, Python to SQL. Applications can be deployed in local, cluster or cloud mode, providing flexibility to the developers [Figure 10]. It also supports built-in connectors to third-party systems for source and sinks, such as Apache Kafka, Apache Cassandra and Elasticsearch.

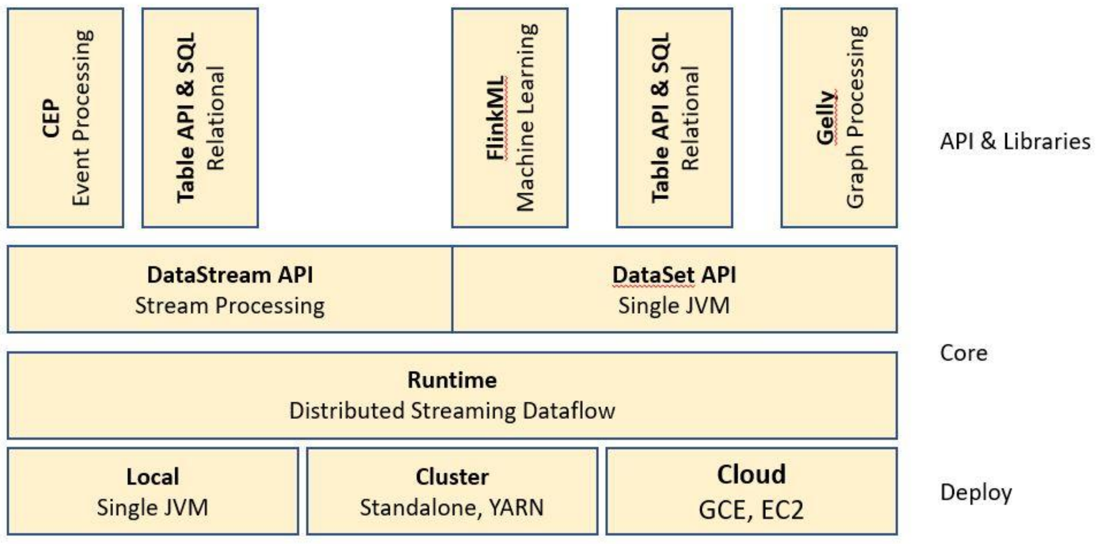


Figure 10: Flink's stack [26]

Flink implements the Dataflow programming approach. When executed, Flink programs are mapped to a DAG, consisting of streams (edges), and transformation operators (nodes). Each DAG starts with one or more source operator and results to one or more sink operator. Flink programs are executed, by their nature, in a parallel and distributed way. During execution a stream is divided into stream partitions and one or more operator subtasks. These subtasks are working independently and execute in different threads and if needed in different machines [Figure 11]. The number of each operator subtasks defines the parallelism of this specific operator.

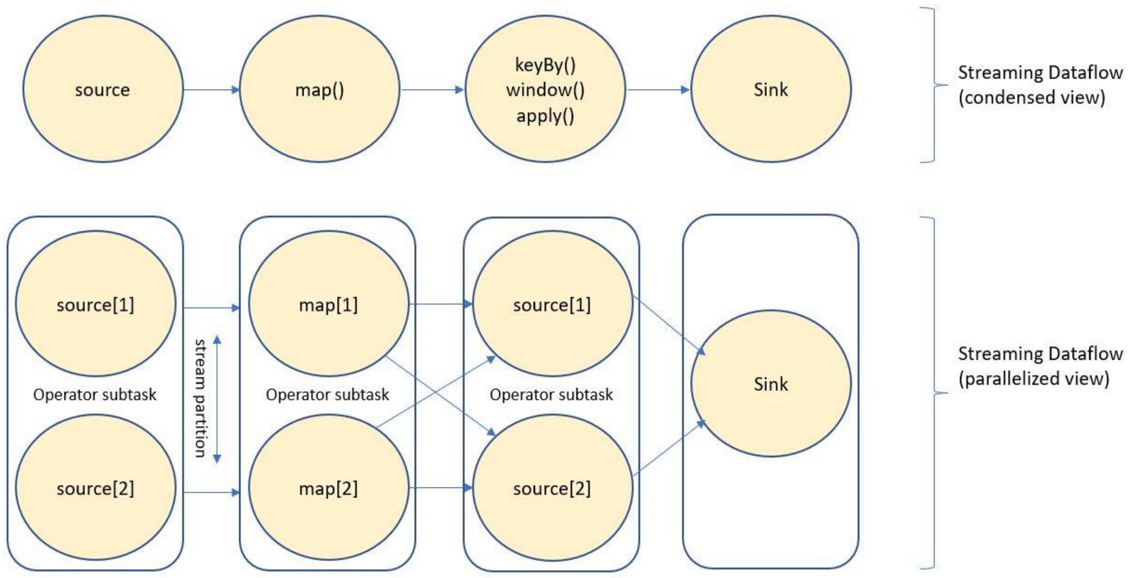


Figure 11: Streaming dataflow of Flink application [26]

As we mentioned before, time is one of the key concepts of any streaming application. Flink supports different notions of time to let developers define how events should be correlated [25]. *Event*, *ingestion*, and *processing* time notions are offered by Flink.

- Event time refers to the data production time, hence it is attached to an element before it enters Flink application.
- Ingestion time refers to the time that an event enters Flink application. Source operator is responsible to attach its current time as a timestamp attribute to the arriving event.
- Processing time refers to the time of the machine that executes the operation to that event.

One of the most common challenges in the field of stream processing engines is the ability to efficiently handle out-of-order data. To overcome this challenge, Flink introduces *watermarks*. In essence, watermarks indicate global progress measure. A watermark of time *t* indicates that all events with lower timestamp than *t* have already entered an operator.

Streaming applications are receiving and processing data continuously. Of course, we can process each incoming event as soon as it arrives, but there are some cases where an application needs to aggregate results of a bunch of events. *Windows* are the provided workaround of this problem. The simplest windows are those based on time. Computations are triggered when the ending time of the window is passed. Time windows are divided into *tumbling* and *sliding*.

- *Tumbling window*, where window overlapping is not allowed [Figure 12].

- *Sliding window*, where windows slide over data. Thus, windows can be overlapping. This technique usually leads to a smoother aggregation over the incoming events [Figure 13]

Another provided window by Flink is the *session* window. Sessions are periods of activity separated by frames of inactivity. In contrast to time windows, session windows do not have a fixed start and end time. When a specified time of inactivity passes then window is considered closed and aggregations are triggered. Apache Flink also supports *global windows,* which do not have a natural end time. In this type of window all elements with the same key are assigned to the same window, which never closes.

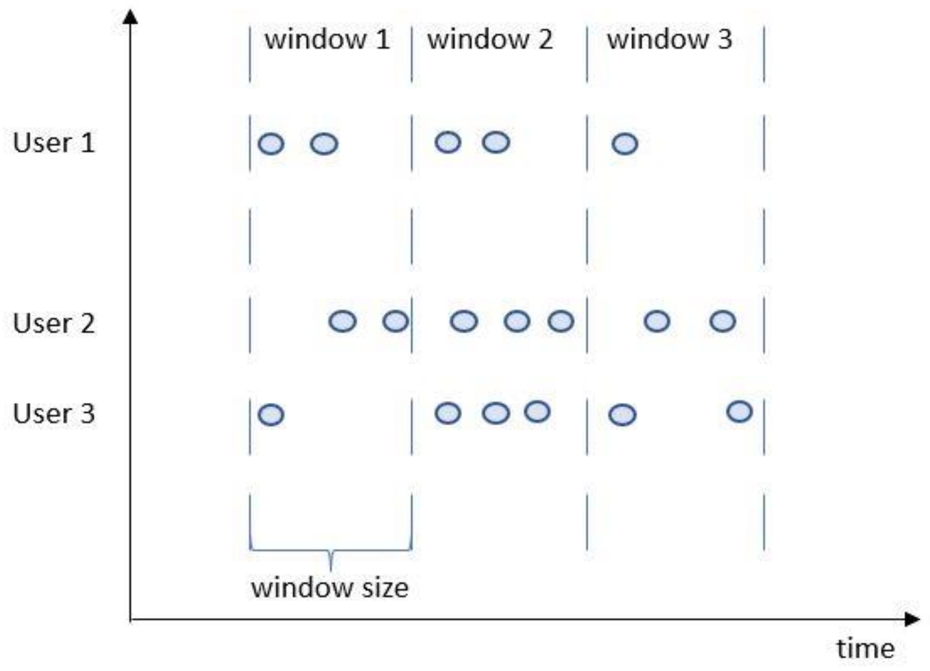


Figure 12: Tumbling window [26]

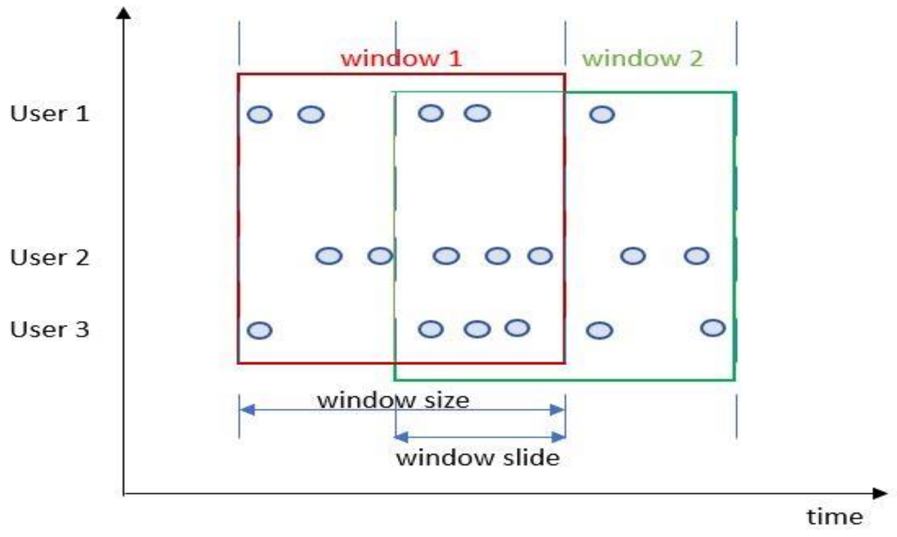


Figure 13: Sliding window [26]

Flink also provides an optional mechanism to define *triggers*. A trigger determines when a window is ready for data processing by the window function. Although each window comes in pair with a default trigger, user can define his own custom trigger. Flink provides different kind of triggers, like triggers based on event or processing time.

Triggers are used to invoke intermediate computations. In case of a global window, since it is never closes, the trigger function should be implemented for computations to be performed.

Flink supports stateful operations, meaning that it is possible to maintain the state of previous events. While many operations in a dataflow look at one individual event at a time, there are operations that need to remember information of multiple previous events.

In order to ensure fault tolerance, Flink uses *checkpoint* and *stream replay* mechanisms. The checkpoint is a global snapshot of the set of operators' states. In case of failure, Flink selects the latest stored checkpoint and recovers the entire dataflow giving each operator the state that had when the failure occurred.

With Flink it is possible to achieve any of the different levels of delivery guarantee (at most once, at least once and exactly once). As we mention above, exactly once is the ideal guaranteed level. This does not mean that every event is processed exactly once. Instead, it means that every event will affect the state being managed by Flink exactly once.

3.1.3. Apache Samza

Apache Samza [1] is an open-source framework for distributed processing of high-volume event streams. It has been developed by engineers at LinkedIn to provide scalable, stateful and fault tolerant data stream processing. Its design goal is also to achieve high throughput, and operational robustness.

Apache Samza was built based on the following foundational abstractions:

- Partitioned Data Processing Model: Processing model of *Samza* consists of streams and jobs. Stream is a collection of immutable, infinitive, and multi-subscriber sequence of *messages*, which can be replayed and are lossless by design. Each stream is internally divided into multiple partitions.

- Fault-tolerant Local State: To ensure fault tolerance, state of each task is stored on the local disk of the processing node. Also, to avoid loss of stored state on local disk, *Samza* stores an append-only log, named changelog, in Kafka topic.

- Cluster-based Task Scheduling: *Apache Samza* has not a built-in mechanism for task scheduling and cluster management. By contrast, it relies on existing cluster managers, and supports two modes of distributed operators:
  (1) *Hadoop Yarn* and (2) *stand-alone* mode using *Apache Zookeeper*

The design of these low-level abstractions is strictly connected with the scalability and operational robustness of *Apache Samza* [2].

The core abstraction of Samza is the message. Samza processes streams, which are collections of immutable messages, usually of the same category. A *job* in Samza performs the logical transformation on a set of input streams and produces messages, which append to a set of output streams.

For scalability to be achieved, streams are divided into smaller units of parallelism, named *partitions* [Figure 14], and jobs are chopped to *tasks* [Figure 15]. Partitions are ordered sequences of messages, with a unique identifier, named *offset*, attached on it. Offset can be of integer, byte or string type. When a message is appended to the stream, it is appended to only one of the stream's partitions. Each task consumes data from one partition for each of the job's input streams. To achieve independently work from each task, there is no defined ordering across partitions and each task reads and processes messages, from partitions that are assigned to it, by sequential order of message's offset. It is not possible the number of tasks to exceed the number of input partitions. The assignment of partitions to tasks is permanent. If a task is assigned to a machine and it fails, this task is restarted elsewhere, but still consuming the same stream partitions. To achieve this, Samza tracks if a message is successfully delivered or not utilizing a checkpoint system, but it can only deliver at least once guarantee.

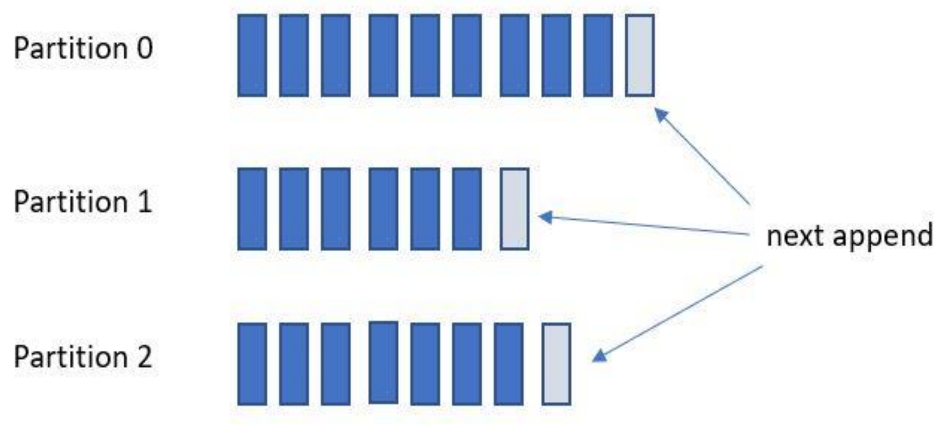


Figure 14: A partitioned stream in Samza [27]

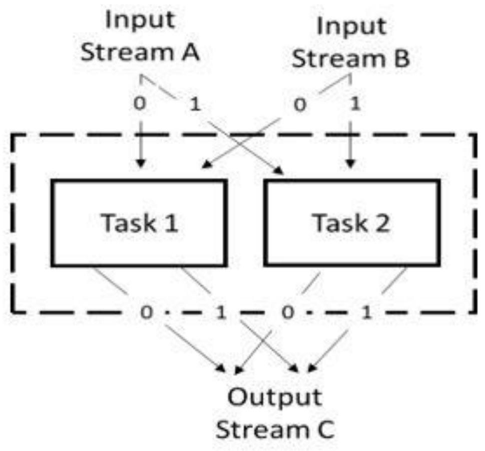


Figure 15: Distributed execution in Samza [27]

3.1.4. Kafka Streams

Apache Kafka [28] was first proposed by engineers at LinkedIn, but later was donated to Apache Software Foundation. It is a distributed and scalable messaging system which offers high throughput. It combines three key concepts to provide end-to-end event streaming:

- Publish and subscribe to streams of records
- Fault-tolerant durable storage of stream records
- Processing stream records as they occur or retrospectively

The core abstraction of Kafka is the *topic*. The topic is the storage of the *events*, which are records with key, value, and timestamp information. A topic is by default multi-producer and multi-subscriber, meaning that more than one *producer* can write events to a topic and more than one *subscriber* can read and process events from a topic. Also, producers can write to more than one topic and subscriber can read and process events from more than one topic.

Topics are divided into several *partitions*. These partitions can be located on more than one Kafka server, named *brokers*. Brokers are designed so they can operate in a cluster mode. The described architectural design is promoting scalability since it allows multiple producers and subscribers to write and read messages respectively at the same time.

Kafka provides five core APIs:

- Admin API, which is used to manage and inspect Kafka objects.
- Producer API, which is used to publish events records to one or more Kafka topics.
- Consumer API, which is used to read and process events records from one or more Kafka topics.
- Kafka Streams API, which is used to implement stream processing applications and microservices.
- Kafka Connect API, which is used to build and run data connectors to allow external systems and applications to integrate with Kafka

Our focus is given to *Kafka Streams API*. Kafka Stream can be used to build highly scalable, elastic, fault-tolerant, distributed applications and microservices [29].

The basic abstraction of Kafka Streams is the *stream,* which represents an infinite, continuously updating data set. To achieve scalability, parallelization and fault-tolerance, Kafka Streams supports running multiple instances of an application. These instances of the application can be deployed in different machines and automatically work together on the data processing. Every Kafka Stream application implements and executes at least one *topology* [Figure 16], which is a graph representation (DAG) of the computational logic of the application. Each topology consists of nodes, called *stream processors*, and edges, called *streams*. Stream processors are just operators, which receive one input at the time form the parent processor, apply data transformation, and produce one or more output records to its downstream processors. There are two special processors on the topology:

- *Source processor*, which is responsible for consuming records from one or more Kafka topics and passing them to its downstream processors.
- *Sink processor*, which is responsible for sending any received records from its parent processors to a desginated Kafka topic.

Kafka Streams provides two APIs to define stream processors:

1. *Declarative, functional DSL (Domain Specific Language)*, which helps the users to define a stream-processing application in just a few lines of code, by implementing a sequence of transformations to events in the stream. This API

provides not only *stateless*, but also *stateful* transformations. When stateful operators are used, it automatically creates and manages state storage. In order to work correctly, state stores ensure fault tolerance by default.

2. *Processor API*, which is a low-level, flexible, and powerful API. It allows users to define arbitrary stream processor that receives one record at a time and processes it, either in a stateless or in a stateful manner. When this API is used, developers compose the processor topology, implementing a customized processing logic. For stateful process to be achieved, one or more state storage must be provided manually. The user can choose from available state storage types, which have fault tolerance enabled by default, or implement their own custom storage type. The defined processor and associated state storage are connected for the processor topology to be composed.

Kafka Streams supports at least once and exactly once delivery guarantee.

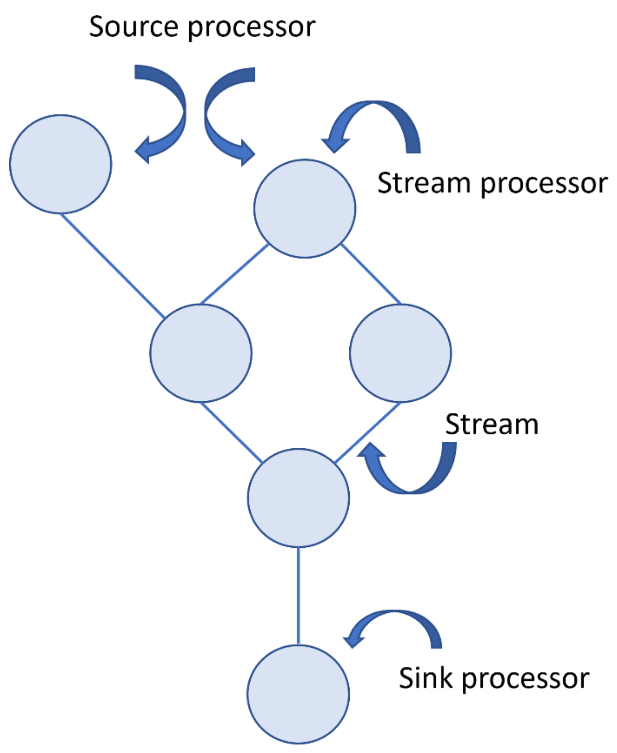


Figure 16: A Kafka Stream topology [29]

3.1.5. Spark Streaming

Apache Spark [30] was originally developed by researchers at the AMPLab of Berkley’s University and later was donated to Apache Software Foundation. It is a unified analytics engine for large-scale distributed data processing, which supports the following programming languages: Scala, Java, Python and R. Spark makes implementation and maintenance of data pipelines a lot easier since it provides high-level APIs (Application Programming Interface) including Spark SQL for SQL queries, MLib for machine learning, GraphX for graph processing and Spark Streaming for streaming processing [Figure 17]. It can be deployed as a standalone installation, but also on Hadoop Yarn, Mesos or Kubernetes.

The key abstraction of Apache Spark is the RDDs (Resilient Distributed Dataset), which are immutable, fault-tolerant data collections partitioned across nodes of a cluster that can be operated in parallel. The user can apply transformations to RDDs, such as map, filter, and groupBy.

To support stream processing, an extension of Spark core API was developed. *Spark Streaming* supports scalable, high throughput, and fault-tolerant processing of live data streams. It receives live data streams as input, divides the data into batches and then processes these batches to generate the output stream in the form of batches [Figure 18].

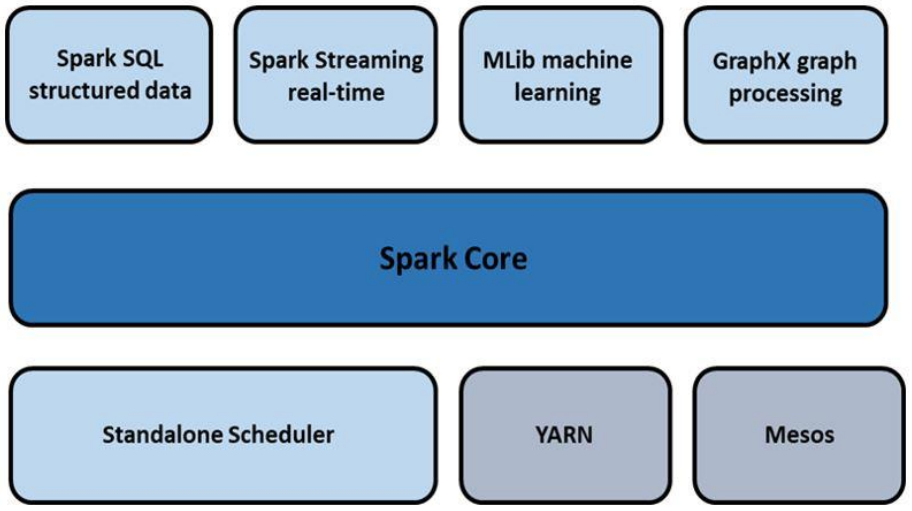


Figure 17: Spark Stack [31]

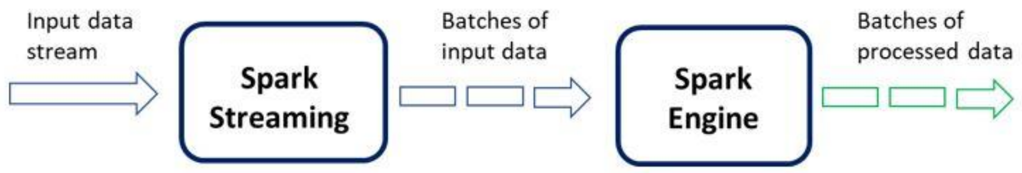


Figure 18: Spark Streaming processing approach [32]

The abstraction of Spark Streaming is called *discretized stream* (D-Stream), and is represented as a sequence of RDDs [34]. D-Streams can be generated from many sources, like Kafka, HDFS, databases or from other D-Stream output. Internally, D-Streams contain data of a specified, small time interval size [Figure 19]. D-Streams support both stateless operations, which apply independently in each time interval, like map operations, and stateful operations, like aggregation over a sliding window, which apply on multiple intervals. Spark Streaming provides two different types of operators over D-Streams: *transformation* and *output* operators.

- Transformation operators, which output a new D-Stream from one or more parent D-Streams.
- Output operators, which allow program to save data to external systems, such as HDFS.

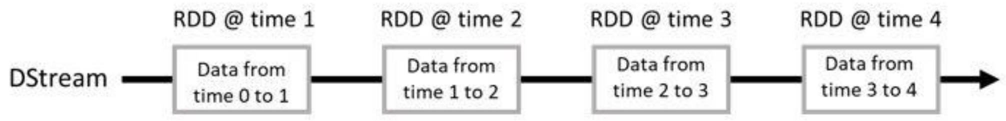


Figure 19: D-Stream: the abstraction of Spark Streaming [32]

Adopting the micro-batch methodology, Spark achieves high throughput, but also higher latency than native streaming engines, like Storm or Flink. Zaharia et al [34] mention that even though Spark Streaming achieves higher latency than true streaming approach, latency can be minimized as low as a second, which is acceptable for the most use cases, due to in memory computations of RDDs.

Spark Streaming also provides windowed computations, which allow user to perform calculations on a sliding window of data, by specifying two parameters: *window length*, which is the duration of window, and sliding interval, which identifies the interval in which the window operation is performed [Figure 20].

To achieve fault tolerance, Spark Streaming implements the approach of *parallel recovery*, by periodically checkpointing some of the state of RDDs. By utilizing parallel recovery, it provides faster and with lower computational cost recovery. Spark Streaming also guarantees that batch level processing will be executed in an exactly once mode, by tracking the lineages in each D-Stream.

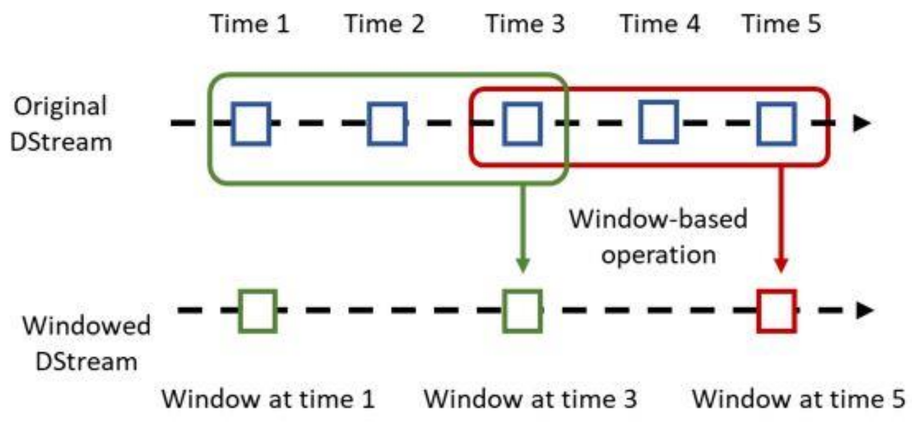


Figure 20: Illustration of window in Spark [32]

3.2. Comparison of Stream Processing Frameworks

Over the last few years, comparisons between different Data Stream Processing Systems have gained the attention of the scientific community. Not only researchers, but also business world is interested in the performance of these systems since more and more companies need to incorporate real-time processing tools into their data analysis pipeline. Many benchmarking experiments have been published, comparing the streaming engines that we discussed in the previous section. To shed some light on the performance of the aforementioned systems, we provide some results of experiments, which compare these stream processing frameworks.

Karimov et al. [6] measure the latency and throughput of Storm, Flink and Spark in production environment, by introducing a benchmarking framework. They conclude that if a stream contains skewed data, Spark is the best choice. On the other hand, Storm and Flink are equally robust to fluctuations in the data arrival rate in aggregation workloads, while Flink responds better on join queries. Flink is the best choice for use-cases that latency is the priority. Overall, Flink has better throughput both for aggregation and join queries.

Another very interesting experiment conducted by Perera et al. [7], compares Flink with Spark, using a deployment orchestration engine which is developed to automate reproducible experiments on both cloud and bare-metal environments. The Yahoo streaming benchmark was reproduced for the stream processing experiment. Both Spark and Flink performed similarly under different loads, but Flink outperformed Spark at lower event rates. CPU behavior was observed similar in both systems, but Flink tended to consume less memory while executing.

Chintapalli et al. [8] developed a streaming benchmark for Flink, Spark Streaming and Storm. In order to mimic real-world production scenarios, a full data-pipeline, using Kafka and Redis, was tested. According to the results, Flink and Storm have much lower latency than Spark Streaming at fairly high throughput, while Spark Streaming can handle higher throughput. Table 1 presents a briefly comparison between the most common stream processing frameworks.

	Apache Storm	Apache Flink	Apache Samza	Kafka Streams	Spark Streaming
Processing Model	Streaming	Hybrid	Streaming	Streaming	Micro-Batch
Streaming Abstraction	Tuple	DataStream	Message	K-Stream	D-Stream
Latency	Very Low	Very Low	Low	Low	Medium
Throughput	Low	High	High	Medium	High
Stateful Operations	No	Yes	Yes	Yes	Yes
Delivery Guarantee	At least once (Exactly once with Trident)	Exactly once	At least once	Exactly once	Exactly once
Fault tolerance	Yes	Yes	Yes	Yes	Yes
Programming language Support	Any	Java, Scala, Python	JVM	Java	Java, Scala, R, Python

Table 1: Comparison of Data Streaming Engines

Even though performance plays a significant role in the choice of the data stream processing framework, it is not the only factor, and we cannot completely rely on it. Real-time stream processing is rapidly evolving and so do the corresponding frameworks. Hence, we cannot conclude to a clear winning framework, but the choice should be based on the requirements of different applications.

3.3. Big Data Storage

Instead of effective real-time processing, effective storage of the output result should be taken into consideration. Under this scope, a database which is able to comply with the characteristics of Big Data should be used. Typical relational databases were not developed to address the agility and scalability requirements of modern applications [9]. The tendency to swift from structured to unstructured, schema-less data, the complexity of data generated by web resources, and the exponential growth of daily produced data have raised challenges for traditional data management systems [10].

NoSQL databases were developed to overcome the problems that relational databases could not resolve. They offer efficient storage, reduced operational costs, high scalability, availability, fault tolerance and consistency. Hence, NoSQL databases have become the default technology for storing big data.

In this section we will briefly introduce the different types of NoSQL databases, explain the differences and how they work.

3.3.1. NoSQL Databases

NoSQL databases can be categorized into *key-value stores*, *document databases*, *column-oriented databases*, and *graph databases.*

- Document databases store data in JavaScript Object Notation (JSON), Binary JavaScript Object Notation (BSON), or Extensible Markup Language (XML) documents. Each document can be accessed using a unique key as identifier. Also, this kind of database gives the capability to the user to execute queries to extract documents that satisfy the query requirements. This functionality differs document databases from key-value stores, since in key-value stores, values are black boxes, while document databases also keep metadata from the stored documents. *Documents* can be grouped together to a structure called *collection*. If documents are analogous to a row, then collection is analogous to a table in relational database world. One of their greater advantages is that they provide schema flexibility. Stored documents in such databases can be very similar but also completely different. This flexibility is very convenient for the developers because data are under the control of the developers, and there is no need for the structure to change. Elasticsearch and MongoDB are two examples of this type of database. Most common use cases of document databases include ecommerce and trading platforms.
- Graph databases store data as a *graph*, focusing on the relationships between them. Each graph consists of edges, which are the stored objects, and edges connecting them, which are called *links* or *relationships*. Data are stored without

predefined schema; hence graph databases are very flexible. Most graph databases are ACID compliant. Graph databases are optimized to search the connections between data elements. A well-known graph database is Neo4j. Most common use cases of graph databases include fraud detection and social networks.

- Key-value store is the simplest type of a NoSQL database. It implements a schema-less model, in which data are stored as a key-value pair. Strings, numbers, binaries, and other objects can be stored as values and accessed with the help of the corresponding key. Thus, key is used as a reference for the value. They are very efficient, providing low latency for queries. A well-known key-value store is Redis. Most common use cases of key-value databases include user preferences and shopping carts.
- Column-oriented databases are organized as a set of columns, while a relational database stores data as rows. Column-oriented databases are highly scalable and consistent. Moreover, aggregations can be performed efficiently because of the inherent structure of this database. Data read and retrieval in such systems can be executed quite fast. A well-known column-oriented database is Apache Cassandra. Most common use cases of a column-oriented database include content management systems and blogging platforms.

3.3.2. Elasticsearch

Elasticsearch [43] is an open-source, distributed search and analytics engine built on top of Apache Lunce.

In Elasticsearch, data are stored into one or more indices. *Index* is analogous to a table in the relational database world. Indexes are used to store documents. *Document* is the main entity in Elasticsearch, which consists of *fields* and *values*. Each field can contain one or more values. Documents are schema-less, meaning that they may have different set of fields. From user's point of view, a document is in JSON format and corresponds to a row in a relational database, while fields correspond to columns. Documents have unique IDs, which can be assigned to them either by the user or automatically by Elasticsearch.

Elasticsearch allows the user to store, search and analyze huge volume of data and return answers in near real-time latency. It can efficiently store, and index structured, semi-structured and unstructured data.

Elasticsearch's tight integration with Kibana – a free open-source frontend layer that effectively visualize data indexed in Elasticsearch – makes it ideal for use cases where dashboards of stored data should also be implemented.

4. Implementation

In this section we initially discuss the proposed architecture of the application that was developed as part of this thesis, the requirements and the challenges that we faced. Afterwards, we present how the desired statistics are calculated, by providing examples and the used algorithms. Finally, we explain how our application handles the delayed data.

4.1. Requirements and Challenges

In the most used implementation, the route report [Figure 1] of the vehicle, is calculated when requested by the user. Emitted data from vehicle's sensors are stored into a database and the calculations are triggered when the end user is asking for the report [Figure 21]. This implementation can lead to undesirable waiting times, especially if the amount of data is growing continuously.

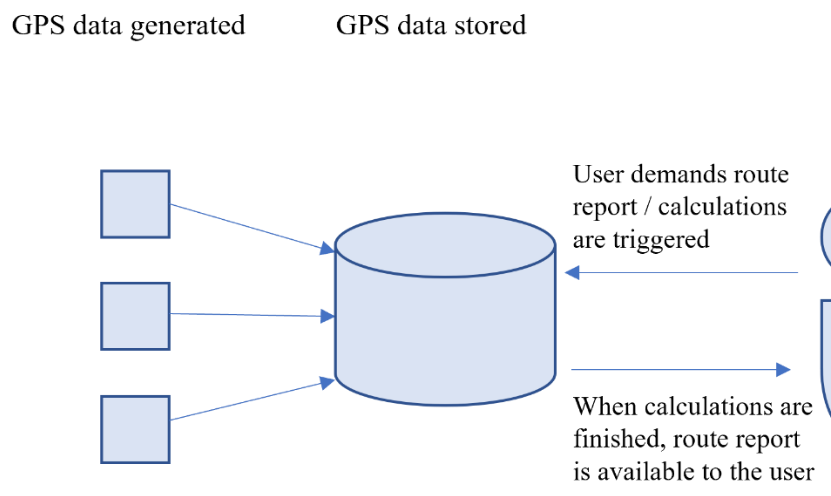

GPS data generated
GPS data stored
User demands route report / calculations are triggered
When calculations are finished, route report is available to the user

Figure 21: Most used architecture

One of the requirements that our system needs to address is the desired statistics to be calculated in real-time, and not when the user asks for the route report. The already calculated statistics of a given trip should be stored in an efficient way, so as when the report is requested by the user, no calculations, but only search of the desired information, will be triggered [Figure 22]. Another important requirement that our application needs to address is the correct handling of delayed data. Correct handling of this kind of data is of crucial importance since the lack of handling out-of-order data can lead to wrongful and meaningless results.

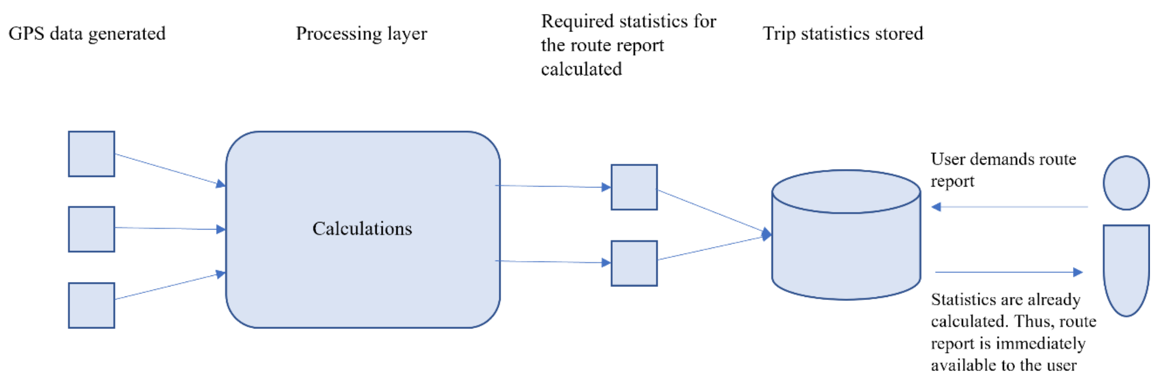

GPS data generated
Processing layer
Required statistics for the route report calculated
Trip statistics stored
Calculations
User demands route report
Statistics are already calculated. Thus, route report is immediately available to the user

Figure 22: Proposed architecture

Reliability of the calculated trip's statistics is one of the main challenges of the proposed system. As we mentioned above, the first step of the implementation, that is shown in Figure 21, is to store all the emitted data. Thus, when the user asks for the statistics of all the trips performed by a specific vehicle, all the corresponding data are already stored. This means that for the computed results, all the corresponding data are taken into consideration. With the implementation that is shown in Figure 22, there is a chance that the values of a given trip have been calculated, without taken into account all the corresponding data. To clarify the challenges that we faced, we introduce the following example.

Example 1

Let us assume that Table 2 presents the already processed GPS data.

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
1	7059	2021-12-02 10:00:00	motion	37.939013	23.648045	10.0
2	7059	2021-12-02 10:05:00	motion	37.939020	23.648050	15.0
3	7059	2021-12-02 10:10:00	motion	37.939025	23.648055	25.0
4						
5	7059	2021-12-02 10:15:00	motion	37.939030	23.648060	12.0
6	7059	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0
7	7059	2021-12-02 10:25:00	parked	37.939040	23.648070	0.0

Table 2: GPS data that arrived at the system as an example of the challenges that our application needs to face

As we have mentioned in Section 1.1., trip duration is one of the values that should be presented in the route report. Since the architecture that is proposed in this thesis is the one shown in Figure 22, statistics of the trip are already computed and available to the user. Thus, by this time, some of the provided information is that vehicle 7059 has performed one trip and the duration of this trip was 25 minutes.

Let us now assume that the GPS record that is shown in the following table is the GPS record that just arrived at the application. This delayed GPS record should be the fourth data of the trip. As soon as the following GPS record arrives at the system, the already calculated values should be corrected.

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
4	7059	2021-12-02 10:12:00	parked	37.939027	23.648057	0.0

Table 3: Delayed GPS record that arrived at the system as an example of the challenges that our application needs to face

The information that should be presented in the route report is that two trips have been performed. The duration of the first trip is 12 minutes and the duration of the second trip is 10 minutes. These recalculations that should be performed in case of out-of-order GPS data is the most challenging part of our proposed application.

4.2. System Architecture

In this chapter we briefly discuss the proposed architecture of our application. Emitted GPS data are stored into a topic of Apache Kafka. Then, Apache Flink consumes these GPS data, it computes the required values for a given trip and passes these values to Elasticsearch, which is the storage layer of our implementation. We choose to integrate Apache Flink with Apache Kafka, since it is a way to overcome low throughput issues that may occur due to backpressure. Moreover, Elasticsearch is used because of its ability to perform extremely fast searches, while it is also scalable and schema free. The final step of our proposed implementation is the data visualization. Kibana, which is built on top of Elasticsearch, provides a fast and efficient way to visualize the stored data.

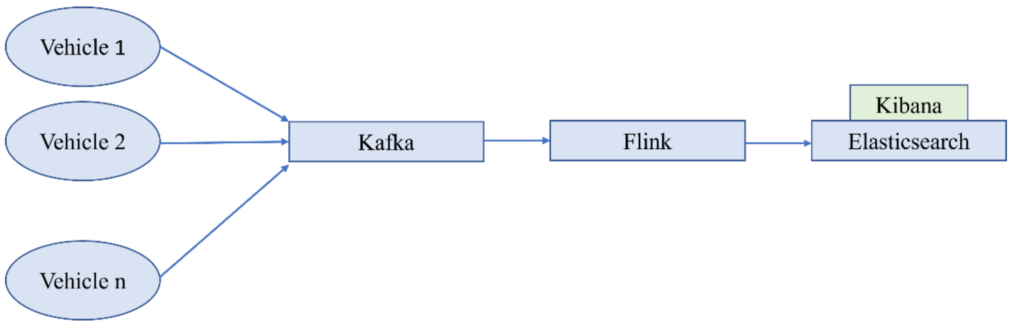


Figure 23: Proposed architecture and tools

4.3. Solution Approach

As we mentioned above our implementation is based on Flink's DataStream API. We designed and developed a real-time application, in which Apache Flink continuously reads data from an Apache Kafka topic, processes the data in order to compute the desired values, and then loads these values to Elasticsearch.

First, Flink reads the emitted data, which have been stored into a Kafka topic, and model them to a GPSEvent class. GPSEvent is a custom object, which is designed to have the fields that are presented in the following table.

GPSEvent Object Fields

String	**vehicleID**
long	**date**
Enum	**EngineStatus**
double	**lon**
double	**lat**
double	**speed**

Table 4: Fields of GPSEvent class

To achieve this in the first part of the designed pipeline, we continuously use a custom map function, which takes as input a comma separated String value, that is saved into a Kafka topic, and transforms it to a GPSEvent object with defined fields. Thus, we create a DataStream of GPSEvent objects. Each GPSEvent object of the DataStream has assigned information of vehicleID, date, EngineStatus, longitude, latitude, and speed. All these values are essential for computing the statistics of each vehicle's trip.

The GPSEvents may arrive in an out of order way for a variety of reasons that we have explained in *Section 1.1*. In order to handle the delayed events, we need to assign timestamps and watermarks in the DataStream, using the AscendingTimestampExtractor function that is provided by Apache Flink and defining as event time the date field of the GPSEvent object.

The next part of the designed pipeline is to compute the desired statistics. Since the statistics of a given trip should be computed for each vehicle separately, we need to logically separate the GPSEvent objects before we proceed with the calculations. To achieve this, we use the keyBy operator of Flink's DataStream API, which partitions the DataStream into disjoint partitions, based on the vehicleID field of GPSEvent object.

After partition of the DataStream is performed, we use a process operator, which is responsible for the core computations of our application. The process operator calls TripFunction, a custom function, which extends KeyedProcessFunction of Flink's DataStream API. By extending KeyedProcessFunction, our implemented function gains access to the basic building blocks of a streaming application, namely events, state, and timers. This function is invoked for each GPS data received. It first checks whether the GPS record is delayed, or not and then it computes the statistics of each trip and creates or updates the values of the Trip object. To update the values of the Trip object, we need to store the values in memory. For this purpose, we use a HashMap data structure with vehicleID as keys and list of Trip objects as values. When we need to update the values of a given vehicle's trip, we retrieve the list of the trips of this specific vehicle and then we retrieve from this list the Trip object, whose values need to be updated. The logic that was used for the calculation of each value will be presented in detail in the following sections. After the processing of each GPS record and the calculation of the fields of the trip object, Elasticsearch is accessed, for the trip object to be stored. For each generated trip, a new record is stored into Elasticsearch.

Trip is a custom implemented class, designed to have the following fields.

Trip Object Fields

int **_Id**	*double* **_motionDuration**
String **VehicleId**	*double* **_idleDuration**
Enum **EngineStatus**	*double* **_parkedDuration**
long **starting_ts**	*double* **_avgSpeed**
double **starting_lat**	*int* **_cntSpeedViolations**
double **starting_lon**	*int* ***TripCount***
long **ending_ts**	
double **ending_lat**	
double **ending_lon**	
Enum **TripStatus**	
long **LatestTimestamp**	
ArrayList <GPSEvent> **Data**	
double **_tripDuration**	
double **_maxSpeed**	

Table 5: Fields of Trip class

The following table presents a brief explanation of how each of the Trip object's fields is used.

Field of Trip Object	***Usage***
int **_Id**	A unique id of the trip.
String **VehicleId**	The id of the vehicle that performed the trip.
Enum **EngineStatus**	The engine status (motion, idling or parked) of the last data of this trip.
long **starting_ts**	The starting timestamp of the trip.
double **starting_lat**	The starting latitude of the trip.
double **starting_lon**	The starting longitude of the trip.
long **ending_ts**	The ending timestamp of the trip.
double **ending_lat**	The ending latitude of the trip.
double **ending_lon**	The ending longitude of the trip.
Enum **TripStatus**	The status of the trip (ongoing or completed)
long **LatestTimestamp**	The timestamp of the last data of the trip.
ArrayList <GPSEvent> **Data**	An ArrayList that stores all the GPS data that are related to this trip.
double **_tripDuration**	The duration of the trip.
double **_maxSpeed**	The max speed of the trip.
double **_motionDuration**	The time that the vehicle was in motion during the trip.
double **_idleDuration**	The time that the vehicle was idle during this trip.
double **_parkedDuration**	The time that the vehicle was parked during this trip.
double **_avgSpeed**	The average speed of this trip.
int **_cntSpeedViolations**	Number of speed violations occurred.
int ***TripCount***	Number of the trip.

Table 6: Usage of each field of the trip object

4.4. Calculations of the required values

In this chapter we present the logic that we follow to calculate of each desired value. Table 7 presents the adopted computation rules for each value that needs to be calculated per trip. In the following chapters we are going to discuss in detail all these rules.

VALUE	RULE
Starting time	We define as starting time of a trip the timestamp of a GPS event of status “motion” or “idling” which is the first GPS event of a given vehicle, or it chronologically follows a GPS event of status “parked”.
Starting location	We define as starting location of a trip the location of a GPS event of status “motion” or “idling” which is the first GPS event of a given vehicle, or it chronologically follows a GPS event of status “parked”.
Ending time	We define as ending time of a trip the timestamp of a GPS event of status “parked”, which follows a GPS event of status “motion” or “idling”.
Ending location	We define as ending location of a trip the location of a GPS event of status “parked”, which follows a GPS event of status “motion” or “idling”.
Trip duration	We define as trip duration the time that the vehicle spent in order to move from the starting location to the ending location.
Motion duration	We define as motion duration of a trip, the time that the vehicle spent being in motion during a given trip.
Idle duration	We define as idle duration of a trip, the time that the vehicle spent being idle during a given trip.
Parked duration	We define as parked duration of a given trip the time passed between the end of this trip and the start of a new trip. If a new trip has not yet started, parked duration is defined as the difference between the timestamp of the last emitted GPS event of status “parked” and the first emitted GPS event of status “parked” for a given trip.
Average speed	To compute the average speed, we take into consideration only the speed of emitted GPS data of status “motion” and “idling”.
Speed violations	We define as speed violation the surpass of a predefined speed threshold. Consecutive GPS events with speed greater than the predefined speed threshold are considered as one occurred speed violation. For a new speed violation to be counted, the speed of the vehicle for a given trip should be dropped below the speed threshold and surpass it again.

Table 7: Adopted rules for each value of a given trip

4.4.1. Computation of Starting Time and Location

To compute the starting time and location of the trip we need to identify the timestamp and location of the first record (GPSEvent) that is of “motion” or “idling” engine status, and its previous engine status was “parked”, which means that the previous trip of the vehicle was finished, or its previous engine status was “NA”, which means that this trip is the first trip of the vehicle. Thus, to be able to identify the desired information, we need to check if the vehicle has already performed at least one trip. If it has already performed at least one trip, we need to identify if its last trip was completed, or if it is still on going. Algorithm of this implementation is presented right below.

Algorithm 1: Compute Starting Time and Location of a New Trip

```
    Input: H, HashMap with VehicleID as key and List of Trip Objects as values
           gpsEvent, the GPSEvent object arrived at the system

    Output: Starting time and location of a new trip
1   containsVehicleID ← H.containsKey (gpsEvent.vehicleID)
2   if containsVehicleID = true then
3       Identify the latestTrip
4   end if
5   if containsVehicleID = false or (latestTrip.TripStatus = finished and
    gpsEvent.EngineStatus = parked) then
6       Initialize a new trip object with trip.starting_ts = gpsEvent.date
                                          trip.starting_lon = gpsEvent.lon
                                          trip.starting_lat = gpsEvent.lat
7   end if
```

Algorithm 1: Compute starting time and location of a new trip

Following we introduce some examples to clarify this implementation.

Example 2:

First, let us assume that the vehicle with ID 7059 has not performed any previous trip, so the presented record in Table 8 is the first GPS record that arrives at the system. Since no previous trips were performed by this vehicle, the Trip object’s fields contain the values that are shown in Table 9. Right after the first GPS record has arrived, the required computations are triggered. Table 10 presents the Trip object’s fields after the finish of the computations. Table 9 and 10 display only the required fields for this example and not all the fields that were introduced in Table 5.

	vehicleID	Date	EngineStatus	lat	lon	speed
1	7059	2021-12-02 10:00:00	motion	37.939013	23.648045	10.0

Table 8: First GPS record that arrived at the system as an example on how starting time and location of the first trip of a vehicle is computed

_Id	VehicleID	EngineStatus	starting_ts	starting_lat	starting_lon
0	null	NA	0	0.0	0.0

Table 9: Trip object's fields before the arrival of the first GPS record

_Id	VehicleID	EngineStatus	starting_ts	starting_lat	starting_lon
1	7059	motion	1638482400	37.939013	23.648045

Table 10: Trip object's fields after the arrival of the GPS record

Example 3:

For this example, we assume that the vehicle with ID 7059 has already performed at least one trip. Table 11 presents the GPS data that has arrived at the system. The first six rows represent data that have already arrived and processed by the application, while the seventh row is the GPS emitted record that has just arrived at the system and it has not yet been processed by the application. Table 12 shows the state of Trip object before the processing of the last arrived record, while Table 13 presents the state of Trip object after the processing of the last arrived record. Table 12 and 13 displays only the required fields for this example and not all the fields that were introduced in Table 5.

	vehicleID	Date	EngineStatus	lat	lon	speed
1	7059	2021-12-02 10:00:00	motion	37.939013	23.648045	10.0
2	7059	2021-12-02 10:05:00	motion	37.939020	23.648050	15.0
3	7059	2021-12-02 10:10:00	motion	37.939025	23.648055	25.0
4	7059	2021-12-02 10:15:00	motion	37.939030	23.648060	12.0
5	7059	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0
6	7059	2021-12-02 10:25:00	parked	37.939040	23.648070	0.0
7	7059	2021-12-02 10:30:00	idling	37.939045	23.648075	0.0

Table 11: GPS data arrived at the system as an example on how starting time and location of a trip is computed, if at least one trip has already performed

_Id	VehicleID	EngineStatus	starting_ts	starting_lat	starting_lon
1	7059	parked	1638482400	37.939013	23.648045

Table 12: Trip object's fields before the arrival of the seventh GPS record

_Id	VehicleID	EngineStatus	starting_ts	starting_lat	starting_lon
2	7059	idling	1638484200	37.939045	23.648075

Table 13: Trip object's fields after the arrival and the processing of the last GPS record

As shown in the second example of this section, when the previous engine status is "parked" and the currently received GPS record is "idling" or "motion", a new Trip object is generated. This Trip object has starting_ts, starting_lat and starting_lon fields initialized with the respectively values of fields date, lat and long of GPSEvent object, which is the first observed GPS data of this trip.

4.4.2. Computation of Trip Duration

Another critical part of our application is the identification of the duration of each trip performed by a specific vehicle. To compute this value, we have taken into consideration two different scenarios. First, for the ongoing trips, since we have already stored the starting time of the trip, we compute the difference between currently received GPS data timestamp and starting timestamp of the corresponding trip. For the completed trips, we compute the difference between ending and starting timestamp of the trip.

Algorithm 2: Compute the Duration of a Trip

```
    Input: gpsEvent, the GPSEvent object arrived at the system
    Output: _tripDuration, the duration of a given Trip
1   if Trip.ending_ts = 0 then // trip is ongoing
2       _tripDuration ← gpsEvent.date – Trip.starting_ts
3   else // trip is completed
4       _tripDuration ← Trip.ending_ts – Trip.starting_ts
5   end if
```

Algorithm 2: Compute the duration of a trip

In the following we present some examples of this implementation.

Example 4:

First, we assume that the trip of vehicle 3030 has not yet completed. Every time a new GPS record arrives at the system, the implemented method for computation of the trip duration is being triggered. Table 14 displays a sequence of emitted GPS data for a specific trip of vehicle 3030.

	vehicleID	Date	EngineStatus	lat	lon	speed
1	3030	2021-12-02 10:00:00	motion	37.939013	23.648045	10.0
2	3030	2021-12-02 10:05:00	motion	37.939020	23.648050	15.0
3	3030	2021-12-02 10:10:00	motion	37.939025	23.648055	25.0
4	3030	2021-12-02 10:15:00	motion	37.939030	23.648060	12.0
5	3030	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0

Table 14: GPS data arrived at the system as an example on how trip duration of an ongoing trip is computed

Given that the received and already processed GPS data are these of Table 14, the Trip object with the required fields for this example is shown in Table 15.

_Id	VehicleID	EngineStatus	starting_ts	ending_ts	_tripDuration
1	3030	idling	1638482400	0	1200 (seconds)

Table 15: Trip duration of an ongoing trip

Example 5:

For this example, we assume that a specific trip of vehicle 3030 has been completed. The received data for this example are shown in Table 16.

	vehicleID	Date	EngineStatus	lat	lon	speed
1	3030	2021-12-02 10:00:00	motion	37.939013	23.648045	10.0
2	3030	2021-12-02 10:05:00	motion	37.939020	23.648050	15.0
3	3030	2021-12-02 10:10:00	motion	37.939025	23.648055	25.0
4	3030	2021-12-02 10:15:00	motion	37.939030	23.648060	12.0
5	3030	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0
6	3030	2021-12-02 10:25:00	parked	37.939040	23.648070	0.0

Table 16: GPS data arrived at the system as an example on how trip duration of a completed trip is computed

Given that the received and already processed GPS data are these of Table 16, the Trip object with the required fields for this example is shown in Table 17.

_Id	VehicleID	EngineStatus	starting_ts	ending_ts	_tripDuration
1	3030	idling	1638482400	1638433500000	1500 (seconds)

Table 17: Trip object with defined _tripDuration field for completed trip

The reason why we do not compute the trip duration of a completed trip by computing the difference between currently received GPS data timestamp and starting timestamp of the trip, can be understood if we assume that the received GPS data are these that are presented in Table 18. In this sequence, application receives two consecutive “parked” GPS data. Since the current GPS data timestamp is the one shown in the last row, if we have followed the same computation logic as we did for the ongoing trips, the resulting trip duration would be 35 minutes. The definition of the trip duration is the time a vehicle spent to move from the starting location to the ending location. As we have already mentioned, we assume that the ending location of the trip is representing by the geographic coordinates of the first "parked" record of the trip. Thus, the results would be wrong since the correct trip duration in this example is 25 minutes [Table 19]. To resolve this issue, we decided to compute the trip duration of completed trips using the following equation:

trip duration = ending timestamp – starting timestamp.

	vehicleID	Date	EngineStatus	lat	lon	speed
1	3030	2021-12-02 10:00:00	motion	37.939013	23.648045	10.0
2	3030	2021-12-02 10:05:00	motion	37.939020	23.648050	15.0
3	3030	2021-12-02 10:10:00	motion	37.939025	23.648055	25.0
4	3030	2021-12-02 10:15:00	motion	37.939030	23.648060	12.0
5	3030	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0
6	3030	2021-12-02 10:25:00	parked	37.939040	23.648070	0.0
7	3030	2021-12-02 10:35:00	parked	37.939040	23.648070	0.0

Table 18: GPS data arrived at the system as an example on how trip duration is computed when more than one data with "parked" engine status have arrived at the system

_Id	VehicleID	EngineStatus	starting_ts	ending_ts	_tripDuration
1	3030	idling	1638482400	1638433500000	1500 (seconds)

Table 19: Trip duration field for a completed trip with more than one "parked" GPS data emitted

4.4.3. Computation of Motion, Idle & Parked Duration

In this section, we present how the duration that the vehicle spent, being in “idling”, “motion” or “parked” state for a given trip is calculated.

To compute the duration that a vehicle was in “idling” state the following algorithm is designed.

Algorithm 3: Compute the Idle Duration of a given Trip

```
Input: gpsEvent, the GPSEvent object arrived at the system
Output: _idleDuration, the duration that the vehicle spent being on “idling” engine
status during a given trip.
1   doCalculation ← false
2   if Trip.EngineStatus ≠ “idling” and gpsEvent = “idling” then
3       _idleStartingTimestamp ← gpsEvent.date
4   else if Trip.EngineStatus = “idling” then
5       _idleCurrentTimestamp ← gpsEvent.date
6       doCalculation ← true
7   end if
8   if doCalculations = true then
9       _idleDuration ← _idleDuration + (_idleCurrentTimestamp - _idleStartingTimestamp)
10      _idleStartingTimestamp ← gpsEvent.date
11  end if
```

Algorithm 3: Compute the idle duration of a given trip

Algorithms 4 and 5 are used to respectively calculate the “motion” duration and “parked” duration of a given trip.

Algorithm 4: Compute the Motion Duration of a given Trip

```
    Input: gpsEvent, the GPSEvent object arrived at the system
    Output: _motionDuration, the duration that the vehicle spent being on "motion" engine status during a given trip.
1   doCalculation ← false
2   if Trip.EngineStatus ≠ "motion" and gpsEvent = "motion" then
3       _motionStartingTimestamp ← gpsEvent.date
4   else if Trip.EngineStatus = "motion" then
5       _motionCurrentTimestamp ← gpsEvent.date
6       doCalculation ← true
7   end if
8   if doCalculations = true then
9       _motionDuration ← _motionDuration + (_motionCurrentTimestamp -_motionStartingTimestamp)
10      _motionStartingTimestamp ← gpsEvent.date
11  end if
```

Algorithm 4: Compute the motion duration of a given trip

Algorithm 5: Compute the Parked Duration of a given Trip

```
Input: gpsEvent, the GPSEvent object arrived at the system
Output: _parkedDuration, the duration that the vehicle spent being on "parked" engine status during a given trip.
1  doCalculation ← false
2  if Trip.EngineStatus ≠ "parked" and gpsEvent = "parked" then
3      _parkedStartingTimestamp ← gpsEvent.date
4  else if Trip.EngineStatus = "parked" then
5      _parkedCurrentTimestamp ← gpsEvent.date
6      doCalculation ← true
7  end if
8  if doCalculations = true then
9      _parkedDuration ← _parkedDuration + (_parkedCurrentTimestamp - _parkedStartingTimestamp)
10     _parkedStartingTimestamp ← gpsEvent.date
11 end if
```

Algorithm 5: Compute the parked duration of a given trip

To clarify our implementation, we introduce the following example.

Example 6:

Let us assume that the emitted GPS data are shown in Table 20. Then, according to the algorithms that we presented above, "idling" duration should be equal to 25 minutes, "motion" duration should be equal to 8 minutes and "parked" duration should be equal to 2 minutes.

	vehicleID	Date	EngineStatus	lat	lon	speed
1	3030	2021-12-02 10:00:00	parked	37.939013	23.648045	0.0
2	3030	2021-12-02 10:05:00	idling	37.939013	23.648045	0.0
3	3030	2021-12-02 10:10:00	idling	37.939013	23.648045	0.0
4	3030	2021-12-02 10:15:00	idling	37.939013	23.648045	0.0
5	3030	2021-12-02 10:20:00	idling	37.939013	23.648045	0.0
6	3030	2021-12-02 10:30:00	motion	37.939018	23.648050	40.0
7	3030	2021-12-02 10:32:00	motion	37.939018	23.648050	50.0
8	3030	2021-12-02 10:38:00	parked	37.939018	23.648050	0.0
9	3030	2021-12-02 10:40:00	parked	37.939018	23.648050	0.0

Table 20: GPS data arrived at the system as an example on how we compute idle duration of a trip

The duration that a vehicle spent being in “motion” or “parked” state is calculated in a similar way.

4.4.4. Computation of Average Speed

To compute the average speed of a given trip, we need to store the sum of the speed for all the emitted GPS data that is in the “idling” or “motion” engine state, and the count of that data. Bellow we present the pseudocode and an example of this implementation.

Algorithm 6: Compute the Average Speed of a given Trip

```
     Input: gpsEvent, the GPSEvent object arrived at the system
     Output: _avgSpeed, the average speed of a given trip
1    if gpsEvent.EngineStatus ≠ “parked” then
2        _nonParkedNumData ← _nonParkedNumData + 1
3        _sumOfSpeed ← _sumOfSpeed + gpsEvent.speed
4    end if
11   _avgSpeed ← _sumOfSpeed / _nonParkedNumData
```

Algorithm 6: Compute the average speed of a given trip

Example 7:

Table 21 presents the received GPS data for a specific trip of vehicle 3030. To calculate the average speed of the trip, we need to store the sum of the speed and the number of non “parked” records. In this example, sum of speed is equal to 72.0 km/h and non “parked” records are 5. Thus, average speed in this case should be equal to 12.4 km/h.

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
1	3030	2021-12-02 10:00:00	motion	37.939013	23.648045	10.0
2	3030	2021-12-02 10:05:00	motion	37.939020	23.648050	15.0
3	3030	2021-12-02 10:10:00	motion	37.939025	23.648055	25.0
4	3030	2021-12-02 10:15:00	motion	37.939030	23.648060	12.0
5	3030	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0
6	3030	2021-12-02 10:25:00	parked	37.939040	23.648070	0.0
7	3030	2021-12-02 10:35:00	parked	37.939040	23.648070	0.0

Table 21: GPS data arrived at the system as an example on how we calculate the average speed of a trip

4.4.5. Computation of Maximum Speed

To identify maximum speed of each trip, we check if the current speed value is greater than all the previous speed values of this vehicle for a given trip. To achieve this, we store the maximum speed value in a variable, named _maxSpeed, and we compare with this variable every GPS data, corresponding to this trip, that arrives at the system. If a GPS record has greater speed value, then _maxSpeed variable should be updated. Below we present the pseudocode of this implementation.

Algorithm 7: Compute the Maximum Speed of a given Trip

```
   Input: gpsEvent, the GPSEvent object arrived at the system
   Output: _maxSpeed, the average speed of a given trip
1  if gpsEvent.speed > _maxSpeed then
2      _maxSpeed ← gpsEvent.speed
3  end if
```

Algorithm 7: Compute of the maximum speed of a given trip

4.4.6. Computation of Speed Violations

The amount of speed violations performed by a driver is also considered as a very significant and meaningful measurement. In our implementation, when speed is greater than a predefined threshold and the speed of the exact previous GPS record is lower than the predefined threshold, then we assume that a speed violation has occurred. Below we present the pseudocode and examples of this implementation.

Algorithm 8: Compute the Speed Violations of a given trip

```
   Input: gpsEvent, the GPSEvent object arrived at the system
          t, the predefined speed threshold
   Output: _cntSpeedViolations, the number of occurred speed violations
1  if gpsEvent.speed > t and _speedViolated = false then
2      _cntSpeedViolations ← _cntSpeedViolations + 1
3      _speedViolated ← true
4  else if gpsEvent.speed ≤ t then
5      _speedViolated ← false
6  end if
```

Algorithm 8: Compute of the speed violations of a given trip

Example 8:

For this example, we assume that Table 22 displays the received data for a trip of the vehicle 8581 and that the predefined speed threshold is 60 km/h. This is a global value which applies on every vehicle, and it is an assumption for this example. According to the designed algorithm, when the record of the third row arrives at the system, a speed violation is counted. The record of the fourth row has also speed greater than the predefined speed threshold, but since speed violation has occurred in the exact previous record, we assume that the amount of speed violations remains unchangeable.

	vehicleID	Date	EngineStatus	lat	lon	speed
1	8581	2021-12-02 10:00:00	motion	37.939013	23.648045	10.0
2	8581	2021-12-02 10:05:00	motion	37.939020	23.648050	55.0
3	8581	2021-12-02 10:10:00	motion	37.939025	23.648055	65.0
4	8581	2021-12-02 10:15:00	motion	37.939030	23.648060	70.0
5	8581	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0

Table 22: GPS data arrived at the system as an example on how we compute the speed violations of a given trip

Example 9:

For this example, we assume that Table 23 presents the received data for a trip of the vehicle 8581 and that the predefined speed threshold is 60 km/h. After the processing of the data, we assume that three speed violations have occurred. The first speed violation should be counted when the GPS record of the third row arrives at the system, the second speed violation should be counted when the GPS record of the eighth row arrives at the system and the third speed violation should be counted when the GPS record of the last row arrives at the system.

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
1	8581	2021-12-02 10:00:00	motion	37.939013	23.648045	10.0
2	8581	2021-12-02 10:05:00	motion	37.939020	23.648050	55.0
3	8581	2021-12-02 10:10:00	motion	37.939025	23.648055	65.0
4	8581	2021-12-02 10:15:00	motion	37.939030	23.648060	70.0
5	8581	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0
6	8581	2021-12-02 10:25:00	idling	37.939035	23.648065	0.0
7	8581	2021-12-02 10:30:00	motion	37.939040	23.648070	55.0
8	8581	2021-12-02 10:40:00	motion	37.939045	23.648075	65.0
9	8581	2021-12-02 10:45:00	motion	37.939050	23.648080	70.0
10	8581	2021-12-02 10:50:00	motion	37.939060	23.648085	45.0
11	8581	2021-12-02 10:55:00	motion	37.939065	23.648090	80.0

Table 23: GPS data arrived at the system as a second example on how we compute the speed violation of a given trip

4.4.7. Computation of Ending Time and Location

A very important information that a trip route report should include is the destination and the time that the vehicle arrives at this destination. To correctly identify when and where a trip has been completed, we should maintain the timestamp, longitude, and latitude values of the first emitted GPS data with “parked” engine status value. It is possible that for a specific trip, more than one GPS record with “parked” engine status arrive at the system. In this case, we assume that the ending time and location of the trip are the corresponding values of the first emitted “parked” GPS record. To clarify our implementation, we introduce the following algorithm.

Algorithm 9: Compute the ending Time and Location of a given Trip

```
    Input: gpsEvent, the GPSEvent object arrived at the system
    Output: _maxSpeed, the average speed of a given trip
1   if gpsEvent.EngineStatus = “parked” and trip.TripStatus = “ongoing” then
2       Trip.ending_ts ← gpsEvent.date
        Trip.ending_lat ← gpsEvent.lat
        Trip.ending_lont ← gpsEvent.lon
        trip.TripStatus ← “finished”
3   end if
```

Algorithm 9: Compute the ending time and location of a given trip

Example 10:

For this example, we assume that the received data for a trip of vehicle 5052 are presented in Table 24.

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
1	5052	2021-12-02 10:00:00	motion	37.939013	23.648045	10.0
2	5052	2021-12-02 10:05:00	motion	37.939020	23.648050	15.0
3	5052	2021-12-02 10:10:00	motion	37.939025	23.648055	25.0
4	5052	2021-12-02 10:15:00	motion	37.939030	23.648060	12.0
5	5052	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0
6	5052	2021-12-02 10:25:00	parked	37.939040	23.648070	0.0
7	5052	2021-12-02 10:35:00	parked	37.939040	23.648070	0.0

Table 24: Sequence of GPS data arrived at the system as an example on how to compute ending time and location of a trip

After computations, trip object's fields corresponding to the ending timestamp and location are displayed in the following table.

_Id	VehicleID	EngineStatus	starting_ts	ending_ts	ending_lat	ending_lon
1	5052	parked	1638482400	1638433500000	37.939040	23.648070

Table 25: Ending time and location of given trip

As we mentioned above, ending timestamp, ending latitude, and ending longitude are the respectively values of the chronologically first GPS record with "parked" engine status.

4.5. Handling of Delayed Data

A challenging part of this thesis is to correctly update the computed statistics, when data arrive at the system later than expected. As we have mentioned above, one of the core concepts of any streaming application is the time semantics. A streaming processing pipeline can depend either on processing time, or on event time. Processing time refers to the current time of the machine clock, while event time refers to the time that an event has occurred, and it is usually embedded within the record. In our case, the timestamp field of GPSEvent is defined as the event time. To calculate all the statistics, our implementation is depending on the event time, namely the timestamp within the emitted GPS data.

As we presented in Table 6, for each trip, an ArrayList with all the corresponding data for the specific trip is created. Since we keep an ArrayList with the corresponding data for each trip, every time we identify a delayed event, we insert it into the correct index of the List, using a binary search algorithm. Thus, our application identifies the index that the delayed GPS record should be and the necessary recalculations are triggered.

The following sections describe how recalculations of each desired statistics are performed.

4.5.1. *Starting Time and Location for Late Events*

As we mentioned above after system detectss that a record arrived at the system later than expected and is inserted into the corresponding ArrayList, in the correct order, recalculations are triggered. Recalculations of starting time and location of a trip are executed only if a delayed event should be the first record of the specific trip. In any other case there is no need to recalculate the starting time and location since it will not be changed. The new starting time and location are those of the record that was late and just arrived at the application. Following we present an example to clarify our proposed solution.

Example 11:

Let us assume that Table 26 presents the data that have been arrived at the system for a specific trip of the vehicle 5052.

According to these data, starting time is 2021-12-02 10:00:00, starting latitude is 37.939013 and starting longitude is 23.648045.

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
			Late event should go here			
2	5052	2021-12-02 10:00:00	motion	37.939013	23.648045	10.0
3	5052	2021-12-02 10:05:00	motion	37.939020	23.648050	15.0
4	5052	2021-12-02 10:10:00	motion	37.939025	23.648055	25.0
5	5052	2021-12-02 10:15:00	motion	37.939030	23.648060	12.0
6	5052	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0
7	5052	2021-12-02 10:25:00	parked	37.939040	23.648070	0.0
8	5052	2021-12-02 10:35:00	parked	37.939040	23.648070	0.0

Table 26: GPS data that arrived at the system as an example on how we compute starting timestamp and location in case of delayed data

The following table presents the next data that arrive at the system which is obviously late since its timestamp indicates that it should be the first record of the trip.

	vehicleID	Date	EngineStatus	lat	lon	speed
1	5052	2021-12-02 09:00:00	idling	37.939000	23.648020	0.0

Table 27: Delayed GPS record that just arrived at the system and it should be the first GPS data of the trip

After the arrival of the delayed data, starting time, starting latitude, and starting longitude should be recalculated considering the delayed data. Thus, the values should be updated. After the amendment of the values, starting timestamp is 2021-12-02 09:00:00, starting latitude is 37939000 and starting longitude is 23.648020.

4.5.2. Trip Duration for Late Events

Recalculation of the trip duration should only be triggered if a GPS event is late and it should be the first or the last record of a trip. If the record that arrived later than expected is placed on the first or the last index of the corresponding ArrayList, which stores the data of the trip, then the same procedure as described in Section 4.2.2 is followed.

4.5.3. Motion & Idle Duration for Late Events

In this section we discuss how the time duration that a vehicle spent being in “motion” or “idling” state is calculated, when delayed data arrive at the system. To effectively update the motion and idle duration of a trip in case of out-of-order GPS records, we consider the cases that are presented in the below table.

Case	Description
1	The delayed event is of engine status “motion” or “idling”, it should be the first record of the trip, and it is followed by a record with the same engine status.
2	The delayed event is of engine status “motion” or “idling”, it should be the first record of the trip, and it is followed by a record with different engine status.
3	The delayed event is of engine status “motion” or “idling”, it should be the last record of the trip, and the previous GPS event is of the same engine status.
4	The delayed event is of engine status “motion”, it should be the last record of the trip, and the previous record is of engine status “idling”, **or** the late event is of engine status “idling”, it should be the last record of the trip, and the previous record is of engine status “motion”
5	The delayed event is of engine status “motion”, while both previous and subsequence records are of engine status “idling”, **or** the delayed event is of engine status “idling”, while both previous and next records are of engine status “motion”.
6	The delayed event is of engine status “motion”, the previous record is of engine status “motion”, while the subsequence record is of engine status “idling”, **or** the delayed data is of engine status “idling”, the previous record is of engine status “idling”, and the subsequence record is of engine status “motion”.
7	The delayed event is of engine status “motion”, the previous record is of engine status “idling”, while the subsequence record is of engine status “motion”, **or** the delayed event is of engine status “idling”, the previous record is of engine status “motion”, and the subsequence record is of engine status “idling”

Table 28: Different cases for recalculating motion or idling duration of a trip in case of delayed data

Below we discuss in more detail the aforementioned cases, given some examples to clarify the designed implementation.

Case 1:

The delayed event is of engine status "motion", it should be the first data of the trip, and it is followed by data of the same engine status, namely "motion". To illustrate this scenario, Table 29 presents the already arrived data, and Table 30 displays the delayed data that just arrived at the system. Before the arrival of the delayed data, motion duration is 15 minutes. After arrival of the delayed data, motion duration should be 1 hour and 20 minutes. To achieve this, we just compute the difference between the first and the second record of the trip and add this difference to the already calculated motion duration. The exact same procedure is followed if the delayed event is of engine status "idling", it should be the first data of the trip and it is followed by a record of the same engine status, namely "idling"

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
			Late event should go here			
2	5052	2021-12-02 10:05:00	motion	37.939020	23.648050	15.0
3	5052	2021-12-02 10:10:00	motion	37.939025	23.648055	25.0
4	5052	2021-12-02 10:15:00	motion	37.939030	23.648060	12.0
5	5052	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0
6	5052	2021-12-02 10:25:00	parked	37.939040	23.648070	0.0
7	5052	2021-12-02 10:35:00	parked	37.939040	23.648070	0.0

Table 29: GPS data as an example on how to compute motion & idle duration in case of late first GPS record, which is followed by a GPS record with the same engine status field

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
1	5052	2021-12-02 09:00:00	motion	37.939000	23.648020	30.5

Table 30: Delayed GPS record that just arrived at the system as an example on how to compute motion & idle duration in case of late first GPS record, which is followed by a GPS record with the same engine status field status

Case 2:

The delayed event is of engine status "motion", it should be the first data of the trip, and it is followed by a GPS record of different engine status, namely "idling" or "parked". To illustrate this scenario, Table 31 displays the sequence of already arrived data, and Table 32 presents the delayed record that just arrived at the system. Before the arrival of the delayed record, motion duration is 5 minutes and idle duration is 15 minutes. After arrival of the delayed record, motion duration should be 1 hour and 10 minutes, and idle duration should be 15 minutes. To achieve this, we just compute the difference between the first and the second record of the trip and add this difference to the already calculated motion duration. The same procedure is followed if the delayed event is of engine status "idling", it should be the first data of the trip and it is followed by a record with different engine status, namely "idling" or "parked".

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
			Late event should go here			
2	5052	2021-12-02 10:05:00	idling	37.939020	23.648050	0.0
3	5052	2021-12-02 10:10:00	idling	37.939025	23.648055	0.0
4	5052	2021-12-02 10:15:00	motion	37.939030	23.648060	12.0
5	5052	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0
6	5052	2021-12-02 10:25:00	parked	37.939040	23.648070	0.0
7	5052	2021-12-02 10:35:00	parked	37.939040	23.648070	0.0

Table 31: GPS data as an example on how to compute motion & idle duration in case of delayed first GPS record, which is followed by a GPS record with different engine status field

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
1	5052	2021-12-02 09:00:00	motion	37.939000	23.648020	30.5

Table 32: Delayed GPS record that just arrived at the system as an example on how to compute motion & idle duration in case of delayed first GPS record, which is followed by a GPS record with different engine status field status

Case 3:

The delayed event is of engine status “motion”, it should be the last data of the trip, and the previous GPS event is of the same engine status, namely “motion”. To illustrate this case, Table 33 presents the already arrived data, and the Table 34 displays the delayed GPS event that just arrived at the system. Before the arrival of the delayed GPS event, motion duration is 5 minutes. After the arrival of delayed GPS event, motion duration should be 15 minutes. To achieve this, we just compute the time difference between the last and the previous GPS event (delayed record) and add this difference to the already computed motion duration. The same procedure is used if the delayed event is of engine status “idling”, it should be the last data, and the previous GPS event is of the same engine status, namely “idling”.

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
1	5052	2021-12-02 10:05:00	idling	37.939020	23.648050	0.0
2	5052	2021-12-02 10:10:00	idling	37.939025	23.648055	0.0
3	5052	2021-12-02 10:15:00	motion	37.939030	23.648060	12.0
4	5052	2021-12-02 10:20:00	motion	37.939035	23.648065	30.0
Late event should go here						

Table 33: GPS data as an example on how to compute motion & idle duration, when the delayed GPS record should be the last record and the previous GPS record is of the same engine status

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
5	5052	2021-12-02 10:30:00	motion	37.939060	23.648090	55.8

Table 34: Delayed GPS record that just arrived at the system as an example on how to compute motion & idle duration, when the delayed GPS record should be the last record and the previous GPS record is of the same engine status

Case 4:

The delayed event is of engine status “motion”, it should be the last data of the trip, and the previous data is of “idling” engine status. Table 35 presents the already processed data, while Table 36 shows the delayed GPS record that just arrived at the system. Based on the already arrived data, motion duration is equal to 5 minutes, and idle duration is equal to 10 minutes. After processing of the delayed record, motion duration should be 5 minutes and idle duration should be 20 minutes. To achieve this, we compute the time difference between the delayed GPS record and the previous GPS record of the given trip, and we add it to the computed idle duration.

Like the aforementioned calculations, if the delayed event is of engine status "idling", it should be the last data of the trip, and the previous GPS record is of different engine status, namely "motion", the difference between the delayed GPS record and the previous GPS record is calculated and added to the motion duration.

	vehicleID	Date	EngineStatus	lat	lon	speed
1	5052	2021-12-02 10:05:00	idling	37.939020	23.648050	0.0
2	5052	2021-12-02 10:10:00	idling	37.939025	23.648055	0.0
3	5052	2021-12-02 10:15:00	motion	37.939030	23.648060	12.0
4	5052	2021-12-02 10:20:00	idling	37.939035	23.648065	30.0
Late event should go here						

Table 35: GPS data as an example on how to compute motion & idle duration, when the delayed GPS record should be the last record and the previous GPS record is of different engine status

	vehicleID	Date	EngineStatus	lat	lon	speed
5	5052	2021-12-02 10:30:00	motion	37.939000	23.648020	30.5

Table 36: Delayed GPS record that just arrived at the system as an example on how to compute motion & idle duration, when the delayed GPS record should be the last record and the previous GPS record is of different engine status

Case 5:

In this case, the delayed event is of engine status "motion", while the previous and the next records are of engine status "idling". To clarify this scenario, Table 37 shows the already arrived and processed data, while Table 38 presents the delayed GPS record that just arrived at the system. Based on the already processed data, the motion duration is equal to 25 minutes, while idle duration is equal to 30 minutes. After the processing of the delayed data, the motion duration should be 35 minutes, while the idle duration should be equal to 20 minutes. To achieve this, we compute the difference between the next and the delayed GPS record. Then, we add the difference to the already calculated motion duration and subtract it from the calculated idle duration.

Similar procedure has been adopted when the delayed event is of engine status "idling", while the previous and the next records are of engine status "motion". The one and only change is that we added the difference to the idle duration and subtract it from the motion duration.

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
1	5052	2021-12-02 10:00:00	motion	37.939013	23.648045	10.0
2	5052	2021-12-02 10:05:00	motion	37.939020	23.648050	55.0
3	5052	2021-12-02 10:10:00	motion	37.939025	23.648055	65.0
4	5052	2021-12-02 10:15:00	motion	37.939030	23.648060	70.0
5	5052	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0
6	5052	2021-12-02 10:25:00	idling	37.939035	23.648065	0.0
Late event should go here						
8	5052	2021-12-02 10:40:00	idling	37.939045	23.648075	65.0
9	5052	2021-12-02 10:45:00	idling	37.939050	23.648080	70.0
10	5052	2021-12-02 10:50:00	motion	37.939060	23.648085	45.0
11	5052	2021-12-02 10:55:00	motion	37.939065	23.648090	80.0

Table 37: GPS data as an example on how to compute motion & idle duration, when before and after the delayed data, there are GPS records of the same engine status

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
7	5052	2021-12-02 10:30:00	motion	37.939040	23.648070	55.0

Table 38: Delayed data as an example on how to compute motion & idle duration, when before and after the delayed data, there are GPS records of the same engine status

Case 6:

Another scenario that is worth to mention is the following: the delayed GPS record is of engine status “motion”, the previous record is of engine status “motion”, while the next record is of engine status “idling”. Tables 39 and 40 illustrate this scenario. After calculations on the already arrived data, motion duration is equal to 40 minutes, while idle duration is equal to 15 minutes. After the processing of the delayed data, motion and idle duration should remain the same as it was calculated by the application before the arrival of the delayed record.

Like the aforementioned procedure, when the delayed record is of engine status “idling”, the previous record is of engine status “idling”, and the next record is of engine status “motion”, then the calculated motion and idle duration should remain unchanged.

	vehicleID	Date	EngineStatus	lat	lon	speed
1	5052	2021-12-02 10:00:00	motion	37.939013	23.648045	10.0
2	5052	2021-12-02 10:05:00	motion	37.939020	23.648050	55.0
3	5052	2021-12-02 10:10:00	motion	37.939025	23.648055	65.0
4	5052	2021-12-02 10:15:00	motion	37.939030	23.648060	70.0
5	5052	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0
6	5052	2021-12-02 10:25:00	motion	37.939035	23.648065	0.0
Late event should go here						
8	5052	2021-12-02 10:40:00	idling	37.939045	23.648075	65.0
9	5052	2021-12-02 10:45:00	idling	37.939050	23.648080	70.0
10	5052	2021-12-02 10:50:00	motion	37.939060	23.648085	45.0
11	5052	2021-12-02 10:55:00	motion	37.939065	23.648090	80.0

Table 39: GPS data as an example on how to compute motion & idle duration, when before the delayed GPS record there is a GPS record of the same engine status and after the delayed data, there is a GPS record of different engine status

	vehicleID	Date	EngineStatus	lat	lon	speed
7	5052	2021-12-02 10:30:00	motion	37.939040	23.648070	55.0

Table 40: Delayed GPS record as an example on how to compute motion & idle duration, when before the delayed GPS record there is a GPS record of the same engine status and after the delayed data, there is a GPS record of different engine status

Case 7:

For this scenario, we assume that the delayed GPS record is of engine status “motion”, the previous record is of engine status “idling”, while the next record is of engine status “motion”. Table 41 displays the already processed data, while Table 42 shows the delayed record that just arrived at the system. Motion duration calculated on the data that have already processed by the application is equal to 10 minutes, while idle duration is equal to 25 minutes. After processing all the data (including the delayed data), motion duration should be 20 minutes, while idle duration should be 15 minutes. To correctly calculate the motion and the idle duration considering the delayed data, we calculate the time difference between the delayed record and the next record, we add this difference to the already calculated motion duration and subtract it from the already calculated idle duration.

If the delayed record is of engine status "idling", the previous record is of engine status "motion", and the next record is of engine status "idling", then the procedure is similar, but the difference is added to the already calculated idle duration and it is subtracted from the motion duration.

	vehicleID	Date	EngineStatus	lat	lon	speed
1	5052	2021-12-02 10:20:00	idling	37.939035	23.648065	0.0
2	5052	2021-12-02 10:25:00	idling	37.939035	23.648065	0.0
			Late event should go here			
4	5052	2021-12-02 10:40:00	motion	37.939045	23.648075	65.0
5	5052	2021-12-02 10:45:00	idling	37.939050	23.648080	70.0
6	5052	2021-12-02 10:50:00	motion	37.939060	23.648085	45.0
7	5052	2021-12-02 10:55:00	motion	37.939065	23.648090	80.0

Table 41: GPS data as an example on how to compute motion & idle duration, when before the delayed GPS record there is a GPS record of different engine status and after the delayed data, there is a GPS record of the same engine status

	vehicleID	Date	EngineStatus	lat	lon	speed
3	5052	2021-12-02 10:30:00	motion	37.939040	23.648070	55.0

Table 42: Delayed GPS data as an example on how to compute motion & idle duration, when before the delayed GPS record there is a GPS record of different engine status and after the delayed data, there is a GPS record of the same engine status

4.5.4. *Parked Duration for Late Events*

In this section, we try to clarify the procedure that we follow to correctly compute the parked duration of a vehicle in case of out-of-order data. The parked duration of a trip needs to be recalculated in the following cases.

Case 8:

Let us assume that Table 43 displays the already processed data. Two trips are completed, and the delayed GPS record, which is shown in Table 44, belongs to the second trip. As we already mentioned, the parked duration is defined as the time duration from a record of engine-off status (“parked”) until a record of engine-on status (“idling” or “motion”).

With that said, before the processing of the delayed data, the parked duration of the first trip is equal to 13 minutes. After the processing of the delayed data, the parked duration of the first trip should be 8 minutes. As we discussed in Section 4.3.1., when a delayed record should be the first data of a trip, recalculation of starting time is triggered. Moreover, we have already defined as parked duration of a given trip the time passed between the end of this trip and the start of a new trip. Thus, to calculate the parked duration we simply compute the difference between the starting time of the trip, whose delayed record should be the first, and the ending time of the previous trip.

	vehicleID	**Date**	**EngineStatus**	**lat**	**lon**	**speed**
1	5052	2021-12-02 10:10:00	idling	37.939035	23.648065	0.0
2	5052	2021-12-02 10:15:00	idling	37.939035	23.648065	0.0
3	5052	2021-12-02 10:30:00	motion	37.939045	23.648075	65.0
4	5052	2021-12-02 10:32:00	parked	37.939048	23.648080	0.0
Late event should go here						
5	5052	2021-12-02 10:45:00	idling	37.939140	23.648277	0.0
6	5052	2021-12-02 10:45:00	idling	37.939150	23.648280	0.0
7	5052	2021-12-02 10:50:00	motion	37.939160	23.648285	45.0
8	5052	2021-12-02 10:55:00	motion	37.939165	23.648290	80.0

Table 43: GPS data as an example on how to compute parked duration of a trip in case of delayed first data of the next trip

	vehicleID	Date	EngineStatus	lat	lon	speed
3	5052	2021-12-02 10:40:00	motion	37.939120	23.648170	55.0

Table 44: Delayed data that just arrived at the system, and it should be the first data of a next trip

Case 9:

In this case, only one trip is completed, and Table 45 shows the already arrived and processed data. It is possible more than one record of engine status “parked” to be emitted for a specific trip. When no trip has been started after a completed trip, then the parked duration for the completed trip is calculated considering the data of engine status “parked”. Thus, based on displayed in Table 45 data, parked duration is 13 minutes.

After the delayed record [Table 46] reaches the system, the parked duration should be 18 minutes. To calculate the parked duration, we just compute the time difference between the current “parked” record and the ending time of the trip. As we have already mentioned, we assume that the ending time of a given trip is the timestamp of the first emitted “parked” GPS record.

	vehicleID	Date	EngineStatus	lat	lon	speed
1	5052	2021-12-02 10:10:00	idling	37.939035	23.648065	0.0
2	5052	2021-12-02 10:15:00	idling	37.939035	23.648065	0.0
3	5052	2021-12-02 10:30:00	motion	37.939045	23.648075	65.0
4	5052	2021-12-02 10:32:00	parked	37.939048	23.648080	0.0
5	5052	2021-12-02 10:45:00	parked	37.939140	23.648277	0.0
Late event should go here						

Table 45: GPS data as an example on how to compute parked duration if the delayed GPS data is of engine status parked and no other trip is performed.

	vehicleID	Date	EngineStatus	lat	lon	speed
6	5052	2021-12-02 10:50:00	parked	37.939155	23.648288	0.0

Table 46: Delayed data that just arrived at the system, it is of engine status parked and it should be the last GPS record of the trip.

4.5.5. Average & Maximum Speed for Late Events

Calculation of average and maximum speed have been discussed in Section 4.2.4. and 4.2.5 respectively. For the computation of these statistics considering out-of-order data, we follow the same procedure. The only crucial part is to assign the delayed data to the correct order. For the maximum speed we compare the already calculated maximum speed with the speed of the delayed GPS record. If the speed of the delayed data is greater, then the maximum speed is the one of this data. For the average speed of a trip, if the delayed GPS record is of engine status "motion" or "idling" the same procedure as described in Section 4.2.4 is followed.

4.5.6. Speed Violations for Late Events

In *Section 4.2.6.*, we described the procedure that we follow to count the occurred speed violations. In case of a delayed record, we compare its speed with the speed of the next and previous record. If the delayed GPS record has greater speed than the specified threshold, and both the next and the previous records have speed below the speed violation threshold, then speed violation has been occurred. Thus, we increase by one the already counted speed violation. If at least one from the next or the previous GPS records has also greater speed than the specified threshold, speed violation has been already counted so we do not increase the already calculated value.

4.5.7. Ending Time and Location for Late Events

As we have mentioned before, the ending time and the location in our implementation is the timestamp and the location of the first emitted data of engine status "parked" for a given trip. Thus, the only case where we update the ending time and the location is when the delayed record should be the first "parked" record of a given trip. In any other case, the already calculated ending time and the location should not be changed.

4.5.8. Updating Trip Statistics if a Parked GPS Data is Late

Correct identification of a trip is of crucial importance for our application. A trip starts when the driver switches on the engine of the vehicle, and it is considered as completed when the engine is switched off. Thus, when after a "parked" GPS record, a GPS record of engine status "idling" or "motion" reaches the system, then a new trip is generated. When a GPS record of engine status "parked" arrives at the system, we assume that the corresponding trip has been completed. There is a chance that a GPS record that should be the last data of a trip is delayed. In this case, trips are not separated in a proper way and statistics are incorrectly computed. Our application should be able to correctly identify trips even in case of out-of-order "parked" data. To illustrate this scenario, let us assume that Table 47 displays the already arrived and processed data.

	vehicleID	Date	EngineStatus	lat	lon	speed
1	5052	2021-12-02 10:10:00	idling	37.939035	23.648065	0.0
2	5052	2021-12-02 10:15:00	idling	37.939035	23.648065	0.0
3	5052	2021-12-02 10:30:00	motion	37.939045	23.648075	65.0
4	5052	2021-12-02 10:32:00	motion	37.939048	23.648080	77.0
5	5052	2021-12-02 10:45:00	motion	37.939140	23.648277	95.2
			Late data event should go here			
7	5052	2021-12-02 10:49:00	idling	37.939150	23.648280	0.0
8	5052	2021-12-02 10:50:00	motion	37.939160	23.648285	45.0
9	5052	2021-12-02 10:55:00	motion	37.939165	23.648290	80.0
10	5052	2021-12-02 11:15:00	idling	37.939266	23.648299	0.0
11	5052	2021-12-02 11:35:00	idling	37.939393	23.648300	0.0

Table 47: GPS data as an example on how to compute trip statistics in case of delayed parked GPS record, which should be the last GPS record of a trip

Since no GPS data of engine status "parked" have arrived at the system, all data will be considered as data of the same trip. Thus, the idle duration of the trip is 41 minutes, the motion duration is 44 minutes, the parked duration is 0 minutes, and the trip duration is equal to 85 minutes. Also, the maximum speed is equal to 80.00 km/h and the average speed is equal to 36.22 km/h. Assuming that the speed threshold is 50 km/h, two speed violations have occurred.

Assuming now that the record that is shown in Table 48 is delayed and just arrived at the system, it is easily understood that statistics for two trips should be computed.

	vehicleID	Date	EngineStatus	lat	lon	speed
6	5052	2021-12-02 10:47:00	parked	37.939155	23.648288	0.0

Table 48: Delayed GPS record of engine status "parked" that it should be the last GPS record of a trip

After processing the delayed GPS record, the statistics of the first trip should be the following: the idle duration is equal to 20 minutes, the motion duration is 17 minutes, the parked duration is 2 minutes, the trip duration is 37 minutes, the maximum speed is 95.2 km/h, the average speed is 47.44 km/h, and one speed violation have occurred. For the second trip the statistics should be the following: the idle duration is 21 minutes, the motion duration is 25 minutes, the parked duration is 0 minutes, the trip duration is 46 minutes, the maximum speed is 80 km/h, the average speed is equal to 25 km/h, and one speed violation has been performed.

To correctly calculate trip statistics, when a delayed GPS record of engine status "parked" arrives at the system, we need to generate a new trip and assign the corresponding data to this new trip. To achieve this, we loop through the data that should be in the new trip, we add them one by one to this trip, and we delete them from the trip that they were wrongly assigned. To calculate the statistics for the new trip we use the same logic that we have described above for events that have arrived at the system in the correct order.

Since some of the data were mistakenly taken into consideration for the calculations of the statistics of the completed trip, we need to recompute some of these statistics. For this reason, the following rules have been adopted. The starting time and the starting location are unchanged. For the ending timestamp and the ending location we just use the timestamp and location of the delayed "parked" GPS data. For the trip duration we compute the difference between the ending timestamp and the starting timestamp of the trip. Parking duration is calculated by calculating the difference between the starting timestamp of the new trip and the ending timestamp of the completed trip. For updating the other durations, we need to detect in what engine status was the trip before the "parked" GPS record. If the trip was in "motion" status, then motion duration is computing using the formula:

Completed trip's motion duration = already computed motion duration – new trip's motion duration – parked duration.

To compute the idle duration, we use the formula:

Completed trip's idle duration = already computed idle duration – new trip's idle duration

In a similar way, we compute the durations if the trip was in "idling" status. If this is the case, the used formulas are the following:

Completed trip's idle duration = already computed idle duration – new trip's idle duration – parked duration.

Completed trip's motion duration = already computed motion duration – new trip's motion duration

To compute the maximum speed of the new trip we calculate the max speed of the ArrayList of the corresponding data. To compute the average speed of the new trip we calculate the sum of the corresponding non parked data and we divided by the number of these data. Moreover, we need to recalculate the average speed of the completed trip. To achieve this, we use the following equations.

Completed trip's sum of speed = already computed sum of speed – new trip's sum of speed

Completed trip's non parked data = already computed non parked data – new trip's non parked data

Completed trip's average speed = completed trip's sum of / completed trip's non parked data

Finally, to compute the performed speed violations we need to consider the following cases: If in the previous and next records of a "parked" GPS data, speed violation has occurred, then the following formula is used:

Completed trip's speed violations = already computed speed violations – new trip's speed violations + 1

In any other case the following formula is used:

Completed trip's speed violations = already computed speed violations – new trip's speed violations

4.6. *Storage & Visualization*

In the proposed architecture, Elasticsearch is used as a storage layer in order to store the aggregated results and provide an efficient query service to the user. To add a visualization layer in our application we used Kibana on top of Elasticsearch.

First, we create an index called "gps_event_idx" using a PUT request from Dev Tools of Kibana. In the following Table we present the properties of this index and their corresponding types.

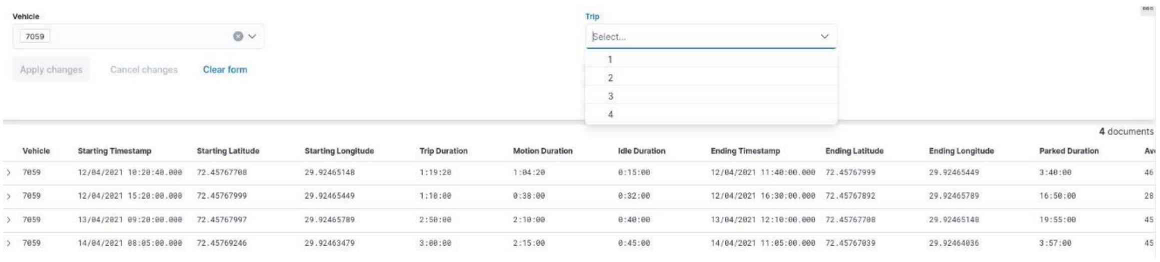
Properties	**Type**
VehicleID	text
TripCount	integer
Starting_ts	date
Starting_lat	double
Starting_lon	double
Starting_location	geo_point
Ending_ts	date
Ending_lat	double
Ending_lon	double
Ending_location	geo_point
TripDuration	double
MotionDuration	double
IdleDuration	double
ParkedDuration	double
AvgSpeed	double
MaxSpeed	double
CntSpeedViolations	integer

Table 49: Properties & types of Elasticsearch index

As we have mentioned in Section 4.2., every time that a new GPS record arrives at our proposed application, a custom function, which is responsible for the computations of the values of a given trip, is triggered.

After the processing of each GPS record, the fields of the trip object are updated and this trip object is sent to Elasticsearch. Thus, Elasticsearch stores one document (JSON) for each trip, with defined values for all the fields that are presented in Table 49.

As soon as the aggregated data are stored into Elasticsearch, the users have access to the dashboards, which help them have a clear overview of vehicles' trips. The user can select the vehicle and the trip, for which he wants to examine the statistics. If no trip is selected, then the statistics for all the trips of the selected vehicle will be shown. Figure 24 presents the table and the values which are shown in Kibana's dashboard. This data representation is equivalent to the route report that is presented in Figure 1.



	Vehicle	Starting Timestamp	Starting Latitude	Starting Longitude	Trip Duration	Motion Duration	Idle Duration	Ending Timestamp	Ending Latitude	Ending Longitude	Parked Duration	Av
>	7059	12/04/2021 10:20:40.000	72.45767708	29.92465148	1:19:20	1:04:20	0:15:00	12/04/2021 11:40:00.000	72.45767999	29.92465449	3:40:00	46
>	7059	12/04/2021 15:20:00.000	72.45767999	29.92465449	1:10:00	0:38:00	0:32:00	12/04/2021 16:30:00.000	72.45767892	29.92465789	16:50:00	28
>	7059	13/04/2021 09:20:00.000	72.45767997	29.92465789	2:50:00	2:10:00	0:40:00	13/04/2021 12:10:00.000	72.45767708	29.92465148	19:55:00	45
>	7059	14/04/2021 08:05:00.000	72.45769246	29.92463479	3:00:00	2:15:00	0:45:00	14/04/2021 11:05:00.000	72.45767039	29.92464036	3:57:00	45

Figure 24: Statistics of all trips per vehicle in table format.

Another useful graphic representation is the bar chart, which presents the duration of the trip and the time that the selected vehicle spent being in motion, idle or parked state per trip. Figure 25 displays an example of this bar chart.

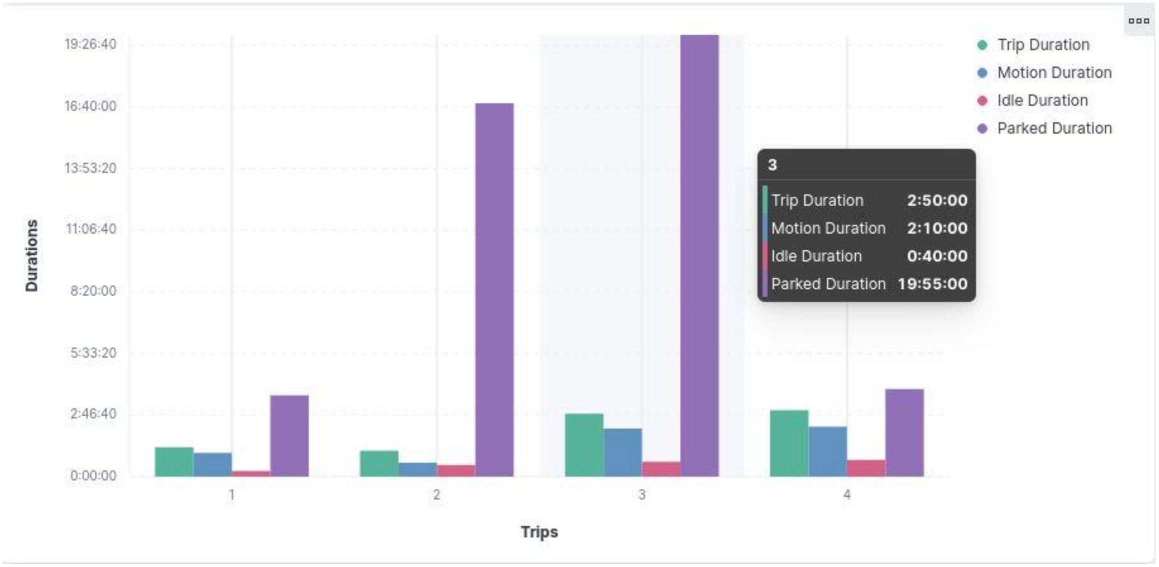


Figure 25: Bar chart of durations per trip.

Monitoring the average and the maximum speed per trip is also particularly important for a fleet management application. For this scope, the following area chart, which presents the average and the maximum speed per trip, was implemented.

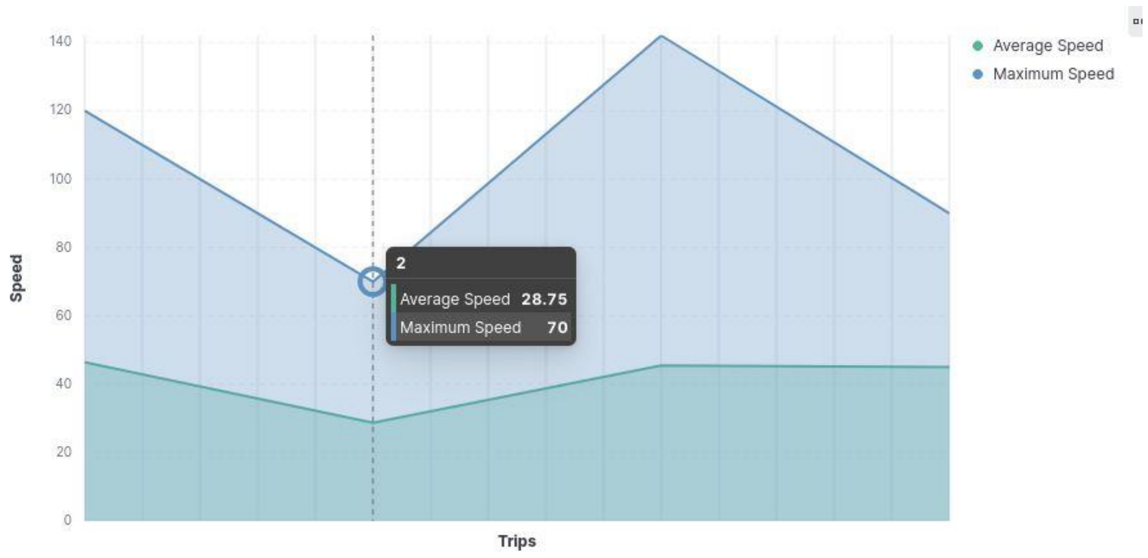
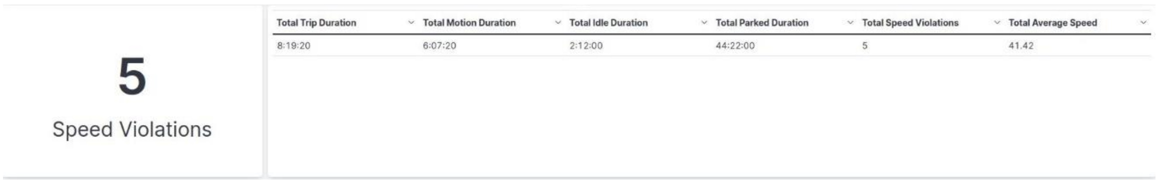


Figure 26: Area chart which presents average and maximum speed per trip.

Figure 27 shows the overview of all trips per vehicle. It presents the total trip duration of a vehicle, the total duration that it has spent being in each of the different engine status (motion, idling, parked), the total speed violations and the average speed of all trips that it performed.



5

Speed Violations

Total Trip Duration	Total Motion Duration	Total Idle Duration	Total Parked Duration	Total Speed Violations	Total Average Speed
8:19:20	6:07:20	2:12:00	44:22:00	5	41.42

Figure 27: Total statistics of all trips performed by a vehicle.

Moreover, we add a map layer in the frontend part of our implementation, which presents the starting and ending location of each trip for the selected vehicle.

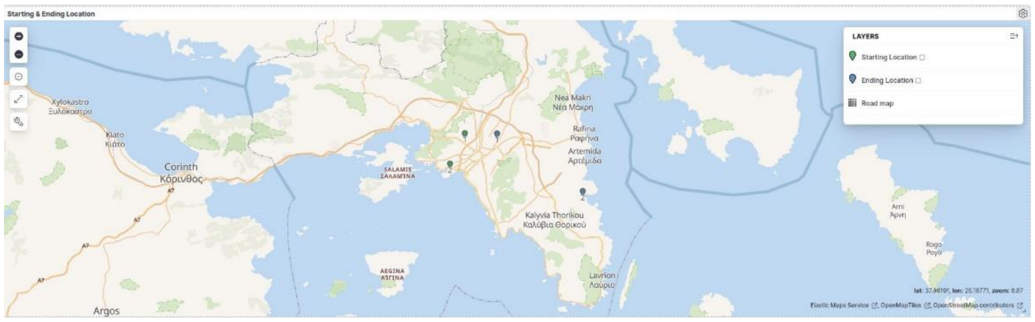


Figure 28: Map that presents starting and ending location of each trip.

5. Performance Evaluation

In this section we present the dataset that was used throughout this thesis and we discuss the performance of our proposed solution.

5.1. Dataset Description

The dataset that was used throughout this study has been provided by Vodafone Innovus, a company which offers fleet management services. It is a synthetic set of data containing anonymous GPS traces from virtual vehicles. This dataset contains 1 million records of the following information: vehicle, utcDate, engine status, longitude, latitude, altitude, angle, speed, odometer, satellites. Table 50 presents the description of the data schema. For our implementation we will focus on the following data: vehicle, utcDate, engine status, location points, and speed. Table 51 displays a sample of the data that we are going to use in the experiments.

	Description	**Type**	**Format**	**Unit**	**Source**
vehicle	Vehicle Random Key	String			Custom Generated
utcDate	UTC Date	Date	YYYY-MM-DD hh:mi:ss.mmm		GPS Date
engineStatus	Indicates if vehicle is moving, idling, or parked	String	Idling, Parked, Motion		Engine Power
longitude	GPS Longitude	Decimal	WGS84	Number	GPS Sensor
latitude	GPS Latitude	Decimal	WGS84	Number	GPS Sensor
altitude	GPS Altitude	Integer		Meters	GPS Sensor
angle	Degrees (0-360^{o}) based on two consecutive GPS coordinates	Integer		Degrees	GPS, Device
speed	Speed of the vehicle based on GPS	Decimal		Km/h	GPS, Device
odometer	Distance in meters between two consecutive GPS coordinates	Integer		Meters	GPS, Device
satellites	Number of satellites seen by GPS Antenna	Integer		Number	GPS Sensor

Table 50: Data schema

vehicle	utcDate	engineStatus	longitude	latitude	speed
7059	2016-04-12 10:20:42.000	motion	29.92465148	72.45767708	70.0
5069	2016-04-12 10:20:43.000	motion	29.92465148	72.45767708	45.0
7059	2016-04-12 10:22:43.000	idling	29.92465200	72.45768100	0.0
5069	2016-04-12 10:22:43.000	idling	29.92465321	72.45767728	0.0
7059	2016-04-12 10:24:43.000	parked	29.92465200	72.45768100	0.0
7059	2016-04-12 10:29:25.000	idling	29.92465200	72.45768100	0.0
5069	2016-04-12 10:29:25.000	idling	29.92465321	72.45767728	0.0

Table 51: Sample of the provided data

5.2. *Performance*

To begin with, we executed performance experiments, using a computer, which is equipped with 4 CPU cores and 16 GB RAM. We conducted 4 experiments using 25%, 50%, 75% and 100% of the data. The following figure shows the number of trips that were produced during these experiments.

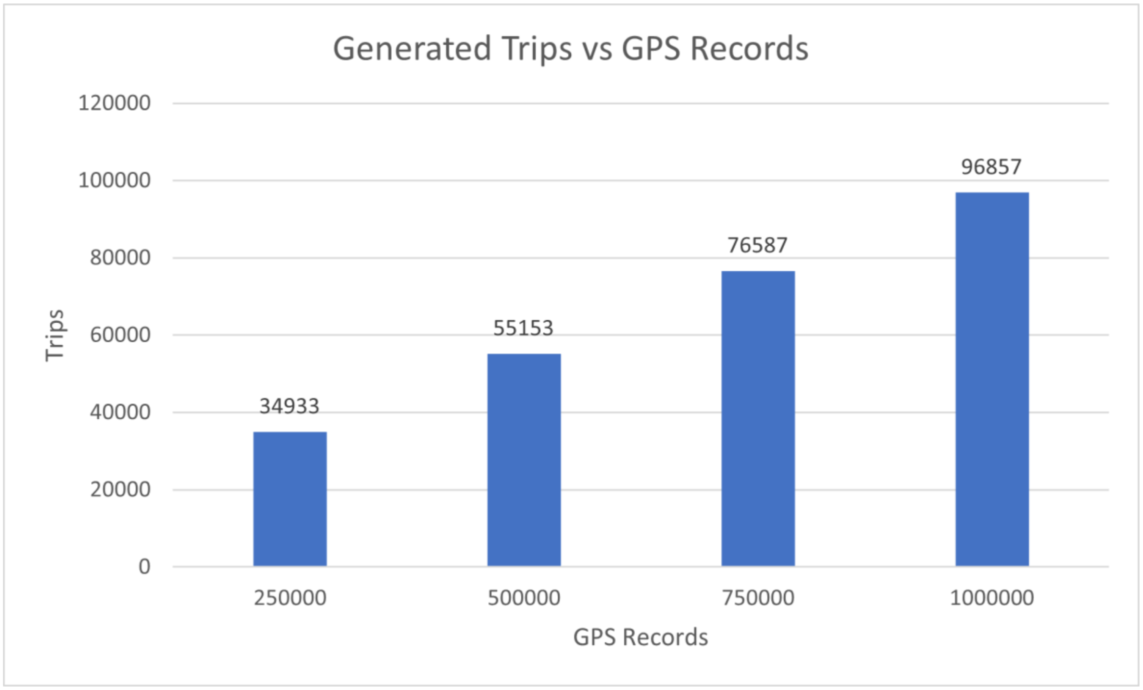


Figure 29: Generated trips vs GPS Records

Our main goal was to examine how the increase of parallelism level affects the processing time. To check this, we have executed each experiment with different job parallelism level (1, 2, 4 & 8). We have observed that when we increase job parallelism, the execution time decreases. In the following figures we present the results from our experiments.

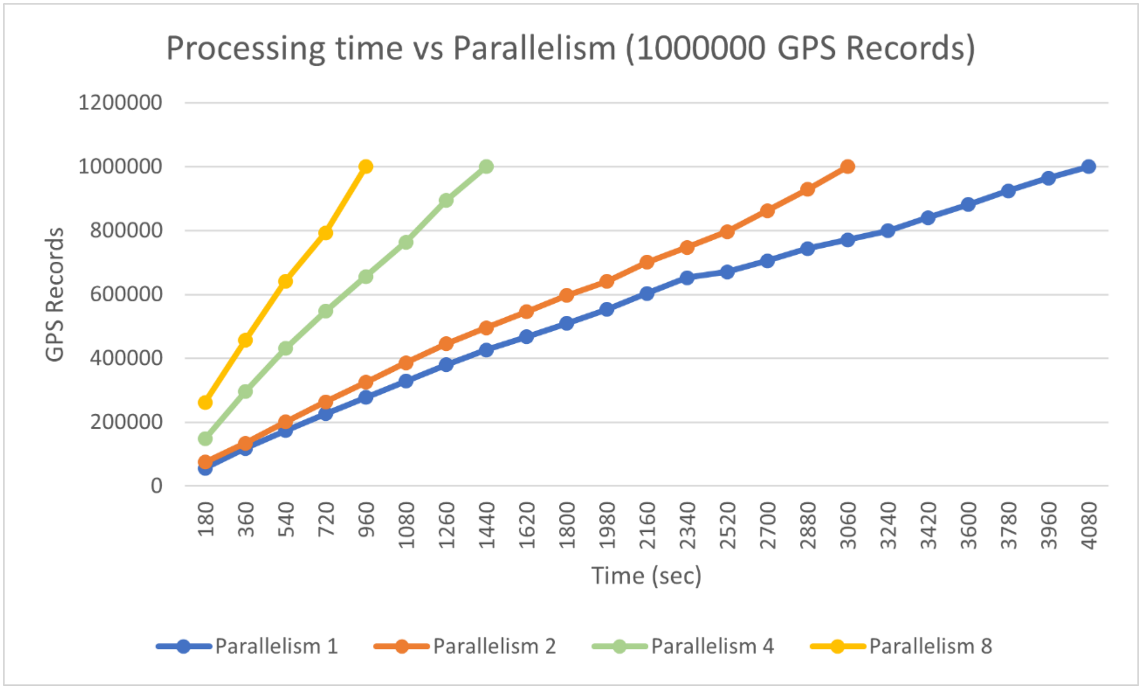


Figure 30: Processing time vs parallelism for 1M GPS records

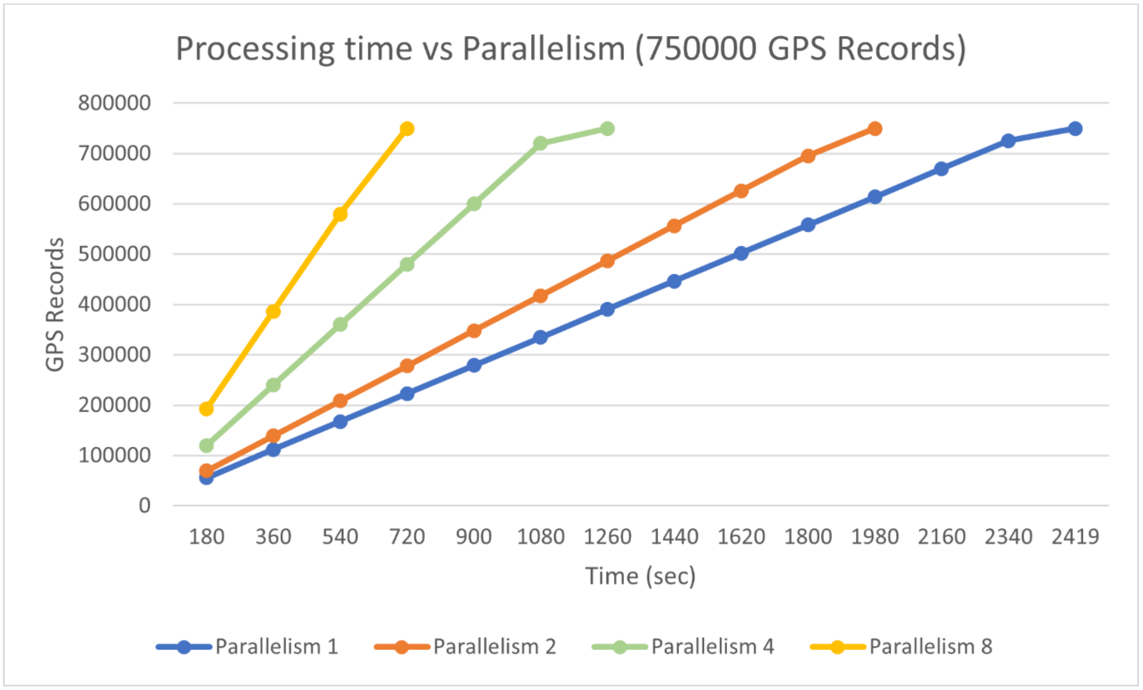


Figure 31: Processing time vs parallelism for 750K GPS records

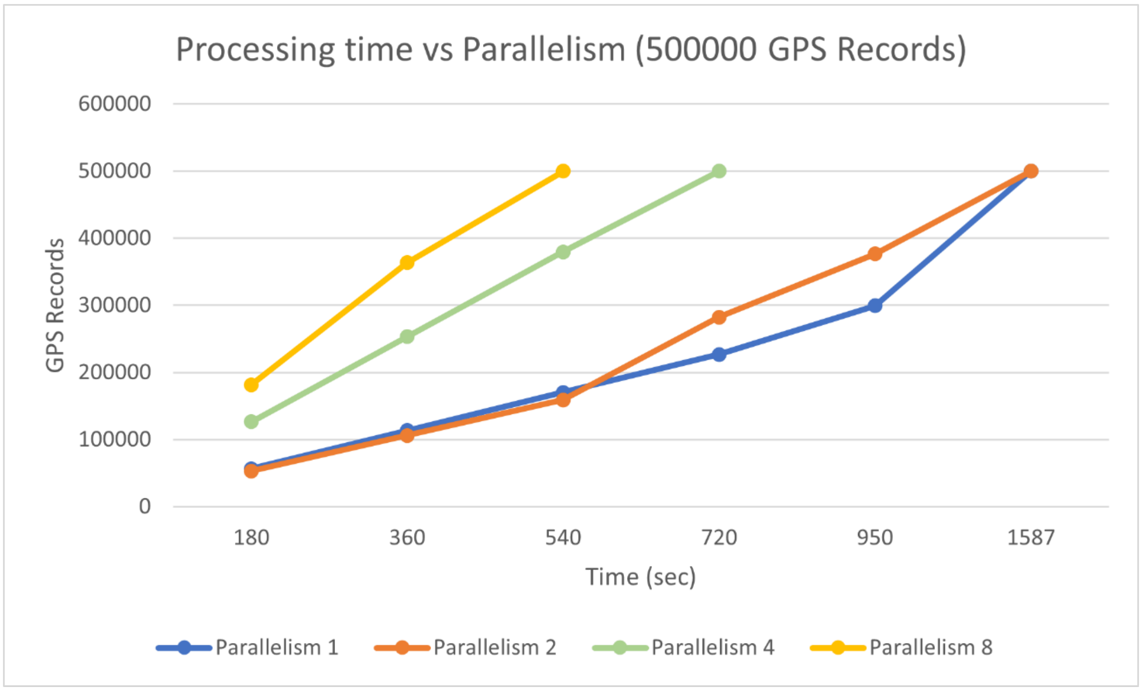


Figure 32: Processing time vs parallelism for 500K GPS records

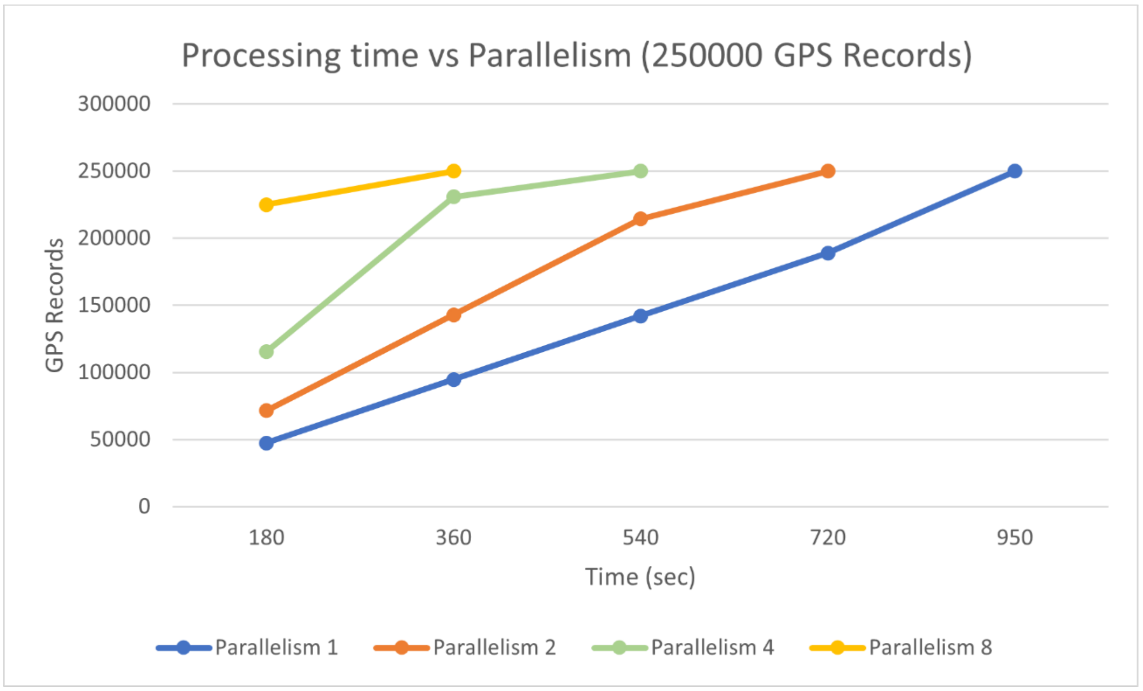


Figure 33: Processing time vs parallelism for 250K GPS records

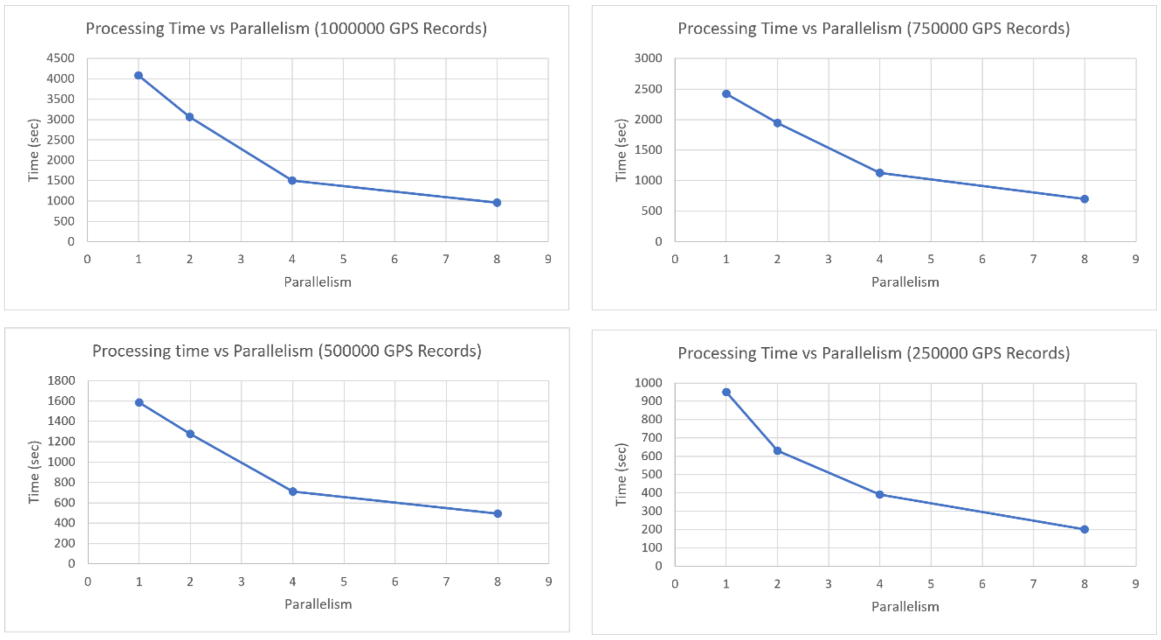


Figure 34: Processing time vs parallelism for all experiments

Another metric that we have taken into consideration during the experiments was the throughput. We wanted to examine how the increase of job parallelism affects the number of GPS records that our application is processing per second. As it was expected, as we double the parallelism level, the throughput almost doubles as well. The following figures presents the results of each experiment.

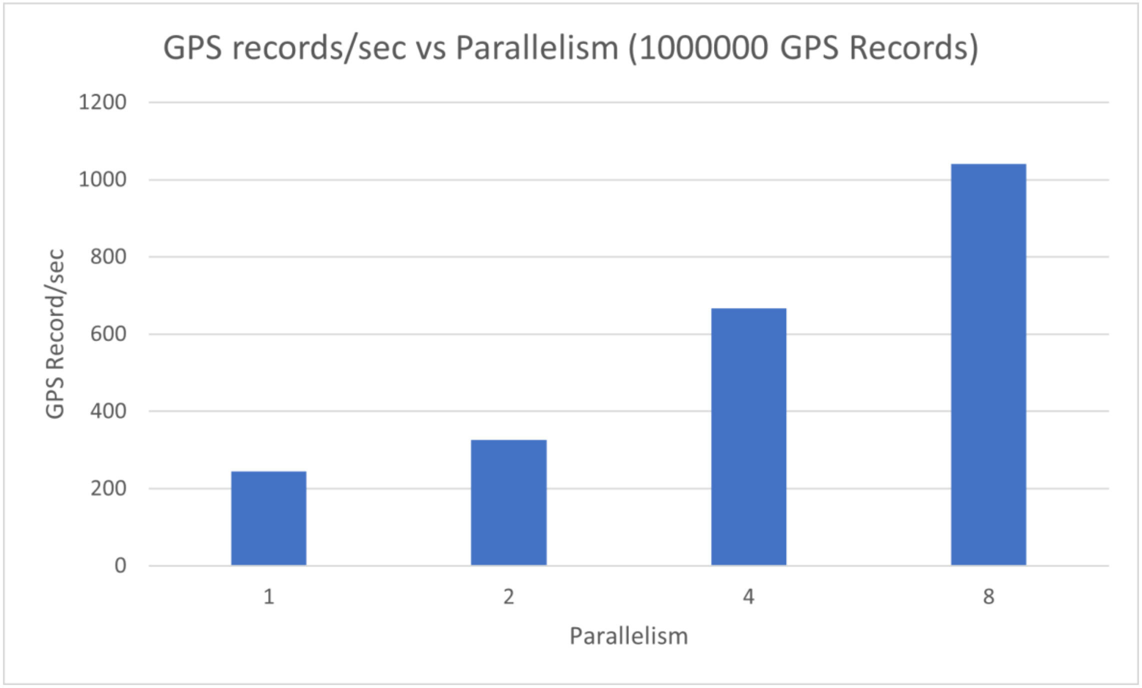


Figure 35: Throughput vs parallelism for 1M GPS Records

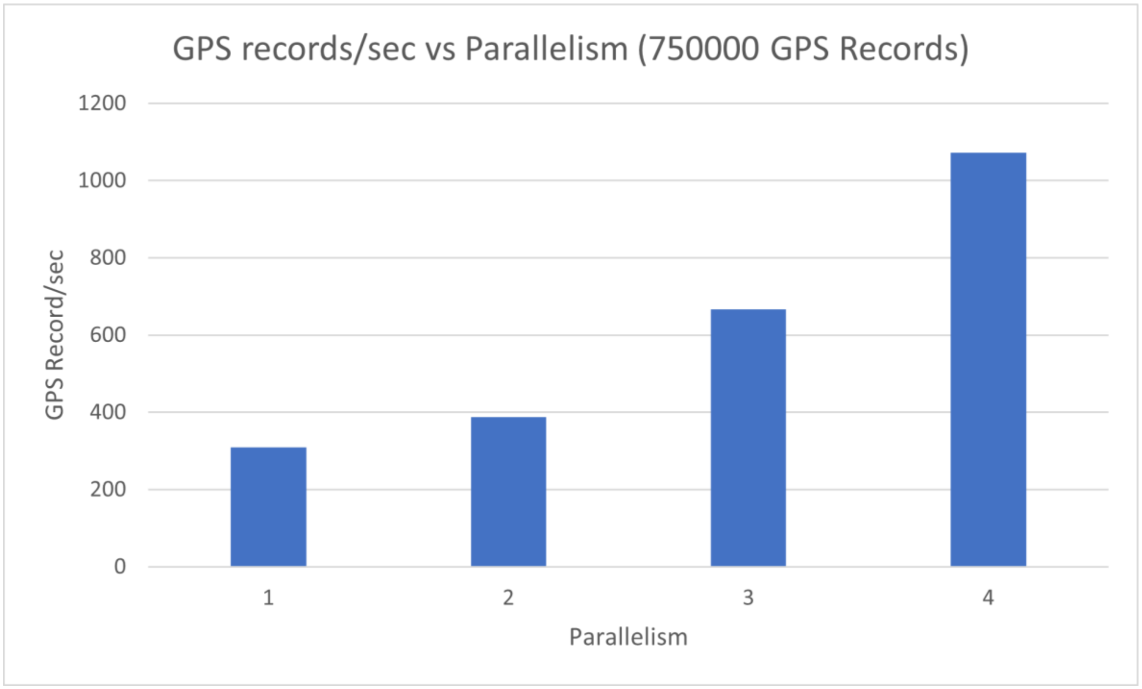


Figure 36: Throughput vs parallelism for 750K GPS records

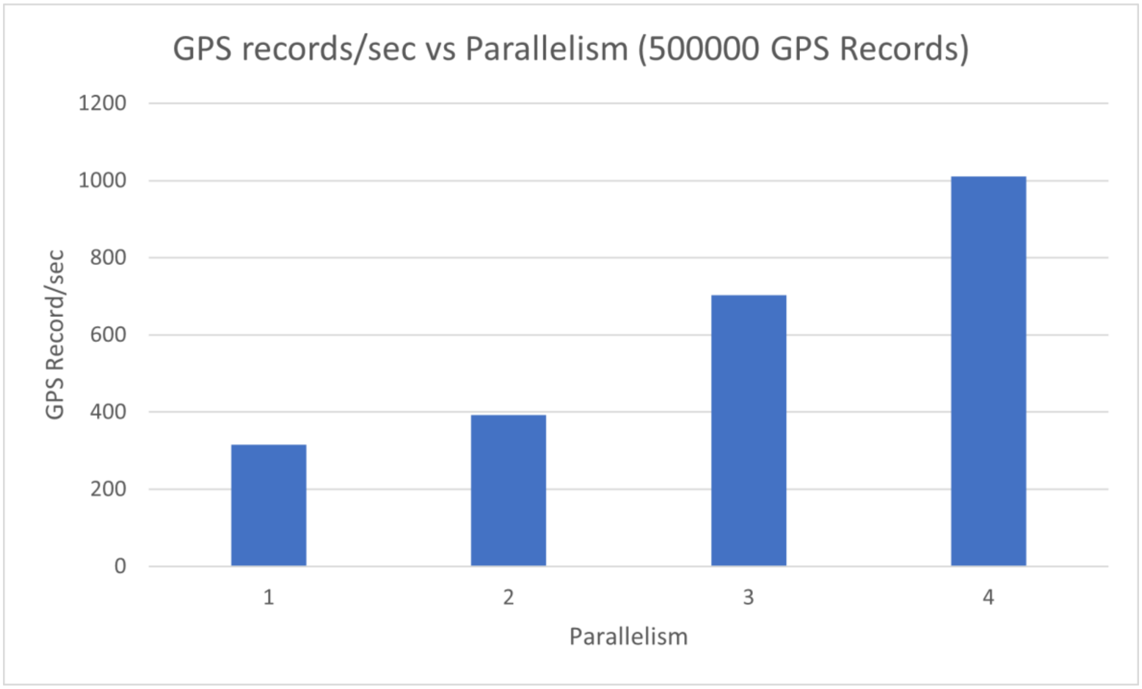


Figure 37: Throughput vs parallelism for 500K GPS records

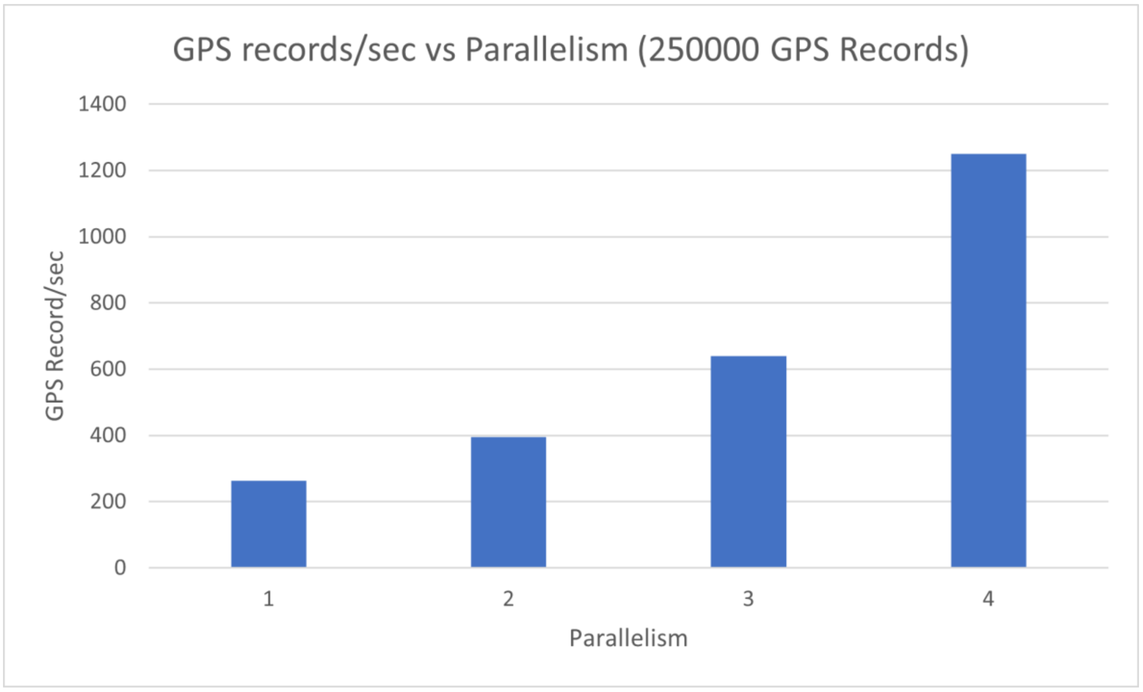


Figure 38: Throughput vs parallelism for 250K GPS records

6. Conclusions & Future Work

In this master thesis we proposed an application which can be considered as a proof of concept on how to compute statistics in real time. These statistics can be used for trajectory analysis of moving vehicles, and for creating a route report [Figure 1]. Our purpose was to design and implement an application, which calculates and updates statistics of a vehicle's trip as soon as new data, emitted by GPS devices, arrive at the system. Since GPS devices may face malfunctions or signal loss, data may arrive at the application later than they have been generated. Thus, for our application to be reliable, the requirements included correct handling of out-of-order data, which may arrive at the system on delay, and update respectively the statistics. The computed trip statistics are stored into Elasticsearch, which provides a fast search engine for users to find the desired metrics for the vehicle and the trip of interest. Last part of the proposed design is the visualization of the metrics, which is handled by Kibana.

For the main part of computations, we used Apache Flink, which is a scalable, distributed and fault-tolerant processing engine that can be used for calculations on bounded (finite) or unbounded (infinite) data. It is a true streaming engine that provides excellent performance with low-latency and high-throughput.

For data ingestion to the application, we used Apache Kafka. Integration of Apache Flink with Apache Kafka provides backpressure handling and guarantees data durability.

As future work we could add into our implementation the following functionalities:

- For computing the speed violations performed by a vehicle per trip, we compare the speed with a predefined threshold. An improvement on this could be to dynamically set the speed threshold based on the current location of the vehicle and the traffic rules.

- Another improvement could be to apply validation rules on the data before performing the computations. We could train machine learning algorithms to detect erroneous data and filter them out to exclude them from the computations.

- To make this application suitable for production, fault tolerance could also be checked and optimized. If our application unexpectedly fails, it should be able to recover properly, so as not to lose any significant data.

7. References

[1] A. Noghabi, S., Paramasivam, K., Pan, Y., Ramesh, N., Bringhurst, J., Gupta, I., & Campbell, R. (2017). Samza: stateful scalable stream processing at LinkedIn. Proceedings of the VLDB Endowment, 10, 1634–1645. https://doi.org/10.14778/3137765.3137770

[2] Kleppmann, M., Sakr S., Zomaya A. (2018) Apache Samza. Encyclopedia of Big Data Technologies. Springer, Cham.
https://doi.org/10.1007/978-3-319-63962-8_197-2

[3] Kleppmann, M., & Kreps, J. (2015). Kafka, Samza and the Unix Philosophy of Distributed Data

[4] Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM, 51, 137–150. https://doi.org/10.1145/1327452.1327492

[5] Doulkeridis, C., & Nørvåg, K. (2013). A Survey of Large-Scale Analytical Query Processing in MapReduce. The VLDB Journal.
https://doi.org/10.1007/s00778-013-0319-9

[6] Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., & Markl, V. (2018). Benchmarking Distributed Stream Data Processing Systems. 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[7] Perera, S., Perera, A., & Hakimzadeh, K. (2016). Reproducible Experiments for Comparing Apache Flink and Apache Spark on Public Clouds.

[8] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, pp. 1789–1792.

[9] Khan, S., Liu, X., Ali, S., & Alam, M. (2019). Storage Solutions for Big Data Systems: A Qualitative Study and Comparison.

[10] Siddiqa, A., Karim, A., & Gani, A. (2017). Big data storage technologies: a survey. Frontiers of Information Technology and Electronic Engineering, 18, 1040–1070. https://doi.org/10.1631/FITEE.1500441

[11] Types of NoSQL Databases. (n.d). MongoDB. Retrieved April 7, 2022, from https://www.mongodb.com/scale/types-of-nosql-databases

[12] What is a graph database? (n.d). Neo4j Graph Data Platform. Retrieved April 7, 2022, from https://neo4j.com/developer/graph-database/

[13] What is NoSQL. (n.d). Redis. Retrieved April 7, 2022, from https://redis.com/nosql/what-is-nosql/

[14] Report-vehicles in use, Europe 2022. (2022, January 19). ACEA-European Automobile Manufacturers' Association. https://acea.auto/publication/report-vehicles-in-use-europe-2022/

[15] Total data volume worldwide 2010-2025. (n.d). Statista. Retrieved April 3, 2022, from https://statista.com/statistics/871513/worldwide-data-created/

[16] Buyya, R., Calheiros, R. N., & Dastjerdi, A. V. (2016). Big Data Principles and Paradigms (pp. 1-468)

[17] Babcock, B., Babu, S., Datar, M., Motwani, R., & Widom, J. (2002). "Models and Issues in Data Stream Systems." Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1–16. https://doi.org/10.1145/543613.543615

[18] Feick, M., Kleer, N., Kohn, M., 2018. Fundamentals of real-time data processing architectures lambda and kappa, in: Becker, M. (Ed.), SKILL2018 - Studierendenkonferenz Informatik, Gesellschaft fur Informatik e.V., Bonn. pp. 55–66

[19] Zhelev, S., & Rozeva, A. (2017). Big data processing in the cloud - Challenges and platforms. AIP Conference Proceedings, 1910, 060013. https://doi.org/10.1063/1.5014007

[20] Kreps, J. (2014, July 2). Questioning the Lambda Architecture. O'Reilly Media. https://oreilly.com/radar/questioning-the-lambda-architecture/

[21] Carbone, P., Gévay, G., Hermann, G., Katsifodimos, A., Soto, J., Markl, V., & Haridi, S. (2017). Large-Scale Data Stream Processing Systems. In Handbook of Big Data Technologies. https://doi.org/10.1007/978-3-319-49340-4_7

[22] Friedman, E., Tzoumas, K. (2016). Introduction to Apache Flink. O'Reilly Media.

[23] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., & Ryaboy, D. (2014). Storm@twitter. https://doi.org/10.1145/2588555.2595641

[24] Documentation. (n.d). Apache.Org. Retrieved April 4, 2022, from https://storm.apache.org/releases/current/index.html/

[25] Carbone, P., Katsifodimos, A., S., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache Flink™: Stream and Batch Processing in a Single Engine. IEEE Data Engineering Bulletin, 38.

[26] Apache Flink Documentation. (n.d). Apache.Org. Retrieved April 6, 2022, from https://nightlies.apache.org/flink/flink-docs-release-1.14/

[27] Samza – documentation. (n.d). Apache. Org. Retrieved April 6, 2022, from https://samza.apache.org/learn/documentation/latest/

[28] Kreps, J., Corp, L., Narkhede, N., Rao, J., &Corp, L. (n.d.). Kafka: a distributed messaging system for log processing.

[29] Kafka streams overview. (n.d.). Confluent.Io. Retrieved April 6, 2022, from https://docs.confluent.io/platform/current/streams/

[30] Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., & Stoica, I. (2010). Spark: Cluster Computing with Working Sets. Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, 10, 10–10.

[31] Damji, J., Wenig, B., Das, T., & Lee, D. (2020). Learning Spark (2nd ed.) O'Reilly Media.

[32] Overview-Spark 3.2.1 Documentation. (n.d.). Apache.Org. Retrieved April 6, 2022, from https://spark.apache.org/docs/latest/

[33] Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.

[34] Zaharia, M., Das, T., Li, H., Shenker, S., & Stoica, I. (2012). Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. 10–10.

[35] Leibiusky, J., Eisbruch, G., Simonassi, D. (2012). Getting Started with Storm, O'Reilly Media.

[36] Dean, A., Cretaz, V. (2019). Event Streams in Action: Real-time event systems with Kafka and Kinesis

[37] Kienzler, R. (2017). Mastering Apache Spark 2.x – (2nd ed.). Packt Publishing.

[38] Marz, N., & Warren, J. (2015). Big Data: Principles and best practices of scalable realtime data systems. Manning Publications.

[39] Akidau, T., Chernyak, S., & Lax, R. (2018). Streaming Systems. O'Reilly Media.

[40] Kejariwal, A., Kulkarni, S., & Ramasamy, K. (2017). Real time Analytics: Algorithms and Systems.

[41] Nicolas Seyvet, I. M. V. (2016, May 19). Applying the Kappa architecture in the telco industry. O'Reilly Media. https://www.oreilly.com/content/applying-the-kappa-architecture-in-the-telco-industry/

[42] Elasticsearch guide. (n.d.). Elastic. Co. Retrieved April 6, 2022, from https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html

[43] A very Brief Introduction to MapReduce. Retrieved April 6, 2022, from https://hci.stanford.edu/courses/cs448g/a2/files/map_reduce_tutorial.pdf