# Exploring real-time data analytics using distributed stream-processing systems

*Arvaniti-Bofili Ioanna-Maria*

Thesis submitted in partial fulfillment of the requirements for the

*Masters' of Science degree in Computer Science and Engineering*

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Assoc. Prof. *K. Magoutis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

This work has been partially supported by the Greek Research Technology Development and Innovation Action "RESEARCH - CREATE - INNOVATE", Operational Programme on Competitiveness, Entrepreneurship and Innovation (ΕΠΑνΕΚ) 2014–2020 through the ProxiTour project (Τ1ΕΔΚ-04819).

University of Crete
Computer Science Department

**Exploring real-time data analytics using distributed stream processing systems**

Thesis submitted by
**Arvaniti-Bofili Ioanna-Maria**
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author:

Arvaniti-Bofili Ioanna-Maria

Committee approvals:

Kostas Magoutis
Associate Professor, Thesis Supervisor

DIMITRIOS PLEXOUSAKIS
Digitally signed by DIMITRIOS PLEXOUSAKIS
Date: 2021.03.03 11:55:46 +02'00'

Dimitris Plexousakis
Professor, Committee Member

Polyvios Pratikakis
Digitally signed by Polyvios Pratikakis
Date: 2021.03.05 07:46:34 +02'00'

Polyvios Pratikakis
Assistant Professor, Committee Member

Departmental approval:

Polyvios Pratikakis
Digitally signed by Polyvios Pratikakis
Date: 2021.03.05 07:47:03 +02'00'

Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, February 2021

# Exploring real-time data analytics using distributed stream processing systems


## Abstract

Processing high-volume streaming data is an important enabling technology in IoT-driven, social networking, and other e-services, having given rise to a new generation of stream-processing systems (SPS). In this thesis, we apply modern SPS technologies to improve the state of the art in two application areas involving real-time position tracking in physical space: maintaining profiles of visitor movement in exhibit spaces and predicting service-level objective violations in mass transit systems. To ensure that scalable SPSs are able to seamlessly adapt to varying levels of load by adjusting their resources, in this thesis we implement a mechanism by which SPSs can scale dynamically even when such capability is not natively supported by the SPS and the underlying resource management platform.

Our first application of SPS technologies to real-time data analytics is on the development of dynamic behavioral profiles of visitors in exhibit spaces based on their movements in physical space. While related approaches have been explored in the past, this thesis applies for the first time stream-processing technologies to materialize behavioral theories developed in social sciences and to collect richer information about visitors' interests. Such profiles can be used to produce recommendations for the visitors about exhibits they should visit, to decide the best content to present to them, or to design personalized questionnaires. Our second application of SPS technologies to real-time data analytics is on training appropriate models for predicting mass-transit vehicles that are likely to violate service-level objectives in their route duration. In this thesis we extend previous delay prediction techniques with the ability to apply predictions in a real-time fashion.

In this thesis we also address the necessary tuning and support for scalable, adaptive data analytics by examining various parameters of the SPS and its ingest engine. Even in cases where dynamic scaling is not explicitly supported by the SPS platform, we demonstrate a technique that achieves scale-out with low downtime.

# Διερεύνηση τεχνικών ανάλυσης δεδομένων σε πραγματικό χρόνο με χρήση κατανεμημένων συστημάτων επεξεργασίας ροών

## Περίληψη

Η κλιμακώσιμη επεξεργασία ροών δεδομένων είναι πολύ σημαντική τεχνολογία στους τομείς των υπηρεσιών του Διαδικτύου των Πραγμάτων (IoT), της κοινωνικής δικτύωσης, και άλλων τομέων των ηλεκτρονικών υπηρεσιών, έχοντας πρόσφατα οδηγήσει την ανάπτυξη μιας νέας γενιάς τέτοιων συστημάτων. Σε αυτή την εργασία εφαρμόζουμε μοντέρνες τεχνολογίες επεξεργασίας ροών προκειμένου να βελτιώσουμε την τεχνολογική στάθμιση σε δύο εφαρμογές που βασίζονται στον εντοπισμό στο φυσικό χώρο σε πραγματικό χρόνο: την δημιουργία του προφίλ κίνησης των επισκεπτών σε εκθεσιακούς χώρους και την πρόβλεψη παραβίασης συμφωνιών επιπέδου υπηρεσίας στον τομέα των μέσων μαζικής μεταφοράς. Προκειμένου να εξασφαλίσουμε πως τα κλιμακώσιμα συστήματα επεξεργασίας ροών μπορούν να προσαρμοστούν σε αλλαγές του φόρτου με αυτόματη προσαρμογή των πόρων τους, σε αυτή την εργασία υλοποιούμε ένα μηχανισμό με τον οποίο τα συστήματα επεξεργασίας ροών μπορούν να κλιμακωθούν δυναμικά ακόμα κι όταν αυτή η δυνατότητα δεν υποστηρίζεται από τα ίδια τα συστήματα και την υποκείμενη πλατφόρμα διαχείρισης πόρων.

Η πρώτη εφαρμογή των τεχνολογιών επεξεργασίας ροών δεδομένων σε συστήματα ανάλυσης δεδομένων σε πραγματικό χρόνο είναι στην ανάπτυξη δυναμικών προφίλ της συμπεριφοράς των επισκεπτών εκθεσιακών χώρων με βάση την κίνηση τους στο φυσικό χώρο. Η παρούσα εργασία εφαρμόζει τεχνολογίες ροών δεδομένων για την υποστήριξη θεωριών συμπεριφοράς από τις κοινωνικές επιστήμες και για την συλλογή πληροφορίας για πιθανά ενδιαφέροντα των χρηστών. Τέτοια προφίλ μπορούν να χρησιμοποιηθούν για την παραγωγή προτάσεων για τους επισκέπτες σχετικά με το ποια εκθέματα να επισκεφτούν, ποιο είναι το περιεχόμενο που θα ενδιέφερε περισσότερο το χρήστη ή για τη σχεδίαση προσωποποιημένων ερωτηματολογίων. Η δεύτερη εφαρμογή των τεχνολογιών επεξεργασίας ροών σε συστήματα ανάλυσης δεδομένων σε πραγματικό χρόνο είναι στην εκπαίδευση κατάλληλων μοντέλων για την πρόβλεψη οχημάτων που είναι πιθανό να παραβιάσουν τους στόχους που ορίζει η συμφωνία επιπέδου υπηρεσίας τους κατά τη διάρκεια δρομολογίου τους. Σε αυτή την εργασία επεκτείνουμε υπάρχουσες τεχνικές πρόβλεψης καθυστέρησης με τη δυνατότητα να εφαρμοστούν προβλέψεις σε πραγματικό χρόνο.

Σε αυτη την εργασία εξετάζουμε τις κατάλληλες ρυθμίσεις συστήματος για την επίτευξη κλιμακώσιμης και προσαρμόσιμης ανάλυσης δεδομένων ελέγχοντας την επίδραση διαφόρων παραμέτρων στο συνολικό σύστημα επεξεργασίας ροών δεδομένων. Ακόμα και σε περιπτώσεις που η δυναμική κλιμακωσιμότητα δεν υποστηρίζεται από την πλατφόρμα επεξεργασίας ροών, παρουσιάζουμε μια τεχνική που μπορεί να επιτύχει οριζόντια κλιμακωσιμότητα με μικρό χρόνο μη διαθεσιμότητας κατά την προσαρμογή.

## Acknowledgements

First of all, I would like to thank my supervisor Prof. Konstantinos Magoutis, for his help throughout this thesis and his guidance. I would also like to thank PhD candidate Antonis Papaioannou for his willing help when I had trouble during my experiments.

Thanks are also extended to all the academic faculty of the Dept. of Computer Science in Crete for consistently sharing their knowledge during my graduate studies. I would specially thank Prof. Dimitris Plexousakis and Prof. Polyvios Pratikakis for serving in the examining committee of this thesis.

I am also thankful to my colleagues for the creative collaboration and the long stimulating conversations in and out of class. A special expression of gratitude goes to my friends for their continuous support and patience during this journey for being my sounding board when things got tough.

And of course, special thanks to my family for their support throughout both my undergraduate and graduate studies. It is their endless support and love that carried me thus far.

*To my family, for all that I am and hope to be.*

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Throughout the centuries societies always placed increased value in certain types of goods. Civilizations used to rise or fall based on their ability to acquire and utilize various commodities. Once upon a time, people fought to collect metals, gold, diamonds, or oil. The need of men to exploit all these commodities sparked innovation and exploration throughout the ages and economies and businesses have been built, based on what the "hottest" commodity was at a certain time.

*Today's hottest commodity is data.* However, the value of data, as with most of the other valuable commodities, is not in the raw data themselves, but in the insights and knowledge that can be derived from them, after refinement. The applications we use every day produce an immense amount of data that will only keep growing. According to Statista, 59 zettabytes of data were created, captured, copied or consumed worldwide in 2020 (Figure 1.1) and that number is only expected to increase in the coming years [17]. We need to be able to process all this information and make decisions fast. Therefore we need applications and tools that allow us to access our information fast and respond in real-time. In this thesis we tackled two applications with real-time response needs.

The first application is centered on using streaming techniques to develop dynamic profiles on museum visitors based on their movements in physical space. We build upon behavioral theories drawn from social science studies, to build models from information we get by tracking visitors' movements, identifying patterns in them. We leverage these profiles to produce recommendations for the visitors about exhibits they should visit, to decide the best content to present to them, and to design dynamic questionnaires to be answered at the end of their visit.

In this type of application we need fast response time from the system. The visitor is moving through the exhibition space and interacting with the content they see on their application. Delays in response time can affect the quality of experience of the user, since their time may be limited and they may have a short attention span.

The other application we tackled was in a different context, yet still based on stream-processing location-based information. We went into the area of mass

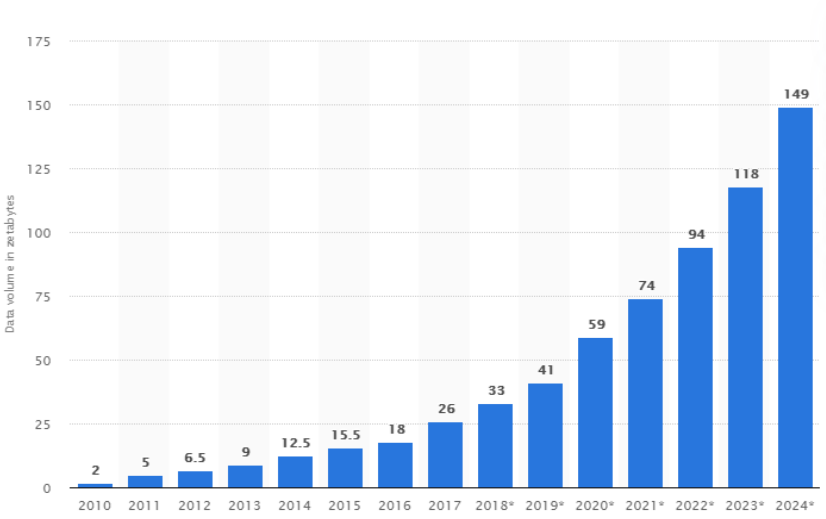



Figure 1.1: Volume of data created/captured/copied/consumed worldwide from 2010 to 2024 (*in zettabytes*) [17]

transit and tried to predict when a service-level agreement and related objectives (SLA, SLOs) were going to be violated. SLOs are performance targets, parts of an agreement that the transport operator can make with the city (essentially an upper bound on deviations from published scheduled routes on a statistical basis) and they are obligated to keep it.

We designed a system that based on the availability of data about mass-transit schedules and trips, can predict if a bus is likely to violate its SLO at any point in time in its route. To do that, every time a bus reaches a *checkpoint stop* (to be defined in Chapter 4) in its route, it should run a regression to predict how late it would arrive at the final stop. If this delay is high enough a warning can be produced and pushed to a live dashboard designed with Microsoft's PowerBI tool. In addition to this, we produce other types of warnings for other violations. To understand what is possible based on openly-available data, we examined an open GTFS-formatted [8, 9] dataset, describing static and real-time information collected from public bus transportation traffic, via an open-data repository [12] as training and testing sets. We describe our conclusions of this study in Chapter 4.

In order to implement these streaming applications in a scalable fashion, we used Spark Structured Streaming [16, 22], a stream-processing engine that is part of the Spark ecosystem and runs on the SparkSQL optimized engine. For the ingestion of data we chose the Kafka [1] scalable messaging system, for its inter-operability benefits, its reliability and the throughput it can achieve.

In the last part of this thesis, we examine the efficiency and scalability of a pipeline comprising of Spark Structured Streaming and Kafka in distributed setups. We examine how our application scales on the system with the default

settings and present a detailed analysis of how we managed to improve our performance. We examine various parameters for both systems and give an overview of how they affected the scalability of our streaming application.

Finally, as Kubernetes is a relatively recent addition to the family of Spark cluster resource managers (RMs), the Kubernetes-based Spark cluster RM is still missing support for dynamic elasticity. On most of the other cluster managers, one can choose to let the system request and release resources as needed. This is good for a streaming application, since it may be never-ending (for infinite streams) and we want to be able to adjust system resources to the level of load in real time. In this work, we leverage Kafka's ability to replay the input stream on demand and Spark's fault tolerance support (by storing set offsets until a job is committed) and employ a method used by other streaming systems to achieve elasticity in similar scenarios (Flink [26]), by terminating the application and restarting it with the new configuration. We study the cost of this transition and present our findings.

In summary, our contributions in this thesis are:

- A dynamic profile building approach, using IoT-driven location tracking, providing technological foundations to behavioral models derived from social sciences.
- The application of streaming technologies to public-transportation data, to augment previous offline prediction approaches with real-time prediction of service-level violations.
- An extensive examination of performance optimization parameters for the Spark Structured Streaming and Kafka tools.
- The demonstration of how to retrofit out-of-the-box dynamic scalability in a scalable stream-processing platform within a distributed infrastructure.

The structure of this thesis is as follows. In Chapter 2 we start by describing necessary background information on key notions and technologies underlying this thesis, such as data stream processing, Apache Spark, Spark Structured Streaming, and Apache Kafka; we also present a focused comparison of the Apache Spark streaming platform with other state-of-the-art SPEs. In Chapter 3 we present our work on the dynamic creation of museum visitor profiles and describe their potential uses in personalized recommendations. In Chapter 4 we present our second streaming application on data-driven modeling and real-time prediction of SLO violations in mass transit using open GTFS datasets. Finally, in Chapter 5 we present our results on tuning and optimization to demonstrate the platform's scalability and our support for achieving and demonstrating elasticity in the underlying stream processing platform. We implement a straightforward way to achieve elasticity via use of fault-tolerance mechanisms, without relying on explicit support for dynamically varying Spark resource allocations by the underlying resource manager.

# Chapter 2

# Background

In this chapter we will cover basic concepts and tools for this work. We will start by explaining what stream processing is and define basic terminology. Next we will present a brief overview of how a Spark cluster is structured and how it operates and define the terms and language we are going to use from now on in our experiments. Then we are going to describe Spark Structured Streaming which is the Stream Processing Engine (SPE) we use and is part of the Spark ecosystem. We give a brief overview of Kafka, which we use in our applications to ingest and output data. Finally, we are going to give a quick summary of other state of the art SPEs and message brokering systems and how they relate to each other.

## 2.1 Stream processing basics

Traditionally, applications that need to perform some type of data processing, have access to a database or a distributed filesystem, from which they can consume the data in its entirety and perform computations on it. This approach is also referred to as **batch processing**. Batch processing splits the data into portions (batches) and performs all the computations on each batch one at a time. Modern applications need a more real time (or near real time) solution.

Distributed stream-processing systems have become particularly important in recent years due to the needs of Internet services such as Google, Twitter, and LinkedIn to process data produced by sources such as Internet users, production systems, and sensors (e.g. sensors that are already installed on vehicles) in real-time for immediate (in near real-time) response allowing for corrective actions. Streaming applications may use stateless operators that operate on incoming data without any notion of memory of previously ingested data (e.g., for producing alerts when a value exceeds a certain threshold); they may also use stateful operators, such as window and join, as part of continuous queries that accumulate large amounts of state in different forms over time.

Data streams are finite or infinite sequences of data (called tuples) that comprise feature values based on a data schema. Stream processing is often expressed

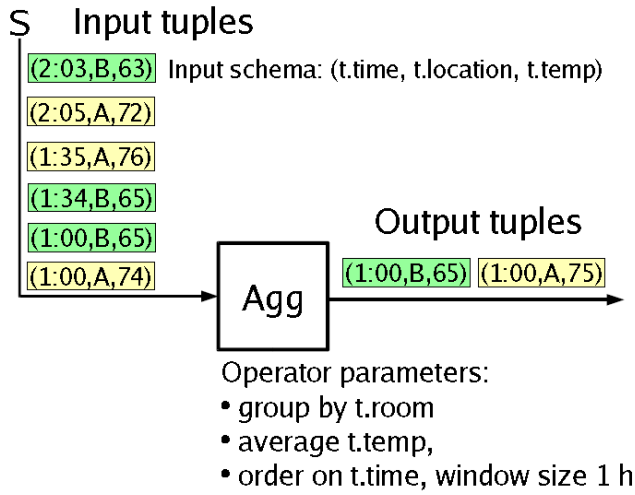



Figure 2.1: Example of a stream-processing (single-pass) operator [4].

as a graph of operators, including stateless and stateful operators. An example of a stateful operator appears in Figure 2.1, in which an aggregate operator computes the average temperature per room in a building over tuples accumulated within a 1-hour time window. Note the schema of input tuples (sensor temperature measurements in different rooms at different times) and output tuples (average temperature per room in the last hour) in this example.

Streaming windows can be defined over time (e.g., window remains open for a certain time interval), based on number of tuples (e.g., window remains open until a certain number of tuples enter it), or other specifications. For a complete background on stream processing and related work on this space we refer to a recent survey in this space [36].

In Chapter 3 and 4 we describe two streaming applications that we designed on two different datasets.

## 2.2 Apache Spark

### 2.2.1 Basic concepts

Spark Structured Streaming is part of the Spark ecosystem so to better understand how a scalable single-pass application is deployed within Spark, we describe the basic components of a Spark cluster and how an application is submitted and executed on the cluster [15]. Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in a main program (called the driver program), shown in Figure 2.2. Every Spark cluster needs a

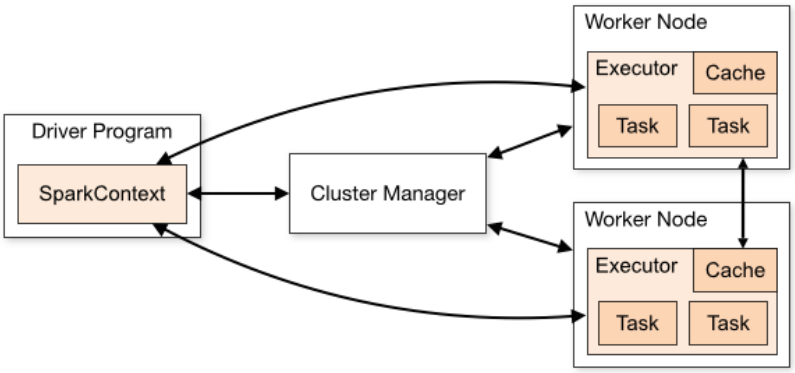



Figure 2.2: Spark architecture [15]

cluster manager (Kubernetes in our case), which allocates resources for applications. SparkContext connects to the cluster manager and requests the resources it needs. Once connected, Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for the application. Next, it sends the application code (.jar) to the executors. Finally, SparkContext sends tasks to the executors to run.

Every application is assigned its own executor processes, which stay up for the entire time the application is running and they run multiple task threads. This way applications are isolated from each other (each driver schedules its own tasks and tasks from different applications run in different JVMs). However, Spark applications (instances of SparkContext) that need to share data must do that through an external storage system.

Spark is agnostic to the underlying cluster manager, which just provides the executors. The driver program must listen for and accept incoming connections from its executors throughout its lifetime. Furthermore, because the driver schedules tasks on the cluster, it should be run close to the worker nodes, preferably on the same LAN. Sending requests to the cluster remotely can be better done by connecting to the driver and have it submit operations from nearby, rather than running a driver far away from worker nodes and increasing latencies (deploy-mode). Table 2.1 lists definitions of key terms [15] from the Spark documentation discussed above.

### 2.2.2 Job scheduling

Each application (instance of SparkContext) runs with an independent set of executor JVMs that run tasks and store data for that application. The cluster managers that Spark runs on, provide facilities for scheduling across applications. Within each application multiple jobs (spark actions) may be running concurrently, if they were submitted by different threads. This is common, if the application is serving requests over the network.

Table 2.1: Basic Spark terminology as defined in official documentation [15].

| Term | Meaning |
|---|---|
| `Application` | User program built on Spark. Consists of a driver program and the executors on the cluster. |
| `Application jar` | A jar containing the user's Spark application. In some cases, users will want to create an "uber jar" containing their application along with its dependencies. The user's jar should never include Hadoop or Spark libraries, however, these will be added at runtime. |
| `Driver program` | The process running the main() function of the application and creating the SparkContext |
| `Cluster manager` | An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, Kubernetes, YARN) |
| `Deploy mode` | Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster. |
| `Worker node` | Any node that can run application code in the cluster. |
| `Executor` | A process launched for an application on a worker node that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors. |
| `Task` | A unit of work that will be sent to one executor |
| `Job` | A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect). |
| `Stage` | Each job gets divided into smaller sets of tasks called stages that depend on each other (similar to the map and reduce stages in MapReduce). |

The simplest option for scheduling jobs across applications, available on all cluster managers, is static partitioning of resources. With this approach, each application is given a maximum amount of resources it can use and holds onto them through termination. All executors are created at start-time and they are allocated the requested resources. The resources an application uses can be limited by appropriate values in the configuration, such as:

- spark.cores.max
- spark.deploy.defaultCores
- spark.executor.memory

With Spark, it is also possible to dynamically adjust the resources used by an application based on the workload. This feature is particularly useful if multiple

applications share resources in a Spark cluster. This feature is not yet available for the Kubernetes cluster manager, but it will be integrated in a future release.

At a high level, Spark should dynamically adjusts the number of executors as needed, releasing executors when no longer used and allocating them again when they are needed. A Spark application with dynamic allocation enabled requests additional executors when it has pending tasks waiting to be scheduled. This condition implies that the existing set of executors cannot simultaneously saturate all tasks submitted but not yet finished.

Spark has 2 types of operations: transformations and actions. Transformations perform some alteration on the data (e.g. map), while actions return a result to the driver (e.g. reduce). A "job" is launched after a Spark action (e.g. save, collect) and any tasks that need to run to evaluate that action. By default, Spark's scheduler runs jobs in FIFO fashion. Each job is divided into "stages" (e.g. map and reduce phases), and the first job gets priority on all available resources while its stages have tasks to launch, then the second job gets priority, etc. If the jobs at the head of the queue do not need to use the entire cluster, later jobs can start to run, but if the jobs at the head of the queue are large, then later jobs may be delayed.

The division to job stages depends on how they can be separately carried out (mainly on shuffle boundaries). Stages are then divided into tasks. Tasks are the smallest unit of work that has to be done by the executor. Tasks refer to the number of partitions we have in the dataset we want to perform the action on. If the source is not partitioned then Spark will perform partitioning as it sees fit (default 200 partitions), otherwise it will follow the partitioning indicated by the source.

## 2.3 Spark Structured Streaming

For the applications we developed as part of this thesis we used Spark Structured Streaming [22], an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of real-time data streams [16]. Spark Structured Streaming can be connected to several types of data sources for ingesting data, such as files, Kafka, Kinesis, or TCP sockets. Our queries can include high-level functions such as map (map by state), reduce (reduce by state), join and window. Results and processed data can be written out to filesystems, databases, and live dashboards (Figure 2.3).

An important feature of Spark Structured Streaming is that one can express a streaming computation similar to the way a batch computation would be expressed on static data. Adapting to the nature of continuously flowing streaming data, the Spark SQL engine runs incrementally and continuously updates the final result with every new micro-batch of input coming in in the form of streaming data.

Streaming aggregations, event-time windows, stream-to-batch joins, etc. are expressed using the Dataset/Dataframe API [16] in Scala, Java, Python or R,


Figure 2.3: Spark Structured Streaming [16]

executing the computation on the optimized Spark SQL engine. Spark Structured Streaming ensures several key properties for robustness, namely end-to-end exactly-once fault-tolerance guarantees through checkpointing and write-ahead logs. The design and heritage of Spark Structured Streaming within the Spark ecosystem ensures that these properties are achieved without the user having to reason about streaming.

Structured Streaming queries are processed using a micro-batch processing engine based on the core design of the Spark approach [22], which processes data streams as a series of small batch jobs thereby achieving end-to-end latencies of around 100 milliseconds with exactly-once fault-tolerance guarantees. Since Spark 2.3, a new lower-latency processing mode called Continuous Processing (similar to the mode of processing of other popular stream-processing systems, such as Flink [26]) has been introduced, which achieves end-to-end latencies as low as 1 millisecond. However, this lower latency comes with weaker atomicity guarantees, namely at-least-once rather than exactly-once semantics (more information in section 4.3). An important feature is that the user can choose the mode of operation (micro-batch vs. continuous processing) without changing the Dataset/DataFrame operations in streaming queries. Continuous processing is still limited and can only perform basic operations, but in future releases more functionality will be added to it.

In an example (depicted in Figures 2.4 and 2.5 [16]), every data item arriving on an input stream is represented as a new row appended to an “Input” table.

A query on the input will generate a ”Result” table. Every trigger interval (say, every 1 second), new rows get appended to the Input table, which eventually will update the Result Table. Whenever the result table gets updated, new result rows should be produced and written to an external sink. The ”Output” is defined as what gets written out to the external storage. The output can be defined in a number of different modes, depending on whether we want all results to be written out to the sink on each update, or we want this process to operate incrementally [16].

Apache Spark Structured Streaming does not materialize the entire table. Rather, it reads the latest available data from the streaming data source, processes it incrementally to update the result, and then discards the source data. It only keeps around the minimal intermediate state data as required to update the result (e.g. intermediate counts in the earlier example).

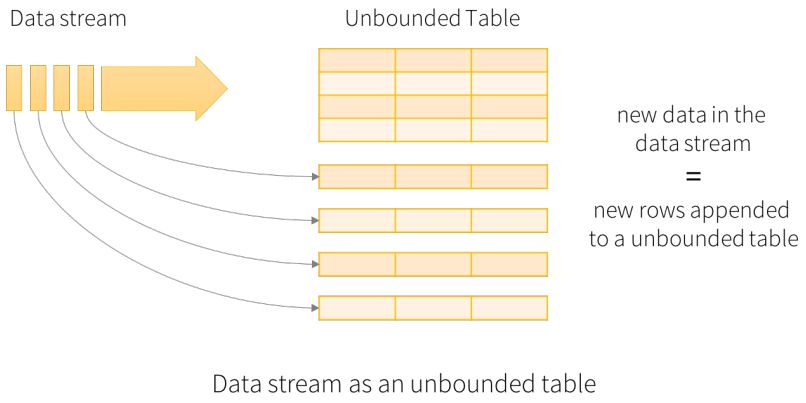



Figure 2.4: Data stream as an unbounded table [16]

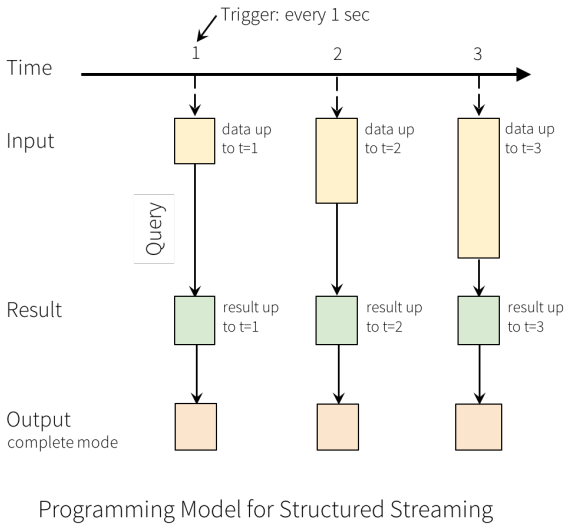



Figure 2.5: Programming model for Spark Structured Streaming [16]

### 2.3.1 Handling event-time and late data

Event-time is an important concept in streaming data and is typically the time that is embedded in the data itself (when the data is produced, rather than when it is ingested into the stream-processing engine). This event-time typically expressed as a column value in each row, with each event from the devices corresponding to a row in the table.

Window-based aggregations (e.g. number of events every minute) are a type of grouping and aggregation on the event-time column – each time window is a group and each row can belong to multiple windows/groups. Therefore, such event-time-window-based aggregation queries can be defined consistently on both a static dataset (e.g. from collected device events logs) as well as on a data stream, simplifying things for users.

Data that may be arriving later than expected based on its event-time can be handled in the updating of the Result Table. Updating old aggregates is possible when there is late data. As many aggregates with increase the size of intermediate state data, it is possible to clean them up after some time to limit the size of that state.

### 2.3.2 Fault tolerance semantics

The goal of several stream processing systems is to achieve exactly-once semantics, namely to record the effect of any incoming tuple just (exactly) once in the state and output of the system in spite of system or network failure. On the other hand, at-least-once semantics allow to record the effect of any incoming tuple more than one time.

Spark Structured Streaming has the goal of exactly-once semantics. To achieve that, Structured Streaming sources, sinks and execution engine reliably track the exact progress of the processing so that it can restart after failure and reprocess without missing or duplicating any inputs. Every streaming source is assumed to have offsets (similar to Kafka offsets) to track the read position in the stream. The engine uses checkpointing and write-ahead logs to record the offset range of the data being processed in each trigger. The streaming sinks are designed to be idempotent for handling reprocessing, ensuring (along with replaying of input) end-to-end exactly-once semantics under any failure.

## 2.4 Scalable data ingestion with Kafka

Apache Kafka [1] is an open source platform designed for the processing of data streams by Linkedln and offered to the Apache Software Foundation. It is a scalable, fault-tolerant and fast publish-subscribe messaging system, similar to a messaging queue. It provides strict message ordering guarantees end-to-end and it is based on push logic, were the applications push events into topics and consumers consume at their own pace.

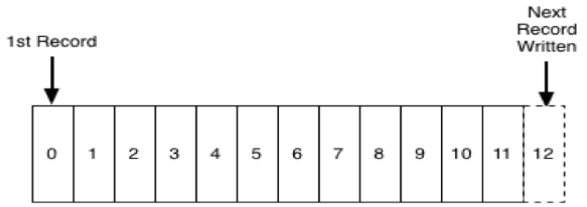



Figure 2.6: Kafka records log

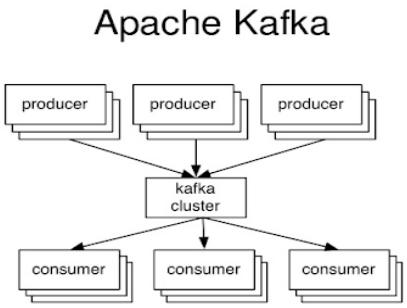



Figure 2.7: Kafka's producer-consumer model consumer. [1]

The main use of Kafka is in data pipelines, in order to transfer data between systems or applications reliably. Since it can connect to many different systems due to a plethora of connectors designed by third parties, it acts as a bridge between applications, offering interoperability. Another function it has is as a stream processing tool, by providing an API to perform Extract-Transform-Load (ETL) operations on data streams. Finally, thanks to its ability to store data in order, it can be used as a buffer for big data ingest in applications that cannot keep up with the incoming flow at peak times.

Kafka is a cluster that can be scaled horizontally achieve near linear clustering performance and its cluster nodes are called *brokers*. It is based on event logic instead of tables and data entries are called events, records or messages. **Messages** are timestamped immutable key-value pairs that can only be appended to the existing messages and are persisted to disk (Figure 2.6). We have **producers**, who write messages to a broker and **consumers**, who need to read the messages from a broker and perform whatever operations they need on them, store them or forward them to the next system (Figure 2.7).

Each broker has a collection of **topics**. A topic is a logical name for one or more **partitions** (Figure 2.8). So we name a topic and on creation we say how many partition this topic will have. Partitions can be added after creation, but they cannot be deleted and all the messages that were already in the existing partitions

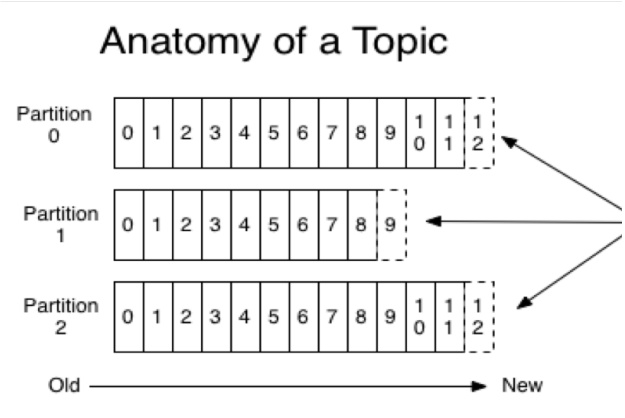



Figure 2.8: Description of a Kafka topic.

will not be redistributed to the new ones. Ordering happens at partition level, so all messages inside a partition are ordered by event-time and messages are split into topics by their key. If the key is null or the same, then messages are assigned to topics in a round robin manner. Replication also happens at the level of the partition.

Each message inside a partition has an **offset**, which is used to keep track of the ordering in the partitions. Offsets are unique, sequential ids (per partition). Consumers also need to track offsets, so the first time they connect to Kafka they will start at offset 0, but at any following time they can start from 0, from the latest message not yet read or from any message between those two points given its offset. This way we can replay the events in the topic as long as the retention period is not up and we can have consumers with different speeds.

## 2.5 An overview of data streaming technologies

Stream processing has been an active field of study for more than 20 years, leading to many stream processing solutions available to users today. In this section we are taking a high level look on some well-known stream processing engines.

The idea of streaming queries was first introduced in 1992 [60], with the core concepts and ideas originating from the area of database systems, leading to some early stream processing systems (TelegraphCQ [27], Aurora/Borealis [28], STREAM [21], etc). These early systems introduced the core ideas of streaming queries and exposed various challenges, like fault-tolerance and availability issues, sliding window aggregations, load balancing and more. The next batch of streaming systems focused mostly on streaming window queries and Complex Event Programming (CEP) (ESPER [7], Oracle CQL/CEP [6], etc) on ordered event streams.

With the rise of cloud computing and the introduction of MapReduce [33], we move to the more modern approaches, like Twitter Storm [45], Naiad[50], Spark

Streaming [64] /Spark Structured Streaming [22] and Apache FLink [26] and more. These systems performed distributed data-parallel computations based on dataflow graphs executed on distributed clusters. Google Dataflow [20] re-introduced the idea of out-of-order processing, proposing a parallel processing model for batch and streaming computations. Today's stream processing engines offer fault-tolerant solutions that scale-out (and up) and can handle massive out-of-order streams.

The dataflow streaming model [20], which is followed by many state of the art SPEs, only requires the presence of a timestamp on the tuples of the input stream. There are three modes of operation: (a) *event time* processing, which uses the time that the events were generated at the sources, (b) *processing-time* which uses the time the events were processed by the SPE and (c) *ingestion-time* which is the time the events arrived at the system. Modern SPEs can handle any time of input stream and the burden of handling semantics and logic falls on the developer of the streaming application.

Apache Storm [45] is an open-sourced system, developed in 2011, mainly written in Java and Closure. It is a low-latency, fault-tolerant and distributed system. Persistent queries in Storm are called topologies, which are direacted graphs with computations as the verices and the data flow as the edges. Storm uses Zookeeper to coordinate its' cluster nodes. It only gives at-least-once guarantees and implements a fail-fast approach in case a node gets killed to avoid any topology execution interruptions. Compared to other SPEs, Storm has lower throughput and lower latency.

Apache Flink [26], is an open-source system for big data analytics, mainly written in Java and Scala. It can guarantee exactly-once processing and can supports both streaming and batch queries (although it is not really as popular for batch applications). Flink uses a master-worker pattern with a Job Manger and one or more Task Managers (similar to the Driver and Executor nodes in Spark). Flink is considered to have a very low latency, but it can also achieve high throughput.

We have already discussed Spark in previous sections. Spark contains two streaming libraries, Spark Streaming [64] and Spark Structured Streaming, which is an evolution of the first. Spark Streaming is based on resilient distributed datasets (RDDs), which are a memory abstraction, while Structured Streaming uses dataframes. Structured Streaming can achieve very low latencies using micro-batching with very high throughput and in recent versions can even support continuous queries with limited functionality (future version will expand on this). Spark also gives exactly-once semantics.

Apache Samza [51] is another distributed stream processing system mainly written in Scala and Java that treats messages as streams. Samza has a relatively high throughput and slightly increased latency compared to Storm. Samza is usually run with YARN [61] as the cluster manager and Kafka [1] for the streaming layer.

There have been a number of works comparing all these stream processing systems [34, 38, 41]. Regarding latency and throughput there are some works that benchmark these systems, although not all together [44, 3]. But since latency can

depend very much on the streaming application the results can vary a lot from one study to another. There are many questions to consider when picking your SPE, such as:

- How important is latency?
- Is high throughput critical?
- Do I need batch as well as stream processing?
- What fault-tolerance semantics do I need in case of failure?

In the following sections we are going to present two streaming applications we designed using Spark Structured Streaming [22, 16]. Related work for each application area is included in the corresponding chapters.

# Chapter 3

# Dynamic profiling of museum visitors

In this chapter we describe a system for personalized recommendations as part of a smart IoT-driven tour guide platform. The system builds user profiles out of historical localization information (movement of users in space within a museum) and indications of their interests, inferred through interaction with digital content, and time spent around exhibits.

The presented system leverages data-streaming technologies for online training and interactive response in coordination with a commercial IoT platform and mobile application. Our evaluation focuses on validating the profile-building and clustering parts of the platform using an experimental testbed that allows for testing under controlled conditions.

## 3.1 Introduction

We all visit museums or archaeological sites from time to time to appreciate culture, to learn, and to overall have a pleasant experience. However museums are usually large spaces that are hard to navigate, with the volume of exhibits and information available sometimes being overwhelming to the visitor. For this reason most museums today provide guided tours, traditionally via licensed staff, and more recently by offering electronic tour guides that help visitors navigate the space, often providing additional information about the exhibits. These tools are undoubtedly helpful to many visitors, but they are typically generic.

ProxiTour [31] is an IoT-driven platform for digital tours, whose aim is to create a more custom-made experience for the visitors. By collecting information produced in real time, such as the visitor's interaction with the space and the exhibits, ProxiTour aims to create on the fly, a custom-made profile for each user. Based on that profile, which is updated during the visit, the platform can suggest exhibits and sights likely to interest the user, improving their experience.

ProxiTour relies on Bluetooth beacons embedded within places of interest (e.g.

museums, historical sites) for localization and for correlating visitor location with exhibits. Visitor tracking begins from the point they enter the space with their mobile device. Every time they come within range of a beacon, they receive recommended information (audio and visual) based on their position, nearby exhibits, and their profile so far. In outdoor spaces, the GPS on mobile devices can also be used to track visitor location. The interaction of visitors with the provided digital content and their movements in space can be interpreted as indirect feedback for the recommendations, enabling adaption.

In this section, we present a novel system for providing personalized recommendations within the context of a smart IoT-driven tour guide platform [31]. Our contributions in this work are:

- Creation of user profiles, based on data collected in real time from a Bluetooth beacon-based localization service and from interaction with digital content through an end-user mobile application
- Creating groups of user profiles with similar interests aiming to use such groups for recommendation purposes
- An implementation, drawing information from state-of-the-art IoT technologies, for localization and data collection.
- Online analysis, using Spark Structured Streaming, to perform data analysis live (online), in a scalable manner
- Experimental validation of the profile-building and clustering parts using a deployment of ProxiTour in a controlled physical environment

The system aims to advance over standard approaches for e-guides that are either unaware of the history of users' movement in space and in relation to exhibits, or have very basic information about it, often requiring significant user involvement (such as manually browsing through digital content or scanning an NFC tag next to exhibits).

## 3.2 Related work

In the past few years e-guides have started becoming a more and more prominent fixture in visitors' museum experience. The more primitive version of these electronic guides were purely based on user input. The visitor had to navigate some application, find the room they were in, pick some exhibit from a list of available items and then they could access any available information (textual, audio or visual) available for that item. This process while undoubtedly foolproof, could be a little tedious.

The next step in the evolution of e-guides, uses QR codes placed close to the exhibits. The visitor had to use their mobile device to scan the code and access any information available this way. Both of these approaches work and the user

will get all the desired information with minimum errors, but we can only track them based on their interactions with the device and the input given. So if they do not press a button or scan some QR code we have no idea were they are in the exhibition space.

User tracking is a well researched area and there have been many approaches studied in the literature. In [63] Steven S. Yalowitz and Kerry Bronnenkant discuss the long history of timing and tracking museum visitors and which variables should be tracked (stopping behavior, situational variables and more). How visitors move in an exhibition space and which exhibits they tend to gravitate towards, is a topic of great interest to museum curators. The main reason for this, is that it is a very efficient way to evaluate an exhibition and gather information about the target exhibition for future reference in design choices.

This information can be invaluable to museums, because like any business they need to attract customers and they want to provide a service that has value. Therefore much work has been focused on gathering this type of information by various means. In outdoors spaces GPS devices can be used to track visitors location, similar to vehicle tracking. In indoor spaces, the most common approach is to use wireless sensor networks. However, tracking and analyzing the movement of a user indoors, is not a trivial issue. The complexity of the space, the accuracy of the sensors and the different types of data collected can be very challenging.

So as we move on from the more outdated earlier solutions, to the more modern approach, we get some state of the art systems, which already use Bluetooth beacons to track visitors. By placing beacons around the museum space they use triangulation to figure out the location of the visitor and track them while they move through the space, eliminating the need for user interaction as a means of user tracking. ProxiTour, the personalized touring platform on which the system presented in this paper builds upon was recently introduced by Chronarakis et al. [31]. Our approach utilizes this form of user tracking to build timeseries of the visitors' movements. We can tell where they are, how they move through the space, how long they spend on exhibits and a plethora of other information.

Other approaches use video to track the visitors' movements [57], or specialized equipment to track the visitor's posture and direction when standing in front of exhibits [54] as well as other methods. While these approaches may perform well, they have an inherent complexity that drives up the operational costs and the costs of equipment and analysis. Bluetooth beacons are cheaper and therefore provide a more feasible solution for many museums.

Aside from providing useful information to museum curators, this rich information gathered can be used to provide recommendations to the users. The field of personalized recommendations is a very well researched area of study and there has been some very interesting work in relation to exhibition spaces in particular. The goal of these systems is to figure out the best subset of options from a set for some particular user. They can help reduce buyer's anxiety, when faced with overwhelming choices and in our case, they can help predict what our visitor would prefer, based on their profile.

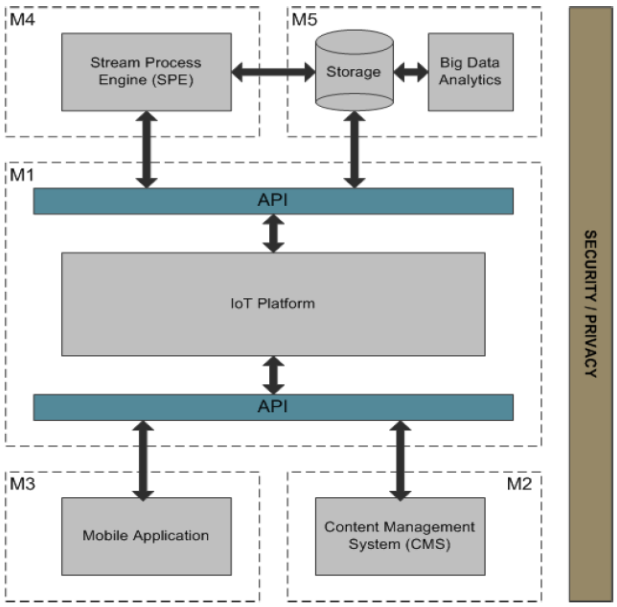



Figure 3.1: ProxiTour system architecure on a high level [31]

Approaches to information filtering include collaborative filtering, content-based filtering, and hybrid filtering [24, 25, 52, 55]. **Collaborative filtering** uses users with similar tastes to make recommendations. It is the most commonly used approach and the most mature. The system constructs tables of similar items offline and then uses those tables to make online suggestions, based on the user's history. **Content-based filtering** predicts the user's interests, based solely on the current user and ignoring information collected from other users. Finally, **hybrid filtering** combines two or more filtering techniques, in order to increase the accuracy and performance of the system and overcome some of the issues inherent in each approach. Our approach to implementing a hybrid recommendation system is described in Section 3.5.

## 3.3 Architecture

In this section we take a deeper look into the architecture of ProxiTour and the various design decisions concerning the handling and the analysis of the collected user data.

ProxiTour has a modular architecture and consists of five subsystems. The subsystems are the IoT platform, the Content Management System (CMS), the Mobile Application, the Stream Processing Engine (SPE), and the Storage and

Big Data Analytics module. In this work we will focus on the Stream Processing Engine and the Big Data analytics parts, but we present the entire system on a high level.

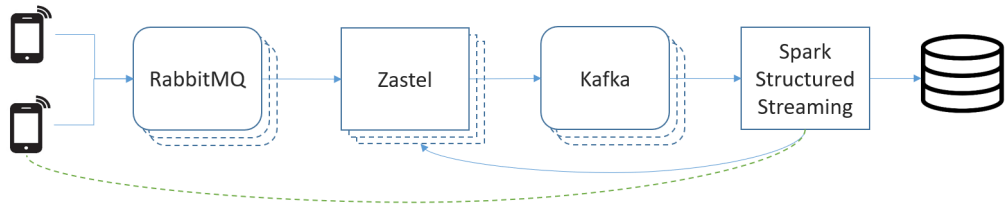



Figure 3.2: Basic workflow for the streaming analytics (M3) and the Big Data analytics (M4)

In Figure 3.1 we can see the modules of the system and how they interact. The IoT platform (M1) receives periodic messages with the approximate location of a visitor inside the space (with an accuracy close to 1-1.5m) from their mobile device (M3). The IoT platform is the central piece in this architecture and taking into account the latest information about the visitor (profile, etc), their location in the exhibit space and the exhibits they are interacting with, it decides which content to fetch from the CMS (M2) to serve to the user's mobile application (M3). Additionally, the IoT platform forwards the information it receives and calculates to the M4 and M5 modules, to feed into the online and offline analytics to update the profiles, calculate statistics and be eventually stored as historical data. These analytics help the IoT platform make more informed decisions on what content to pick from the CMS. In the next section we will go into more detail about the analytics performed on the data.

Now that we have a basic understanding of how the various parts of the system connect and operate together, we can take a closer look at the subsystems that we focused on. Specifically, we worked on the M4 and M5 modules, so lets take a closer look at the workflow concerning those two modules. In Figure 3.2 we describe this workflow. The platform supports a variety of mobile devices (phones, tablets, etc.), via which the users interact with the beacons in the museum and the digital content provided by the platform. The devices communicate events over a RabbitMQ service to a commercial IoT platform, Zastel, whose main functions are the management of location services (beacons, GPS), dynamic data sharing and decision making, managing/monitoring beacons and administrative users and connecting with third party systems and applications through an API.

Relaying events between the IoT platform and a stream-processing service is Apache Kafka, a distributed messaging and streaming platform. Spark Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine [22] that we use to implement our application. We use it to run our analytics on the messages produced by the mobile devices. These messages are structured as json arrays, with the information that is shown in Figure 3.3 in the messages table and an array of message_data objects. Each message_data object contains a key-value pair, with information like the coordinates of the visitor (longitude: x, latitude: y), the visitor's email (user: user@email.com), the selected

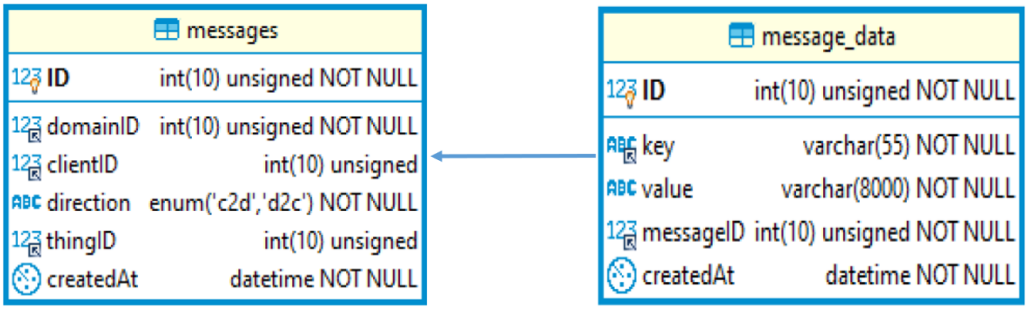



Figure 3.3: Schema of the stored messages.

language (language:gr) and many more.

The results of the analysis sessions are stored in a MongoDB database, to be used in the future for training models or to help us initialize the profile of a user who used the app in the past. During a user's visit, the application will send recommendations as push notifications to their device through Zastel. If the user has not given their permission to store their data, the session information will be deleted after the retention period is over.

Figure 3.4 depicts the experimental space where we run our controlled tests, in order to evaluate the system and our applications. Beacons have been set up in various locations/rooms and associated with certain thematic areas, exhibits, or people. In our evaluation, we examine test scenarios by having users emulate certain behavioral patterns. For more details regarding the architecture of the platform and the beacons used, we refer the reader to [31].

Next we describe the concepts and implementation underlying our profiling applications and our recommendations system.

## 3.4 Towards a recommendations system

The first step towards any recommendation system is building a **profile for its users**. Initially, that profile is based solely on the general information provided by the user[1] (e.g. gender, age) and historical information known for the studied population (e.g. most popular exhibits). As specific users interact with the system and we see how they respond to the digital content and to the recommendations (they seem to follow them or not), the profile can be updated to enrich knowledge of their preferences (preferred exhibits, recommendations taken, etc.), which can lead to more customized suggestions.

In the beginning of the tour, we assume that the initial information about a user is a userID and their age category (e.g., 5-12 (primary school), 13-17 (high school), 18+ (adults), etc.) entered during registration. Users will be asked to enable Bluetooth and Location services on their mobile devices and be informed

[1]We assume GDPR-compliant processes involving explicit user consent.

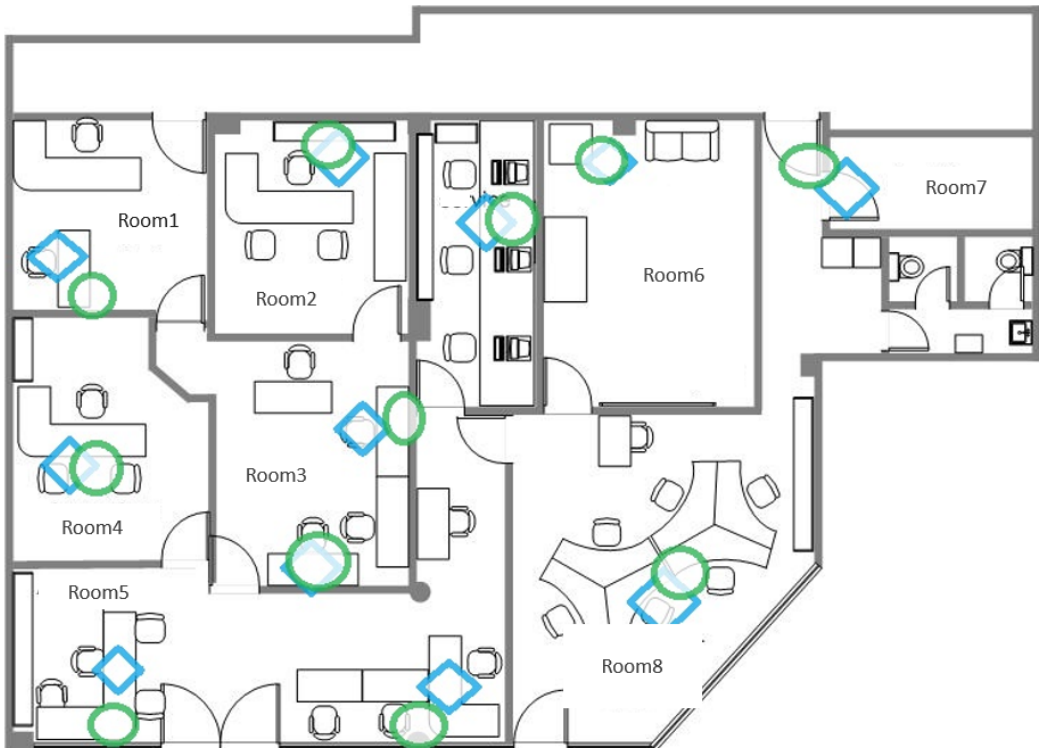



Figure 3.4: Overview of experimental space used for controlled testing. Circles indicate the location of BLE beacons and diamonds the associated exhibits.

that without those features, users cannot be tracked and therefore recommendations will have to be based only on basic information and the general historical information collected from previous visitors. It is our expectation that the anticipation of richer content and better targeted (personalized) services, in conjunction with minimal privacy implications, will entice most users to enable the required support, as has been the case for online shopping and search services.

From the historical information, ProxiTour can create a list with the most popular exhibits based on direct and indirect feedback from the users. The direct and indirect feedback we collect, leads to the creation of **profiles of exhibits** as well. The profiles can be used by the museum curators to make decisions regarding the current or future exhibitions.

Assuming that a user allows location tracking, the tracked characteristics of the current tour can enrich the default content (e.g., the *top-K exhibits list*, typically the well-known exhibits in a museum's collection) with a list of the most interesting exhibits based on their profile (past visits and current tour track). The list of featured exhibits can be dynamically adapted based on the users' current location (e.g. room) and their interaction with the information on the application and the time they spent on each exhibit. If they spent a long time at an exhibit, they most likely found it intriguing, especially if they also interacted with the digital content that was served to them about the same exhibit. If the user visits some exhibits more than once that can also indicate increased interest in relevant exhibits.

### 3.4.1 Visitor profiling

In order for the museum curators to capture the visitors interest and provide useful information, they first have to understand who their audience is. For example groups of students visiting a museum may have different information needs and interests than a professional visiting (archaeologists, historians, etc) or a tourist, due to different starting points and depth of background knowledge.

There are many different ways to differentiate visitors, due to their many characteristics to choose from and the lack of a widely used approach. However there are some common features that we can use as a starting point for our categorization:

- **socio-demographic characteristics:** age, sex, occupation, education, local or non-local resident, etc;
- **museological characteristics:** purpose of the visit (professional, informational), background knowledge of the topic, etc;
- **range characteristics:** individual visitor, (various types of) groups of museum visitors, frequency of visits, timescale of museum visit, etc;
- **psychological or physiological characteristics:** intelligence, memory, imagination, visual, auditive, motoric abilities, etc.

There are many combiantions of the above mentioned characteristics, as well as others that have been used and can be found in literature to categorize museum visitors [32, 35, 37, 40, 49]. In 1991 Veron and Levasseur in their work "Ethnographie de l' exposition: : l'espace, le corps et le sens" [62] claimed that visitors of exhibition spaces can be categorized based on their movement patterns into four categories, using animal movements as a metaphor. Specifically they invoke four different animal movement patterns:

- **Ant:** the ant-visitor tends to spend a long time observing each exhibit and moves close to the walls and the exhibits, avoiding the empty spaces.
- **Fish:** the fish-visitor moves mostly in the empty space, making a few short stops and seeing the majority of the exhibits, but without observing the details about them.
- **Grasshopper:** the grasshopper-visitor seems to prefer certain types/categories of exhibits and visits them, spending a long type studying them. The rest of the time he moves in the empty space.
- **Butterfly:** the butterfly-visitor does not follow the path indicated by the tour he is on, he tends to change direction at random and often stops at exhibits to study them in detail.

These types of theories utilize models from social sciences, as a starting point to build computational models of human behavior that can be used by various systems. Our approach uses a combination of the basic characteristics described and the movement patterns suggested by Leron and Levasseur to build our visitor profiles. These different types of movements can be good indicators of different visitor personality traits and levels of interest. For example, an "ant" is likely to visit most of the exhibits in order, so we can prefetch information for them in order and since. "Ants" are also the most studious, so they will be more likely to answer surveys in the end of their visit and give feedback. On the other hand, "grasshoppers" have clearer preferences, so we can give them more targeted content or direct them to more obscure exhibits matching their interests, or merchandise in the shops. "Fish" can have varying degrees of engagement, as can "butterflies", so we will need to handle them accordingly on a case by case basis.

So, how do we construct the profiles? When a visitor enters the museum space, they pick up a mobile device and they create a new account with their email address. When they register, a new empty visitor profile is generated in the system and a unique account is created for that user. If the visitor already has an account from a previous visit, then some information will be loaded from the historical data (assuming they agreed for their data to be kept in our system), to initialize their profile and our models so we don't start from scratch and the new visitor session will be linked to their existing account.

Every visitor has to fill in a basic profile, with information about their age group (e.g., 5-12 (primary school), 13-17 (high school), 18+ (adults), etc.), purpose for

the visit (educational, professional, recreational), whether they are visiting with a group or alone and if they have some disability (motoric, auditory, visual or other). This information provided by the visitor forms their basic profile. As the visit progresses we will build on that profile and enrich it, based on the visitors actions in the space.

This basic profile could have been more extensive, as we have seen in certain studies, where users are asked to fill in questionnaires at the exhibition space or in advance, but this approach can be very tedious for the visitor and it consumes a lot of their time, which may be limited, affecting their overall experience. Therefore, in an effort to be the least invasive we can be, we chose to sacrifice more information and chose the above mentioned characteristics, after discussion with the museum experts, as the main ones needed to describe a new visitor.

After filling in the basic information, the visitor will be asked to activate the Location Services on their mobile device (Bluetooth or Location) in order for the system to track their movements in the space. At this point the user will receive a notifications, informing them that without activating this service, we cannot track their location and therefore they will receive information and handling based on the generic guess we initially made, based on their basic profile. We believe that visitors will prefer to have the best possible experience that we can provide, using a more personalized approach with a guarantee to be as little invasive as possible in our data collection and as they do every day in the case on online shopping or search engines, they will choose to use this feature.

The visitor can now start their tour. For visitors with location services activated, we can start constructing a timeseries of the messages produced by their movement. This way we can start calculating additional profile characteristics:

- A vector describing the visitor's pattern of movement in the space (We will explain this further in the following paragraphs)
- The sequence in which they visited the exhibits.
- Various statistics about duration of their stay per exhibit (mean, median, max, min, std)
- Various statistics about duration of their stay per room (mean, median, max, min, std)
- The number of exhibits visited per category (the categories have been declared by the ephorate of antiquities)
- The top K exhibits visited, based on the duration of the visitors stay at them (We count revisited exhibits as well for this total)
- Top N exhibit categories, based on the interest in exhibits belonging to them
- The percentage at which they followed the suggested tour path (if one was available)

- The animal whose movement better reprisents the visitor (ant, fish, grasshopper or butterfly, based on the theory of Levon and Levasseur)
- The confidence level for the animal classification (The probability with which the classifier gave this label to the visitor)

This profile is built and changes dynamically, in real time as the visitor moves through the space. The above characteristics are calculated and updated constantly, with every new message we receive about the visitor's movements. When the visit starts a new session starts for the visitor. A state object with the profile information we collect is created and updated, with every new message we receive from the mobile device, as long as the visitor moves through the space.

The movement vector we create describes how the visitor moves in the exhibition space. In particular a visitor has multiple options to view the exhibits. They can move along the edges of the room and close to the exhibits, avoiding the empty space; they can stay in the center most of the time taking in the whole room, but not approaching or they can cross back and forth from the center to the edges. These choices describe different patterns of movement. As we mentioned in the previous section, we used the controlled space of figure 3.4 for experimentation. Knowing the position of the exhibits and some fixed points in each room, we split the room in zones corresponding to center and edges. With every new location message we check which zone the visitor is in. We measured how long the visitor stayed in each zone total, how often they crossed and the mean, median, maximum, minimum and standard deviation between crosses. This is an initial approach and it can be reevaluated and improved when we have access to the museum space for the pilot and can judge the performance with real visitors.

The movement vector of the visitor, the percentage of adherence to the suggested path and the statistics we calculate regarding the duration of their stay at the exhibits, are used to predict which pattern of movement best fits the visitor. The classifier is built using old visitor sessions from the historical data and tries to match the visitor to one of the four movement pattern categories (ant, fish, grasshopper or butterfly, based on the theory of Levon and Levasseur). We tested various machine learning models using nested cross validation and we will discuss this further in the experimental analysis and evaluation section. Periodically, we try to classify the visitor and we update the profile. As we have already mentioned, each animal category possesses certain characteristics that can help us place the visitor in a certain group with other visitors that behave in a similar way.

For example, ant visitors tend to follow the path suggested by the guides an they pay attention to the exhibits and the provided materials. Therefore, there is not much point in trying to direct them to specific exhibits, since they will visit everything with high likelihood. However, we could ask them to answer questionnaires as feedback for their experience at the end of the visit, since they paid so much attention to the space.

In ProxiTour, the movement pattern of each visitor is continuously analyzed to categorize them accordingly. Each user is eventually represented by a vector,

containing the information summary of their profile. This profile can be either stored in a database at the end of their visit, to use in order to initialize their profile in future visits and in an anonymized version for training our models and improving the quality of the provided services or it will be deleted. This depends on what the visitor chooses before exiting the application.

### 3.4.2 Exhibit profiling

In the previous section we described how we construct the visitor profiles. In this section we will discuss how we can use the information we collect from the visitors to create richer exhibit profiles.

Every exhibit has a unique ID number and it is associated with some room, a set of coordinates in the room and some Beacons. For each room/exhibit there is a list of available key words that give a very generic description of that room/exhibit (e.g. time period, associated historical figures, type of exhibit: (painting, artifact, etc), etc). Each keyword is unique and it represents all the exhibits in some category (e.g. Renaissance).

Furthermore, each exhibit is associated with a list of additional information and visual/auditory sources related to it (videos, images, text, etc). All this material is available to the visitor through their mobile device. We can use direct or indirect feedback to evaluate the material and whether the visitor found it interesting. Direct feedback could be a rating on a scale (e.g. 1-5), which is very simple, clear, but tedious for the user.Indirect feedback is derived from the interaction the visitor has with the material. Did they read/watch it and how long did they spend on it? This information is much less invasive and therefore less "annoying" for the visitor, not affecting their quality of experience. However without the visitors direct input, our estimation of their level of engagement is approximate and it is based on the visitor's location, therefore visitor tracking services must be enabled.

Similar to the visitors, the profiles also have a basic profile. This profile is stored in the system and it is constructed with information provided by the Ephoria of Antiquities. The information it provides include:

- The name/id of the exhibit
- The keywords that are assigned to it
- The room they're in
- The id(s) of the beacons they are assigned to
- Its coordinates
- A list with the available material (their ids)

In this profile we add some metrics that we calculate from the information collected in visitor profiles:

- Visits counter
- Various statistics about the duration of the visitor's stay at the exhibit (mean, median, max, min, std)
- Various statistics about the number of visits of the visitor to the exhibit (mean, median, max, min, std) (Some may have revisited certain exhibits)
- Various statistics about whether they chose to access some of the additional material provided (mean, median, max, min, std). In general and for each item available.
- The top categories of visitors that exhibit interest for this exhibit.

With access to all this information the museum curators can extract valuable information, like:

- Which exhibits are the most popular (in general and per category)?
- Which exhibits are the most popular by age group?
- Which rooms attract the most visitors?
- Which exhibits may need to be moved?
- What type of content is more interesting for the visitors?

and of course, many more questions that they can now make more informed decisions on. All this information can be combined to produce reports that answer questions like these (we plan to implement them when we run the pilot) and provide insights to the curators.

## 3.5 Background on recommendation systems

Recommendation systems are a broad and interesting field of study and the goal is to figure out the best subset of options for some particular user, based on their profile. They can help reduce "buyer's anxiety" when faced with overwhelming choices; in our case they can contribute to a more personalized experience for visitors. The three main approaches to this task are: collaborative filtering, content-based filtering, and hybrid filtering [24, 25, 52, 55].

**Collaborative filtering**, the most commonly used and mature approach, groups users with similar tastes to make recommendations. The system constructs tables of similar items offline and then uses those tables to make online suggestions, based on the user's history. In our case, if we wanted to suggest a list of exhibits to the visitor, it would look at the visitors profile and much it to all the visitor profiles on record, then it would suggest to the visitor a list of exhibits that the visitors most similar to him enjoyed the most. The main problems with this

approach is the *cold start* initially, when we do not have many visitors to derive information from, the *sparsity* of the tables and the *scalability* of the solution, since these tables can get very large as more data is collected and the operations between them expensive.

**Content-based** filtering predicts the user's interests, based solely on the current user and ignoring information collected from other users. Specifically, it filters the list of all available objects based on the user's profile. In our case, this would be the visitor's profile and the list of exhibits available. The system would match each exhibit to the visitor's profile and then present the top ranked exhibits to the visitor. The main advantage of this approach is that it does not need any data from other users, which is the main issue with collaborative filtering. The model can identify specific areas of interest for the visitor that may be unusual for the majority and recommend the perhaps less popular items. The main problem with this approach is the *overspecialization* of the model since it can only use the existing interests of the visitor and cannot expand from them. Another important issue is the dependence we have on the *features* we use to represent the user's interests, since they are often constructed by the engineers and require a lot of in depth domain knowledge.

**Hybrid filtering** combines two or more filtering techniques to increase the accuracy and performance of the system and overcome some of the issues inherent in each approach. There are many ways to implement hybrid filtering. The main advantage to this method is that we can solve both the cold start and the overspecialization problem. In our case, we would match the visitor to the history of visitor profiles and extract a list with the top exhibits for those visitors. Then we would filter this list further based on the exhibits that match the specific visitor's profile as is done in content-based filtering and if the items we have gathered are not enough or not ranked very high for the user, we can use content-based filtering on the exhibits table to fill in the rest. This way we avoid the cold start, since content-based filtering does not depend on having many visitor profiles at hand and collaborative filtering will help us avoid overspecialization by introducing new items to the visitor from visitors similar to them.

In summary, ProxiTour takes a hybrid approach combining collaborative and content-based filtering. It compiles a list of proposed exhibits based on the similarity of user profiles. It then ranks the items in the list using our visitor's profile as a guide. The list is presented via the application, using direct and indirect feedback to evaluate the recommendations and fine-tune user models. The recommendation system has not yet been implemented, since we are still in development and do not have access to actual user data, but it is part of the upcoming pilot and we have laid the groundwork for it in this work.

Recommendations could be made for exhibits the visitors would find interesting, for filtering the content that will be provided to the user, for compiling questionnaires for them based on their interests and experience, for prefetching information and many more purposes that aim to providing the visitor with the best possible experience.

## 3.6 Experimental analysis and evaluation

In this work we use experimental data from the testbed depicted in Figure 3.4 to validate the interoperability of the various components of the system, and to demonstrate that the created profiles can be used to differentiate the behavior of different users who move in the monitored space. The setup of Figure 3.4 features 11 (hypothetical) exhibits and 11 beacons laid out in distances that would be typical of a museum environment. Users (project members) moving around the museum space test the beacons' functionalities and perform controlled experiments by emulating users with specific behavior patterns. The events of the resulting streams contain key-value pairs of collected information (e.g. longitude, latitude, timestamp, etc).

Our preliminary tests demonstrate that the setup is functional and that ProxiTour can successfully track users moving about the experimental site. We tested this, by having project members perform a specific route and tracked them through the timeseries produced. All the exhibits that the test subject was supposed to approach were marked in the stream and some additional exhibits as well due to the limited accuracy of the beacons and the close proximity of the exhibits in the space (we discuss this further 3.6.1). Thanks to the localization and exhibit information included in the messages, we can calculate for example, how long each user spends on each exhibit (or group of exhibits). The information we produce is used to enrich the profile of the user that evolves, as more information is collected (Section 3.4).

The experiment involved 40 users (project members run multiple experiments simulating different users), moving about the test space and interacting with the exhibits in a predefined manner. Each user moved emulating one of the species-inspired behaviors we described in Section 3.5.

- *Ant users* would move closer to the walls and the exhibits, avoiding empty spaces and remaining close to the exhibits for long stretches of time.
- *Fish users* would move mostly in the empty space, making a few short stops at some exhibits and seeing almost all the exhibits.
- *Grasshopper users* went to specific exhibit categories each and spent a some time "studying" them, but the rest of the time they moved in the empty space.
- *Butterfly users* did not follow a particular path, changing direction often and taking a more random route, stopping often at exhibits.

In order to identify if the users were in the empty space or closer to the walls we marked coordinates that split the space into zones. We calculated the distance from the walls and labeled the data with a zone (1: peripheral, 0: empty space). This way we compiled a test dataset with 40 labeled user profiles to study, where we had equal representation for each class (10 users for each category). If distinction

of the patterns is possible, then recommendations should be customized to fit the pattern of behavior detected within each group. Our evaluation in this work focuses on the former aspect (detecting pattern of behavior and profile building); testing of the recommendation system itself is planned for an upcoming pilot with real users.

For each visitor we get a timeseries of events that were produced by the application (Section 3.3). The events had two types: *user_location_events* and *exhibit_events*. User_location_events are produced at a frequency $f$ that we can set, while exhibit_events are produced by proximity to some exhibit. For the purposes of this experiment we set f to 5 seconds so we could gather many position events and increase the precision of the tracking. Ideally, we would have preferred a larger sample, but we will have a chance to perform a larger experiment in an upcoming pilot (some data was also thrown out due to errors in the collection process).

For the model selection and hyperparameter tuning we used nested cross-validation, since our dataset was small. Nested cross validation (NCV) is an extension of classic cross validation (CV) that integrates an inner CV loop inside another CV loop. In k-fold CV we split the training set into K equal parts and train on K-1 and test on the one left. We repeat this process for each combination of K-1 folds and report the mean error. We can perform 10-fold CV on multiple models and pick the one with the best performance, then whichever model gave the best performance train it on the entire training set and test again on the validation set.

We performed NCV with K=10 folds by spliting the data into 10 CV folds in a stratified manner. For each fold k (k=1..10) we evaluated a model on a selected hyperparameter. We set validation set = fold k and training set is the remaining data. We now perform 10-fold cross validation of the training set with different values for each hyperparameter value and average the score over the 10 folds. Then we pick the model with the best performance on the training set, evaluate its performance on the validation set and store its score for the result of fold k. We calculate the mean score over all K folds and this is our error.

Table 3.1: Models examined.

| **Model** | **Hyperparameters** |
|---|---|
| `KMeans` | [number of clusters (K)] |
| `BisectingKMeans` | [] |
| `Gaussian Mixture Model` | [] |
| `Random Forest` | [number of trees, maximum depth of the trees, impurity] |
| `Logistic Regression` | [elasticNetParam, regularization parameter, max iterations] |

The models we tested and the hyperparameter values can be seen in Table 3.1. The best performing models are marked with bold and their performance can be

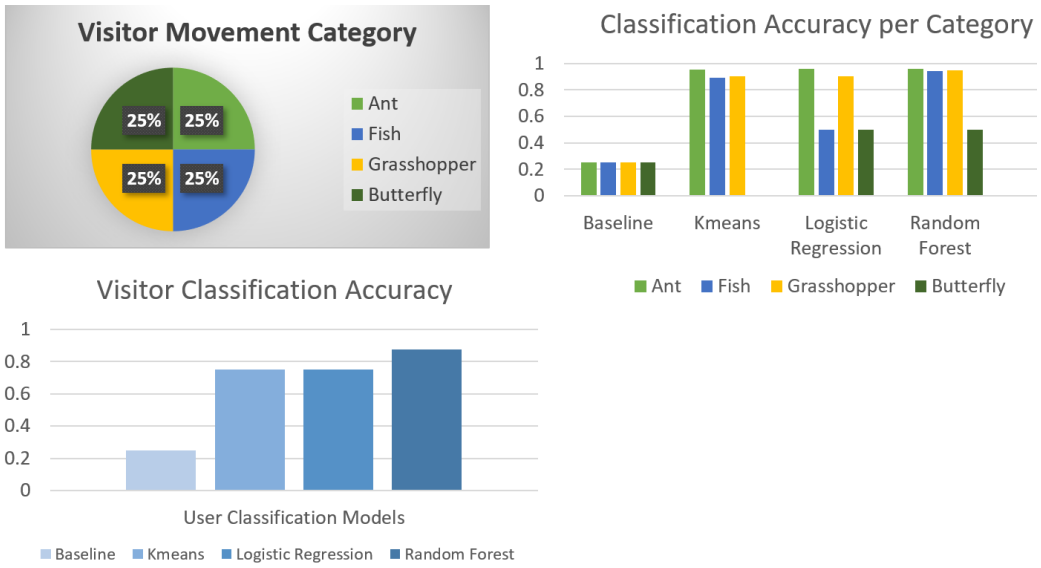



Figure 3.5: A representation of the class split in the data (upper left), results of the best performing models from Table 3.1 (bottom left), and a sensitivity analysis for each class from the best performing models (right).

seen in Figure 3.5. Our baseline for the performance of the models was a random predictor which would give as a 25% chance of picking the write class at random for an unknown user (not accurate in a real world scenario, but for the purposes of this experiment we assume equal representation for each class). We can see that all of our best models performed much better than the baseline, with Random Forest performing the best out of the three with an accuracy of 87.5%.

However, we also want to know which visitors troubled the most our models. In Figure 3.5-(on the right) we have a breakdown of performance of each model in correctly identifying each category. Baseline is trivially (25%). All three models have trouble identifying butterflies, since they can resemble either grasshoppers or more commonly fish users. Their random movement in space can sometimes resemble other patterns, which confuses the models. KMeans does not recognise them at all, matching them to other clusters as extreme cases. Logistic Regression mixes them with fishes, which makes sense if they moved in the empty space mostly. Random Forest performs the best, by only missing 50% of butterflies and classifying them as fish or grasshoppers, based on how close to the exhibits they went.

From the achieved accuracies and the sensitivity analysis we can safely conclude that the four types of behavior are distinguishable and the implemented profiling system can differentiate them. Next steps will include larger experiments in the context of a pilot at the Archaeological Museum of Ioannina. At that point, we can re-evaluate the classifiers when we run the pilot, but so far Random Forest seems to be performing the best and this would be the one we chose to test further

in a real scenario, based on the current experiments.

### 3.6.1 Challenges

The main challenge we faced was the noise in the dataset from the beacons due to their close proximity. When a user was moving close to a wall, on the other side of which there was a beacon, a message would be generated that would mistakenly position him in a different room, muddling the data.

To handle this issue, we marked the beacons that were producing confusing events and we distance_from_beacon parameter that along with the accuracy metric included in the messages that identified the position of the visitor, helped us reduce the noise a lot. We also kept track of the room the visitor was in. So, if a visitor was in room A and and we started receiving messages from beacons in room B we would wait to receive a few more messages from that room before changing the room status. If the room changes the timestamp of the change is marked. If we start getting messages from more than one rooms we keep the ones from the room we have in the status, if this goes on for more than a few seconds, then we generate a warning to the system that will be examined by the administrators, since it can be caused by beacon malfunction or the beacon may just need to be adjusted.

Another issue we faced, was that due to the setup at the testing space, some rooms only had one beacon, which made tracking movement from exhibit to exhibit irrelevant. In this case we disregarded those exhibits from the movement patterns of the visitors, by disregarding them in our calculations of the visitor's movement patterns. We also tried another approach of making one beacon appear to be connected to multiple exhibits in different locations in the room, but due to furniture in the room and too much noise from nearby beacons this approach was abandoned. At the moment this problem is studied by other members of the team and we expect to be solved by the time the pilot is up and running.

Due to the close proximity of some exhibits we received multiple messages with different exhibit id, without the visitor moving much. We cannot do much in this case, since we cannot detect the direction the visitor is facing. We assume that exhibits that are close together are probably in the same category and virtually group them together.The category will be mark the visitor's interest and we depend on visitor interaction with the application content provided to detect specific preferences to exhibits.

Deciding how to split the space in zones was also difficult, since quite a few rooms had only 1 beacon with very little leaway on how to get to it due to furniture. In the more spacious rooms we designed routes and marked coordinates that split the space into zones. We calculated the distance from the walls and labeled the data with a zone (1: peripheral, 0: empty space). We ignored room 9 in Figure 3.4 for example, because there is a very narrow path to get to it.

These are issues that seem to be managed by our solutions in the test space, but we may need to adjust our approach when we get data from the pilot.

## 3.7 Conclusions

This work describes a streaming application for building user profiles and exhibit profiles by processing localization and activity information/events collected by the ProxiTour IoT-enabled platform. Our experimental evaluation in a controlled testbed demonstrated that classification of users (key to effective personalized recommendations) is possible through an online stream-processing platform using ML libraries.

This work is on-going and we are trying new methods as the project develops. Future work on a larger scale pilot will study the effectiveness of recommendations and further investigate the potential of other ML techniques such as reinforcement learning to more aggressively explore the space of possible recommendations.

The results of this work were presented in Proceedings of 14th Symposium and Summer School on Service-Oriented Computing (SummerSOC'20) and published in Springer Communications in Computer and Information Science (CCIS) [23].

# Chapter 4

# SLO violation prediction for mass-transit systems

## 4.1 Introduction

Millions of people depend on mass-transit systems daily to get them where they need to go. However, when people depend on public transportation to get to work on time or to make an appointment, the unpredictable delays that often occur are a big deterrent for many [39, 46]. There are many reasons that can cause publication transportation, especially fixed-route bus transportation, to be late: traffic, construction, weather, special events, etc. It is this occasional unreliability that stops many people from relying on public transportation more often.

Just as in any other case where quality of service is important, cities may want to set up service-level agreements with bus operators, in order to try to keep them accountable for any deviations from the published schedules to the extent possible. Trip delays that are avoidable, due to organizational conflicts, lack of flexibility, or low availability of buses, need to be penalized. If the operator can predict when a bus is going to violate their service-level objective (SLO) they can take action to prevent it or to minimize the fallout. Like adding extra buses to cover a crowded shift, when buses have to skip stops due to overfull capacity.

The goal of this work is to develop a system for real-time prediction of SLO violations by continuously analyzing datasets and feeds containing static and real-time data collected and reported by mass-transit systems. As telematics services and smart applications for public mass-transit systems are increasingly being developed in recent years, a trend to create and offer open datasets with transportation schedules, associated geographic information, and real-time information about actual trips (such as stop times) have materialized into a commonly accepted data format, the General Transit Feed Specification (GTFS) [8, 9] and several open-data feeds [5, 10, 12]. Another goal of this work is to contribute techniques for discovering common behavior on such datasets and deviations from it. These techniques will be used to reason about compliance to expected target metrics, as well

as to respond with corrective measures to anticipated deviations from goals. We develop our approach using open GTFS datasets and feeds, thus our developed techniques are applicable more broadly to the smart transportation industry.

In this chapter we address the problem of measuring expected duration of entire routes, as well as between stops, and correlating with certain days, times of day, and specific routes and stops. We build profiles for each service, route, stop and vehicle that are updated dynamically with new information as we receive events from the trip. Finally, we study the problem of determining to what degree service-level objectives are likely to be violated taking into account the expected outlook of a trip and the history so far. We leverage a combination of batch and stream-processing technologies to perform the training of our models, as well as to implement real-time response to a range of challenges in intelligent bus transportation.

This chapter is structured as follows: We start by discussing the GTFS and GTFS-realtime specifications and open-data feeds available for analysis. We then describe what is a service-level agreement (SLA), a service-level objective (SLO), and how they apply to the mass-transit sector in Section 4.3. Then we describe how we transform our dataset to building statistical models and calculating SLIs, in order to predict if an SLO violation is going to occur. In Sections 4.4 and 4.5 we present a prediction model training/testing pipeline and the methods that can be used. Then, in Section 4.6 we present a live dashboard for visualizing generated warnings. Finally, in Section 4.7 we describe our experience with an open GTFS dataset collected from a provider in the city of Rome. The rest of the chapter presents related work, our conclusions and future work.

## 4.2 Open datasets and feeds

As telematics services and smart applications for public mass-transit systems are increasingly being developed in recent years, a trend to create and offer open datasets with transportation schedules, associated geographic information, and real-time information about actual trips (such as stop times) have materialized into a commonly accepted data format, the General Transit Feed Specification (GTFS) [8] and GTFS-realtime [9], and a number of open-data feeds are currently publicly available [5, 10, 12]. GTFS was created in 2005 by Google and partners and has since become a generally accepted data format for route data produced by mass-transit services. A large number of transit agencies worldwide produce GTFS data and share them openly with the general public. As an example, the OpenMobilityData site [12] currently serves open-data feeds from 1327 mass-transit providers in 677 locations worldwide.

The **static GTFS** data model is depicted in Figure 4.1 showing entities (files) that are published typically within zip files. These files describe the scheduled service, with information about the routes, trips, stops, stop_times and more and they are updated by the agency responsible for the feed. The Stop_times file for

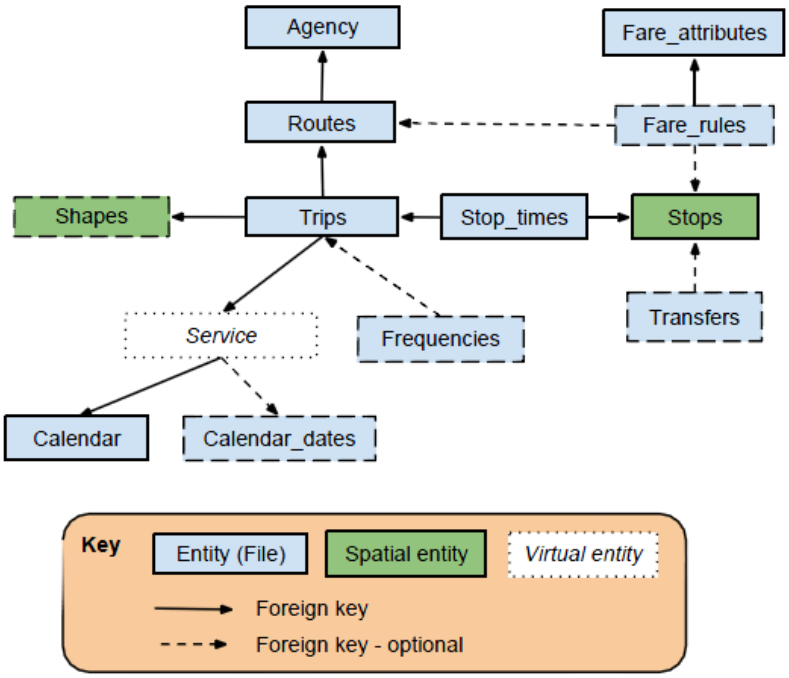



Figure 4.1: The GTFS data format (from https://www.transitwiki.org/).

example, contains times at which a vehicle arrives at and departs from stops for each trip. Each line in Stop_times is a comma-separated line with the following information:

- trip ID (identifies a trip)
- arrival/departure time (at a specific stop for a specific trip on a route)
- stop ID (identifies the serviced stop)
- order of stop in sequence
- plus other parameters

Stops and trips are further described in files Stops (column stop_id) and Trips (column route_id). We reviewed several datasets from Open Mobility Data [12] and other repositories to determine an appropriate one to use in this study.

A **realtime GTFS** feed provides realtime information about disruptions in the service (e.g. service interruptions, important delays, closed stations, etc), the position of the vehicles and updates about trips in-progress. These feeds are produced by automated vehicle management (AVM) monitoring systems and are published as binary files with the .pb extension (protocol buffers). These files are updated at frequent intervals (e.g. approximately 30 seconds for Romamobilita [13]).

Our primary motivation to use an open dataset in the GTFS format is to ensure the reproducibility of our work and its applicability and reusability on a larger range of data sources. Our survey showed that several such datasets are amenable (with pre-processing, to adapt to different preferences across feed providers, given the flexibility in the quality of data allowed by GTFS) to our planned analytics. In section 4.7 we examine one such open GTFS dataset, made available by Romamobilita [14] on their website, which contains regular updates from their AVM system, formatted in the way described in the GTFS protocol.

## 4.3 Service-level objectives

A **service-level agreement (SLA)** between a mass-transit system operator and city authorities is a form of contract that specifies the service to be provided, schedule times, locations, costs, performance and responsibilities of both parties. It could be set up for example, specifying the acceptable travel times (route start-to-end, as well as at what time to call each stop of a route). A **service-level objective (SLO)** describes the specific measurable characteristics of the SLA, such as availability, frequency, delay, etc. It could also specify what percentage of trips (e.g. 90%) should meet the goals set for the bus driver in the SLA, such as:

- A trip of line A should not be more than 5 minutes late at the destination, 90% of the time.
- No more than 1 minute delay to begin the trip 85% of the time for any vehicle.
- No more than 3 minutes delay to arrive at any stop in the route 95% of the time for any vehicle.

The SLO may be composed of one or more **service-level indicators (SLIs)**. SLIs are quality-of-service measurements that when combined can produce the SLO achievement value. Examples of SLIs in our case can be the deviation from the schedule (e.g. trip starts at 10:00, but bus leaves at 10:35). Figuring out whether a bus operator violated their SLO is a complex process that depends on defining the right SLIs.

For every service in the dataset we have the programmed days they run on and their start and finish dates. We can find if the bus operator is in violation of his SLO using the historical data. We can also try to predict if a trip in progress will lead to an SLO violation, by predicting how long the trip will delay and *observing how the delay affects the overall statistical model for the operator service/route.* For simplicity we assume that our SLO is solely on the total duration of the trip, but we calculate many SLIs in the profiles we construct for each service, route, vehicle and stop. These SLIs are updated dynamically in real time and can be used to construct more complex SLOs. Next, we describe how we build these profiles and how we use them to produce warnings and predict SLO violations.

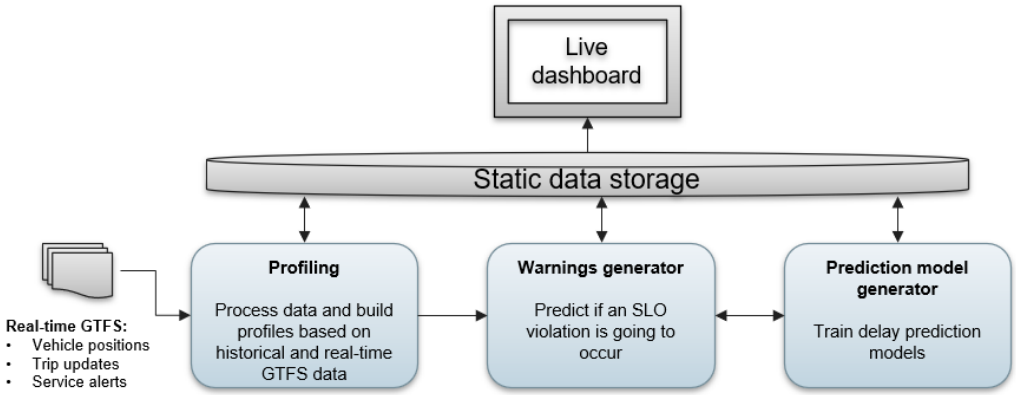



Figure 4.2: Architecture of SLO violation prediction system

## 4.4 Dataset transformation and SLI construction

A static GTFS-dataset, in its raw form consists of a number of tables that are stored in static format. This data is updated by the responsible agency, so that it contains the most up-to-date information (e.g. Romamobilita updates the static GTFS every day). From the static data we can extract information about the trips we expect to be performed as part of a service, the routes these trips are to follow and the times they are scheduled for.

In addition to the static data, we have three realtime GTFS feeds: *vehicle positions, trip updates and service alerts*. The **trip updates** feed contains realtime updates about the progress of a vehicle along a trip. A trip update can concern a scheduled trip (its trip_id will be in the static trips table), it can be about a trip that is on a specific route but has no schedule (the route will be in the static data, but there will be no corresponding id in the schedule) or it can inform us that a trip has been added to or removed from the schedule. These updates can refer to events that have occurred or predicted events (e.g. predicted arrival time at a stop).

The **vehicle position** messages show information about the position of a vehicle, along with other optional information, like the current stop sequence, the status of the vehicle with respect to the stop (INCOMING_AT, STOPPED_AT, IN_TRANSIT_TO), speed, occupancy percentage and more. This feed is mainly used by applications to visualize vehicle positions on a map. The **service alert** messages indicate some incident in the public transit network. Each message is associated with some entities whose users must be notified, a text for the alert and the headline and a number of optional fields, like the time we want the alert to appear and more.

Based on these feeds (seen as streams) we will build profiles for each service, route, stop and vehicle, so that the operators can make informed decisions and get insights into the operation of the transportation system and any problems that

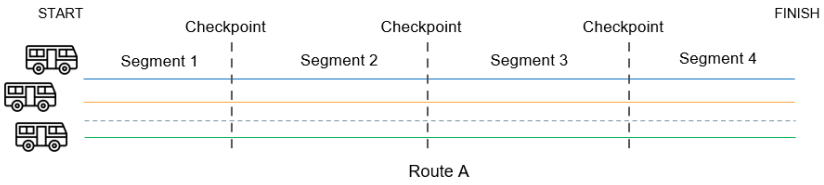



Figure 4.3: Route sessions, segments, and checkpoints.

might exist. This way we can produce SLIs for SLOs that are wide in scope (e.g. 90% of the trips in a route must be less than X minutes late) or very specific (e.g. 90% of the arrivals at each stop must be on time) and others in-between.

A trip in our data has a unique identification code (trip_id). Different trips on the same route have different codes. For each new trip seen in a day, our system starts a new **session** and calculates various characteristics of it. We keep track of the duration of the trip, the duration of each **segment** (Figure 4.3, described in more detail later) in a trip, and the time it spent at each stop. As we load more data from the stream this information gets updated in the state of the session.

Then for each route we calculate statistics about the time it took to be executed (TOTAL_DELAY). Specifically, we calculate the mean, median, 25-,75-,90- and 99-percentiles and the standard deviation. This way we can check the distribution of each trip and see what constitutes normal behavior or a deviation that could lead to an SLO violation. If a trip's execution of a route falls into the 90th percentile, it is considered to be late and we check if there is a violation of the SLO, by checking how the service or route distribution changes (if our SLO is dependent on one of these two). Without loss of generality, we use a hard threshold of 5 minutes, which we consider to be the SLO for total delay that the operator needs to keep for 90% of their trips. We track the same statistics per-stop (STOP_DELAY), using the delay of the bus in entering a stop, per-vehicle, summarizing the information of the trips the vehicle operator performed and per service, combining the information of all routes/trips in that service. This way we have constructed profiles with many SLIs for many levels of abstraction that can be used to compose SLOs. At the end of each session these profiles are stored with a timestamp.

The next thing we examine is a relationship between the delay at certain stops and the total delay of a trip. We examine the correlation between the stop delay and the total delay for different times of the day and we find that there are certain stops that are statistically important to the bus being on time. Many times these stops are common for multiple routes, which makes sense as bus routes may cross or overlap at certain locations (bottlenecks). This correlation is of significant importance for predicting if a trip will be late.

In order to predict if there is going to be an SLO violation we need to know ahead of time how late a trip is going to be, therefore we need to predict the final delay. We pick a number of stops per route to use as our **checkpoints**, splitting the route into **segments** (Figure 4.3). If a route has stops that were

highly correlated with the total delay, then they will be used as checkpoints. If a trip has no such stops, we check if another route with overlapping stops has marked some as a checkpoint, otherwise we split at the most common stops or take equidistant checkpoints (as in Figure 4.3). When a bus arrives at a checkpoint, a prediction is made about how late it is going to be, based on the delay it has already accumulated up to this point, the day of the week, the percentage of the route remaining, the time of day and a weighted average of the delay of previous trips on the route for the past 2 hours (older trips are less relevant). Our selected feature set is summarized in Table 4.1. We can check in which percentile of the appropriate distribution this delay falls and we make a decision about whether the operator is going to be in violation.

| |
|---|
| - Current delay |
| - Day of week |
| - Time of day |
| - %Route remaining |
| - Weighted average of delay for the last 2 hours |

Table 4.1: Feature set.

## 4.5 SLO violation prediction and evaluation

Apache Spark has limited support for streaming-based machine learning analysis. We can use the implemented machine learning algorithms trained in batch mode for streaming prediction with some adjustments, but training on streaming datasets is not yet supported for Spark Structured Streaming[1]. There is however support for models that are trained on static dataframes, but with an extra step in the process can be applied for prediction on streaming datasets.

In order to train our batch model we implement a pipeline using the Spark MLlib library. We use 80% of the data for training and testing and we keep 20% for validation. We need to pay attention to preserving the distributions in the training and test set (e.g. 1 out of every 5 days (depends on the size of the dataset) is used for testing). We train regression models (linear regression, SVMs, RandomForest, etc) with a number of hyperparameters (number of iterations, regression parameter, elastic net parameter) and pick the model that performs the best and train it on all the data. We can check the statistical significance of our features, for the prediction, by examining their p-values. We can choose to train a single general model or a model for each route. Delay prediction usually works better when we use the second approach, so this is the approach we opt for.

To evaluate the performance of our models we can use 3 measures: the mean absolute error (MAE), the mean absolute percentage error (MAPE) and the root

[1]Spark Streaming has support for the streaming linear regression and kmeans algorithms, but uses the RDD API (outdated since the introduction of Spark Structured Streaming)

mean square error (RMSE). The equation for each measure is as follows:

$$MAE = \frac{\sum |t_{observed} - t_{predicted}|}{N}$$

$$MAPE = \frac{1}{N} \frac{\sum |t_{observed} - t_{predicted}|}{t_{observed}}$$

$$MAE = \sqrt{\frac{\sum (t_{observed} - t_{predicted})^2}{N - 1}}$$

where $t_{observed}$ is the observed bus delay and $t_{predicted}$ is the predicted delay for the trip and N is the number of observed trips.

Based on preliminary experiments, we determined that the offset of the checkpoint affects the performance of the model on average. When we predict delays too early (first quarter of the trip) there is a lot of time for the operator to make up for their delay, which means we are more likely to make a wrong prediction. Checkpoints at the halfway point are more reliable, while the checkpoints of the last quarter are almost always correct. Accuracy of prediction here refers to the occurrence of an SLO violation (however, it also holds for predicting the delay of this specific trip). This means that we can rate our prediction with a different level of confidence based on the point of the route the prediction is made.

For training data on the streaming path we need a learning model like regression with Stochastic Gradient Descent, which can train on batches of data. However, there is currently no implementation of them for the Spark dataframe/dataset API we use. Another practical alternative is to train approximate models on batches of data and perform a weighted average on the coefficients of the produced models, if the distribution of the observations and the dependent variable was consistent across all batches. The approximate incremental model would not be as reliable, but if batch training takes too long it could be a faster way to update the old batch model with newer observations, while a new batch training is running. Finally, another approach to include realtime data to our prediction process, is to use a model to predict a delay and then use Kalman filters on that prediction.

The reason we can be more lax with delay prediction accuracy, is that we only want to find if we are going to pass a threshold (e.g. 5 minutes), to be considered extremely late and affect the distribution that determines if our SLO has been violated. We calculate how the predicted delay will affect the distribution and if the 90th percentile exceeds 5 minutes (which is a test threshold we have set), then we predict that the operator will violate his SLO. This means that the only critical errors are the ones made on trips that are often late and can be found in violation if we predict a larger delay than the actual one or if we miss it by underestimating it. In order to test the validity of our solution as a whole we need to examine the true positive rate (TPR) and False positive rate (FPR) of our predictions.

$$TPR = \frac{TP}{TP + FN}$$


| Timestamp | Warning text | Trip | Vehicle number | Shift | Stop Code |
|---|---|---|---|---|---|
| 9/16/2019 8:52:56 AM | Danger of SLO violation. (>10 minutes late) | 12 | 1141 | 41 | PL121 |
| 9/16/2019 10:58:56 AM | Start delay (>5 minutes late) | 31 | 4326 | 13 | A0896 |
| 9/16/2019 11:13:21 AM | Danger of SLO violation. (>7 minutes late) | 31 | 4326 | 13 | PL451 |
| 9/16/2019 12:26:21 PM | Start delay (>5 minutes late) | 36 | 1141 | 21 | C1056 |
| 9/16/2019 2:46:21 PM | Start delay (>5 minutes late) | 98 | 2536 | 54 | PL105 |
| 9/16/2019 8:45:09 PM | Start delay (>5 minutes late) | 15 | 3105 | 42 | PL105 |

Figure 4.4: Dashboard warning system with PowerBI, with color-coded warnings

$$FPR = \frac{FP}{FP + TN}$$

where TP are the true positive predictions (the trips we correctly predicted as leading to SLO violation), FN are the false negatives (the trips that lead to SLO violations but we missed them), FP are the false positives (the trips we marked as leading to an SLO violation that did not) and TN are the true negatives (the trips we correctly identified as safe of a violation).

Our experience with an active open-data feed that forms an initial basis for evaluating this work is described in Section 4.7.

## 4.6 SLO violation warning system

When we predict that an SLO violation is about to occur we push a message into a sink (Figure 4.2) that is connected to a live dashboard created using PowerBI. On the dashboard we display all the violations that occur. A sample view of the dashboard is depicted in Figure 4.4. The warnings table view is color coded for different types of warnings. SLO violation warnings are marked red and they are reported along with the trip they are on, the vehicle number, the timestamp, the current delay and the expected (predicted) delay.

We also send warnings to the dashboard for the following events:

- Delay in starting a trip.
- Missing a stop.
- Staying at a stop for too long (We can use an arbitrary threshold or base it on the calculated statistics in the stop profile).

- Trip delay prediction (We can use an arbitrary threshold or base it on the calculated statistics in the trip profile)
- SLO violation prediction.

All the above mentioned messages are shown as warnings, which we can color code by importance or urgency as is shown in the figure. This is customizable as are the filters we use. We can also visualize the profiles of each trip, vehicle and stop on the dashboard, so that the operators can visualize the summarized information easily and get the desired insights. This is particularly useful if the operator sees a warning about a vehicle and wants to evaluate the situation fast, they can look at the profile of the trip and the vehicle performing it and make more informed decisions.

## 4.7 Empirical evidence with GTFS feed data

The system we described assumes GTFS-formatted data that adhere to GTFS protocol rules. However in a real-world open datasets are rarely perfect. We tested a real GTFS dataset to evaluate its quality and interpret the events included. After a thorough evaluation of open GTFS feeds, we identified the public Open Data initiative of Romamobilita.it, a company responsible for handling mass-transit in the city of Rome, as an appropriate one. Romamobilita uploads each day a static GTFS file with the information of the schedule. Additionally, three feeds in the form of protocol buffer files, for trip updates, vehicle positions and service alerts, are updated approximately every 30 seconds with new information collected by the AVM system. We built a dataset by automatically downloading published Romamobilita files over several days.

The format of the static GTFS dataset was found to follow the specifications of the protocol. All the required files were included as well as a Shapes file and a Calendar dates file. We analyzed the static data and found that the files changed from day to day, with some trips being included in every one and others being added or deleted daily. Every static GTFS set of files was approximately 300MB. The realtime GTFS data however, showed a range of sizes depending on the time of day, with the trip updates feed having the largest files at 1,3MB average and over 2MB for the largest ones. This led to a very large amount of data being collected over the course of 10 days (15GB in protocol buffers serialization, close to 40GB unpacked). If we project this growth over longer periods of time we can see that storing and parsing this data in their entirety can be costly. The rate of incoming data over the course of a day can be seen in Figure 4.5. We can see a significant drop in the size of incoming messages during the night, which makes sense since we have much fewer active vehicles.

The Romamobilita.it website indicates that the realtime feed is updated approximately every 30 seconds. However, the actual update rate seems to vary somewhat. Oftentimes we get duplicate files after 30 seconds. We also get periods

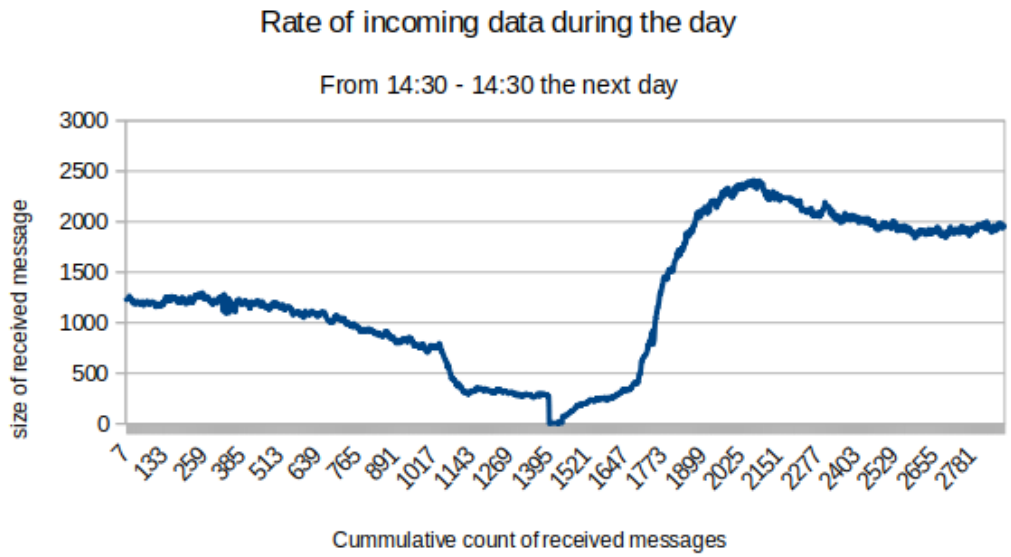



Figure 4.5: Incoming data rate over the course of 24 hours.

where the files are not updated for up to 10 minutes; these large delays are not many, however they are evidence of variability in the system that provides the feeds. After cleaning the data from duplicate files and duplicate messages within the files, from trips or vehicles that showed no change, we get a significant decrease to 69% of the original size of the data. The percentage of the messages that were duplicate was approximately 21%, however since some of the duplicate files were large, this proportionally affects the final count. This behavior is heavier on certain days, thus it is an issue that is not always present.

After unpacking the realtime data, we found that they did not have proper field names as described in GTFS documentation, unlike the static files. Realtime data had encoded field names that we had to decode and map to the fields documented in the protocol for each feed. After this, we wanted to see how well the schedule mapped to the recorded feed. The results were somewhat surprising: 3% of the trips in the feed were not in the schedule (about 2000 trips), which is not a small number. We used the route_id to help us find an expected duration for the trip, when possible. However, we also found discrepancies in how vital information like the route_id, the service_id and the trip_id were represented. Information that should be repeated precisely to map the feeds to the schedule were not, for example the route_id was written differently or the combination of the route and the direction differed, but the sequence of stops was the same. Handling such cases, requires going through them one-by-one and trying to detect patterns to fix the issue. Since we do not have access to the provider to ask for clarifications, this is a difficult task with 22,350 trips per day on average. This makes it clear that cleaning of the data is necessary, a task that is out of the scope of this work but

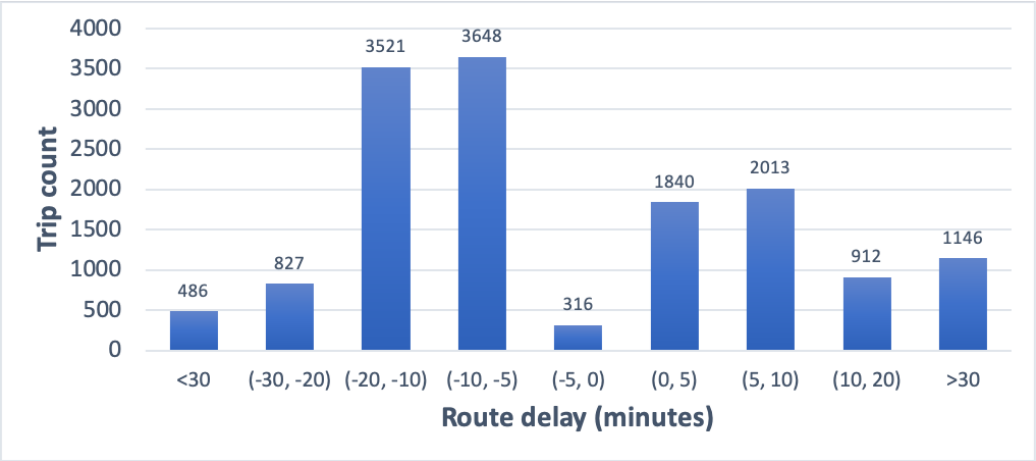



Figure 4.6: Average number of trips per different range of delays (minutes).

an interesting direction of future work.

When focusing on the scheduled trips, we again found unexpected behavior. We mentioned earlier that some trips in the feed could not be mapped to the schedule in some way; we additionally found scheduled trips that were not in the feed. These were far more (46%), possibly due to the fact that not all vehicles are recording their routes. In an effort to compare the normal behavior in the schedule with the behavior in the trip updates, we calculated a statistical profile for each route, containing 0, 0.25, 0.5, 0.75, 0.9, 0.99, 1 percentiles, the average, the standard deviation of the duration of each trip on that route and the count of the number of trips on that route. We compared these values with the standard behavior in the schedule and found that there were more trips that were early than late. Figure 4.6 depicts the distribution of trips across delay ranges, based on how late or early they were, averaged over 10 days. Negative ranges indicate early arrival and all values above and below 30 minutes are grouped together. We would consider the latter to be the result of errors in clock synchronization or in the schedule or realtime feed, but cannot tell for sure without operator feedback.

It is obvious from the results of Figure 4.6 that either schedule inconsistencies are widespread in the public transportation system in Rome and/or there are occasional issues with the data. Since we are reverse-engineering the process of analyzing normal and scheduled behavior, reasoning about these issues is in no way trivial and it would require making assumptions based on what we consider a logical explanation. Therefore, the ideal would be for the provider to be involved in a process like this, in order to provide insights about the inner workings of their system and clarify these types of behaviors. If the behavior we observe is a correct representation of the provided service, then they would appear to violate most reasonable SLOs, which is perhaps an indicator that no SLA is actually in place.

## 4.8 Related work

SLO violation prediction is a research field with many areas of application (systems, transit, etc). Such systems provide accountability for the guarantees that various services claim [19, 47]. In the area of transportation this type of violation detection is very important, since many people depend on it. Predicting if a public transportation vehicle, like a bus, is going to violate their SLO, is very closely connected with predicting how late it is going to be. There have been many approaches to predicting delay in mass-transit.

The more basic approach uses basic statistical models, based on the average delay observed in the historical data [42]. The problem with this type of model, is that it is limited by the consistency of route delay patterns and it does not take into account real-time information. Others, have used both historical and real-time bus data [48, 58] to estimate the expected bus arrival times at a stop, by using real-time speed information from GPS devices to project where the bus will be at a future time. Since the variables used for delay prediction are correlated, regression models are also used for delay prediction [29, 53]. They used real data to predict travel times on route segments, testing various models. The final approach to delay prediction is based on Kalman filters, which have been widely used for their ability to filter noise and continuously estimate states from real-time data [30, 56, 43, 59].

Our work leverages ideas from these previous works on how to construct features sets and to implement prediction models. We leverage the ideas of segment delay prediction and training regression models on historical data to implement an incremental online training model to predict delay. We utilize this delay to predict if the SLO will be violated, based on statistical information from historical data.

## 4.9 Conclusions and future work

In this chapter we discussed SLOs and their adherence in the mass-transit sector. We discussed what an SLO violation is and how it could be detected in GTFS data. We described a system that reads a stream of data and builds statistical summary profiles for each service, route, stop and vehicle. It passes a set of characteristics updated in realtime from this profiles to an application that checks if an SLO violation is going to occur based on delay prediction models or if other type of anomalous behavior is exhibited and produces warnings for them. The system feeds the warnings to a live dashboard displaying them for analysis, along with other analytics about the data, like profile information summaries.

We tested a real open GTFS dataset collected from the Romamobilita.it website and we tested how the GTFS protocol is reflected in a real-world scenario. We tested our profile-building application on it and studied the produced summaries for insights. However, the data exhibited behavior that is hard to fully interpret without the help and contribution from the providers and we can only guess at the

causes at this point. Therefore, future work could include broader experimentation with GTFS data in concert with a provider, who can clarify things and help, when black-box interpretation of exhibited behavior is not possible.

# Chapter 5

# Scalability analysis and performance optimization

The streaming applications we described in the previous chapters were implemented using Spark Structured Streaming [22, 16] (Section 2.3) as our stream processing engine (SPE) and Kafka for data ingestion and as the pipeline between our micro-services. In this section we will see how a streaming application performs out of the box and then we will start tweaking parameters and trying different combinations to see how the system responds. In each step we explain our reasoning for picking some parameter and testing certain values. Eventually we pick a configuration, explain why we believe it will perform the best and test the streaming application again. The final results show whether we were successful in improving performance and by how much.

We start by describing the cluster we run our experiments on, see how well our solutions can scale and discuss a few optimization techniques for the improvement of the performance.

## 5.1 Setup

The cluster we run our experiments on has 15 nodes with a total of 644 cores and 2.41TB memory available[1]. Each machine has from 16 to 64 cores and ranges between 94GB and 754GB of memory. On this system a Kubernetes (k8s) Spark cluster manager[2] has been setup and our Spark applications are deployed to the master setup in one of the nodes (Figure 5.1).

We create a jar file with our application, create a docker image of Spark with the jar loaded and submit our application to the kubernetes cluster master node (Kubemaster) (k8s://api-server-host:k8s-apiserver-port) on the. Prefixing the master string with k8s:// will cause the Spark application to launch on the

[1]We are thankful to Antonis Chazapis for support and the FORTH-ICS CARV laboratory for access to this infrastructure during our experimentation on the contents of this chapter.

[2]https://kubernetes.io/docs/concepts/overview/components/

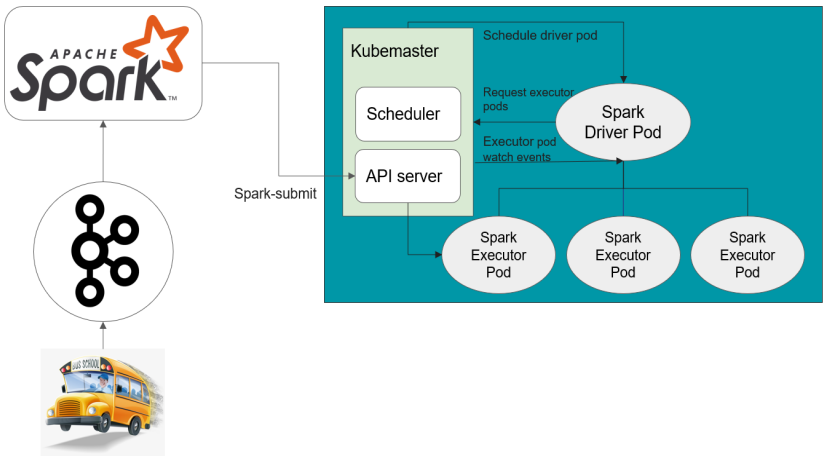



Figure 5.1: Spark scheduling on Kubernetes.

```
~]$ kubectl get pods --sort-by=.metadata.creationTimestamp -o wide | grep streaming-app
streaming-app-driver                          1/1     Running     0          29s     10.244.
streaming-app-1589918905729-exec-5            1/1     Running     0          25s     10.244.
streaming-app-1589918905729-exec-4            1/1     Running     0          25s     10.244.
streaming-app-1589918905729-exec-1            1/1     Running     0          25s     10.244.
streaming-app-1589918905729-exec-6            1/1     Running     0          16s     10.244.
streaming-app-1589918905729-exec-8            1/1     Running     0          8s      10.244.
```

Figure 5.2: Successful provisioning of Kubernetes pods for streaming application.

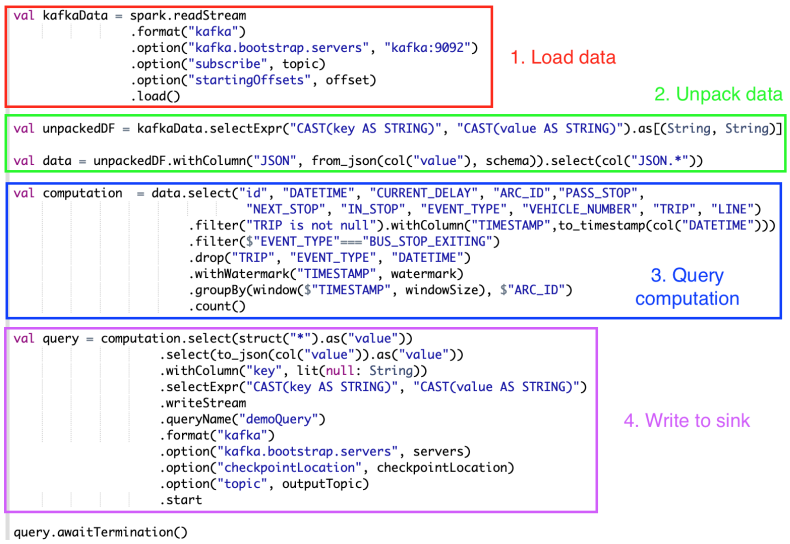
Kubernetes cluster, with the Kubernetes manager being contacted at api-server-url.

In Kubernetes mode, the Spark application name that is specified by spark.app .name or the `--name` argument to spark-submit is used by default to name the Kubernetes resources created, like the driver and the executors. On submit, we specify configuration properties regarding the resources (e.g. memory, CPU cores) to be used by the container pods as well as other configuration options (e.g. logging). As our image has already been pushed and is available in the Docker repository, we set the IP address of the repository and the image name. Finally, we specify the location of our application executable file and the necessary application parameters.

Kubemaster launches the driver pod using the image we have specified and executor pods that will run our code, provided that everything was in working order (Image was found and functional) and that all the resources we requested were available (Figure 5.2). Kafka is also running on this system as a separate service.

## 5.2 Out-of-the-box performance

Part of this work was to prove that our solution was scalable, so we tested an out-of-the-box deployment of Spark 2.4.5 and Kafka on the system we described




```
val kafkaData = spark.readStream
                .format("kafka")
                .option("kafka.bootstrap.servers", "kafka:9092")
                .option("subscribe", topic)
                .option("startingOffsets", offset)
                .load()

val unpackedDF = kafkaData.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)").as[(String, String)]

val data = unpackedDF.withColumn("JSON", from_json(col("value"), schema)).select(col("JSON.*"))

val computation  = data.select("id", "DATETIME", "CURRENT_DELAY", "ARC_ID","PASS_STOP",
                               "NEXT_STOP", "IN_STOP", "EVENT_TYPE", "VEHICLE_NUMBER", "TRIP", "LINE")
                       .filter("TRIP is not null").withColumn("TIMESTAMP",to_timestamp(col("DATETIME")))
                       .filter($"EVENT_TYPE"==="BUS_STOP_EXITING")
                       .drop("TRIP", "EVENT_TYPE", "DATETIME")
                       .withWatermark("TIMESTAMP", watermark)
                       .groupBy(window($"TIMESTAMP", windowSize), $"ARC_ID")
                       .count()

val query = computation.select(struct("*").as("value"))
                       .select(to_json(col("value")).as("value"))
                       .withColumn("key", lit(null: String))
                       .selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
                       .writeStream
                       .queryName("demoQuery")
                       .format("kafka")
                       .option("kafka.bootstrap.servers", servers)
                       .option("checkpointLocation", checkpointLocation)
                       .option("topic", outputTopic)
                       .start

query.awaitTermination()
```

Figure 5.3: Example of a streaming query.

in the previous session. Since streaming jobs normally do not stop, we added action listeners to our application that terminated the streaming query when it got 0 input tuples after a trigger and gave us information of the throughput and ingestion-rate and execution time for each trigger.

In order to be able to interpret our results, we used a simplified streaming query. We show the code for this query in Figure 5.3. In the first phase it consumes data from a Kafka source (*Load phase*), it extracts the message from the value field of the key-value pair that Kafka uses (*Unpack phase*) and then it performs some of the basic transformations we always perform, like cleaning null values and transforming columns (e.g. DATETIME to TIMESTAMP and DATE column creation) and then computes how many vehicles were in each segment (ARC_ID) per hour (*Query phase*). We reconstruct our results into key-value pairs and produce them back into another Kafka topic (*Sink phase*).

This query has two stages. Stage 1 reads the data from Kafka, unpacks it, performs the filter and select operations and when it reaches the watermark operation, which is an action, we switch to Stage 2 (More information on Spark execution can be found in Chapter 2). Stage 2 performs the aggregate operation, repackages the result state and sends it to the sink. Stage 1 has as many partitions as the source (50 in this case) and after reshuffling Stage 2 has the default 200 partitions.

Our data have been loaded into a Kafka topic with 50 partitions (we will discuss topic partitioning further in Section 5.4), meaning we can have up to 50 threads consuming data in parallel. In order to increase the load that our application has to handle, we copied our 15 million tuples into Kafka five times, creating a stream of data approximately 45GB long.

We run this application multiple times for various configurations, trying to

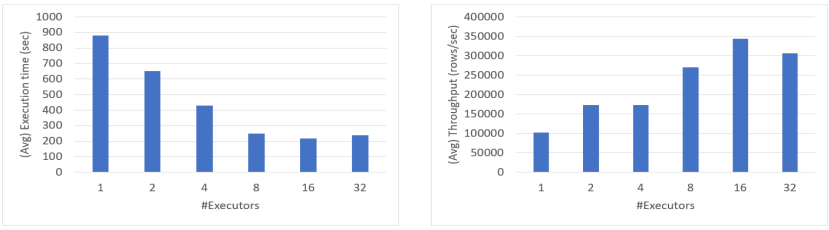



Figure 5.4: Horizontal scaling, execution time (left) and throughput (right).

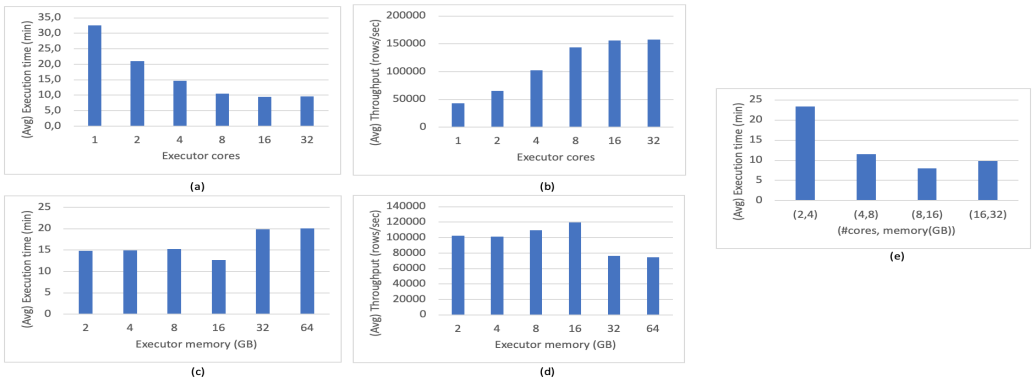



Figure 5.5: Vertical scaling scenarios, single executor.

scale it vertically and horizontally. The results of this experiment can be seen in Figure 5.4 and Figure 5.5. For each scenario we show the average execution time and average throughput results (how many rows/second were processed by our query).

Figure 5.4 shows how our streaming query scales horizontally by increasing the number of executors while keeping resources per executor stable at 4 cores and 8GB of memory each. We see that performance improves up until we reach 16 executors and then it plateaus for a while and gradually starts getting worse. In particular, while initially we expected that by doubling our resources we would cut execution time almost in half, this is not the case, as we note only small improvement. At 16 executors, we have exceeded the maximum number of executors that can consume data simultaneously from a 50 partitions topic (#cores $\times$ #executors = # parallel consumer threads) and after this point performance gradually grows worse. We assign this to the fact that some processes are idle during the data loading phase and only start contributing at Stage 2, only adding communication and management costs for the first part of the job.

Figure 5.5 shows how our streaming query scales vertically on a single executor, by increasing the number of cores (Figure 5.5-(a,b)), memory (Figure 5.5-(c,d)) or both (Figure 5.5-(e)). In Figure 5.5-(a,b) we keep the memory stable at 8GB per executor and double the number of cores. We see an obvious benefit by adding cores to the executor, which starts to wane after 8 cores. Increasing memory while

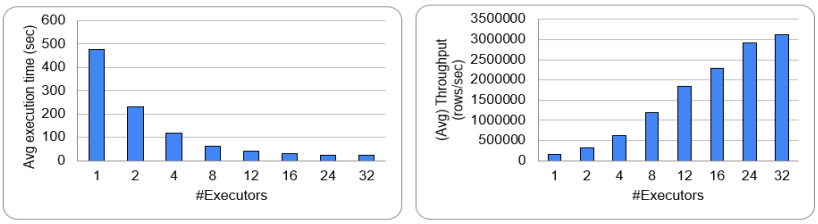



Figure 5.6: Horizontal scaling with Execution time results on the left and Throughput on the right, using files.

keeping fixed the executor cores at 4, does not seem to offer that much of a benefit. Which means that the performance we noted at the horizontal scaling was not due to lack of memory. We even see worse performance for 32 and 64GB of memory, which could be explained by the fact that when we ask for that much memory we always get a pod on the same node and that node might be slower. In Figure 5.5-(e), we see how scaling both values affects performance. Giving more cores to an executor will lead to higher memory usage, which makes increasing the memory available to that node, a logical solution to avoid unnecessary disk I/Os. We notice that the 8 cores and 16GB of memory seems to be the optimal configuration in our case, which is also reflected by the other experiments.

It is obvious that this performance is far from optimal and we need to make changes in our design to make it scale better. To our knowledge there is not a study yet in how the metrics together affect performance and this makes sense since each system and application behaves differently. There are various suggestions from the Spark community about what constitutes best practices, but they are on a case-by-case basis and most are based on developers' experience instead of numbers.

In the rest of this chapter, we are going to examine different parameters for improvement, both for Spark Structured Streaming and for Kafka. We have two systems that need to be configured to perform optimally together. Therefore we check each one's parameters separately and then use the best configurations to test the entire pipeline. The final results are tested again on the sample application and in Section 5.5 we also show a before-and-after performance comparison.

## 5.3 Scaling data ingestion

The first thing we need to check, is how much of our performance cost is due to the delay caused by Kafka. The easiest thing to check is whether having the data in files on shared storage, instead of a remote Kafka server, will improve performance. We replace the Kafka part of our application with a path to a shared folder and we rerun the experiment keeping everything else the same. This way the only thing that changes is how we consume the data. We expect to see improvements in performance and the difference removing the first two steps of the analysis can be seen in Figures 5.6, 5.7.

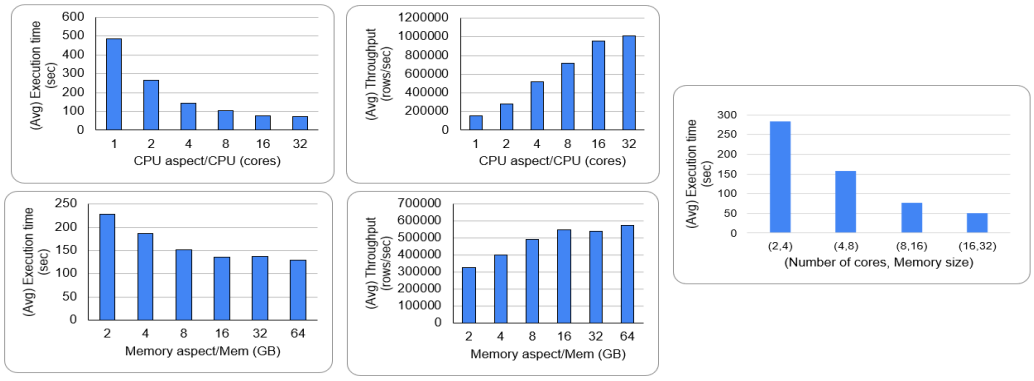



Figure 5.7: Vertical scaling scenarios with 1 executor, using files.

Figure 5.6 shows the performance improvement we get from scaling our system horizontally and keeping the fixed resources for each executor at 4 cores and 8GB of memory (same configuration as before). The performance we see at this point, should be as close to the optimum as we can achieve on data ingestion, other than having them in memory already. So the performance we get at this point cuts minutes from our execution time and continues showing improvement after 16 executors, in contrast to what we observe in Figure 5.4, where performance actually starts to drop. We also can see a benefit that slightly decreases with scaling, but is still significant, with the best performance of the Kafka version being over a minute slower.

The same trend is reflected in Figure 5.7 with vertical scaling following a similar trend, regarding the improvement achieved by the additional resources, but showing a significant drop in the total cost when compared to Figure 5.5.

We need to bring the performance of our application when we use Kafka, as close as possible to that of the files. In order to do this we need to optimize the configuration of the ingestion path.

## 5.4 Optimization parameters for Kafka and Spark

In this section, we will present the parameters that we examined and we will explain what they are and in what way we believe they will affect the performance of Kafka. We tested each one in isolation (by keeping everything else the same) and in some cases we also examined them in relation to each other (i.e. partition number and level of replication).

There are parameters we need to examine on our Kafka server configuration and the way we setup our topic and there are parameters on the Kafka consumer side, which is the client API in our Spark application when we try to contact Kafka to receive data. We will explain the factors on the Kafka server side then we move on to the Kafka client.

Table 5.1: Key examined parameters, values, and summary of conclusions

| Parameter | Examined values | Short conclusion |
|---|---|---|
| `Number of partitions` | (Default =1)-5-10-**50**-100-500 | This parameter bounds the ingestion rate we can achieve. |
| `Replication` | (Default =1)-**3** | Replication increases writing cost by a little, but it reduces the number of failed tasks that need to rerun. |
| `Number of network threads` | (Default =3)-4-**6** | Increment as we increase the number of partitions. Affects how many requests can be handled at a time. |
| `Micro-batch size` | d=dataset-size, [d-d/2-d/4-d/10] | The micro-batch size affects the latency we see in getting the results. |
| `Timeout (ms)` | (Default =60000)-**90000**-120000 | By increasing this, we reduce the number of failed tasks we need to rerun. |
| `Consumer poll timeout (ms)` | (Default =512)-**1000**-1500 | By increasing this, we reduce the number of failed tasks we need to rerun due to broker's delayed replies. |
| `Garbage collection` | ConcGCThreads=**1** and ParallelGCThreads=**5** | Reduces the GC pauses we observe. |

### 5.4.1 Kafka server side

We start from the server side of things and what we can configure there. Some are manageable when we create a new Kafka topic, while others need to be set on the server configuration and then restart the service to make them take effect.

**Kafka partitions:** Topics are very important in Kafka, since they are our unit of parallelism and a topic is just a logical name for a collection of partitions. By default, each new topic has only 1 partition. Writes to different partitions are run in parallel, while each consumer is bounded by it, since they can only use as many consumer threads as the number of partitions. This leads us to the conclusion that the more partitions we have the higher throughput we can achieve. This is confirmed by our experiments. While producer throughput in general can reach 10s of MBs/sec based on this benchmark [2], the throughput that the consumer can achieve is highly dependent on the application.

The number of partitions can be increased later on. However, the messages that were already stored are not rebalanced across the new partitions and if a key is used to assign messages to partitions by hashing, then this could affect

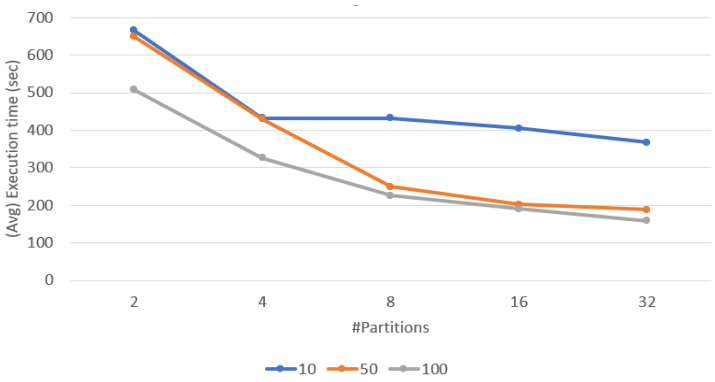



Figure 5.8: Scaling out for various partition sizes [10-50-100], using executors with 4 cores and 16G of memory.

the guarantee that Kafka gives, about routing events with the same key to the same partition, guarantying in-order delivery. This is why the safest practice is to over-partition from the start.

However, we need to consider some other aspects of picking a number of partitions. Each partition is mapped to a directory in the broker, which contains a file with the events and an index. More partitions means more files. This is not a very big issue for today's systems. However, if we use replication to increase availability, we also get the costs of synchronizing them across partitions. The producer and the consumer both need to connect to the leader partition, so with many partitions we also get the risk of high unavailability if a broker crashes, until the system rebalances. End-to-end latency is the time from the client publishing the data to the time they are consumed. Messages are only commited after all replicas of a partition have been updated. Therefore more partitions can mean higher end-to-end latencies. But still,normally, we need really high number of partitions to start seeing this effect.

We examine how increasing the number of partitions can affect performance. We test various numbers of partitions (Table 5.1), while scaling out our application to see if the performance will improve. Since the number of partitions determines the number of tasks that will be generated for Stage 1 (this is the unit of parallelism for Spark), we know that this parameter bounds our performance. We try an increasing number and in Figure 5.8 we see how each partitioning schemes scales horizontally.

Figure 4.8 shows how the execution time is affected by the number of executors [2-4-8-16-32] for 10, 50, 100 partitions. We can see that with 10 partitions we reach the maximum ingest rate at 4 executors. After that we have a very small improvement by scaling out our execution. For 50 partitions we reach a plateau at 16 executors, after which performance does not show significant improvement. From 50 to 100 partitions the behavior of our application does not change much, therefore we pick 50 as the number of partitions we will use in our deployment,

to avoid the cost of unnecessary partitions. We tested various other values and chose these as the most representative for our conclusions. When the number of partitions is equal or smaller than the number of tasks we can run in parallel (executors x cores), we cannot improve ingestion rates by much.

We have noticed that when we increase the number of partitions over 100, tasks start failing to connect and the job fails. This leads us to the next parameter we wanted to test.

**Partition replication:** When we create a topic we can set the number of replicas we want to be created of each partition. Since Kafka version 2.4.4 consumers can also read from the replica files, which was not possible before (read only from leader). Like we mentioned when we discussed the number of partitions, increasing the replicas, adds a synchronization cost, which we are able to handle.

By adding replication to our partitions we increase availability. If one of the brokers goes down or is temporarily unavailable due to networking issues, which is a phenomenon we have seen at times, our applications can use the replicas in the other brokers to access the data. The replication adds a small overhead when writing to a Kafka topic, but it is small enough to ignore.

In conclusion, by using replication we improve performance by reducing the number of failed tasks due to connection errors, which improves the overall execution time.

**Network threads:**, This parameter is set at the Kafka server configuration file and we need to restart the service for any changes to take effect. The default value is 3 threads and documentation suggests trying to increase it, if there is an expectation of too many connections or as we increase the number of partitions. We tried increasing it to 4 and 6, and we saw fewer failed tasks that needed to be repeated, improving overall performance this way.

Other parameters to improve broker performance include: message.max.bytes, num.io.threads, background.threads, queued.max.requests, socket.send.buffer.bytes, socket.receive.buffer.bytes, socket.request.max.bytes.

### 5.4.2 Kafka client side

There are several important parameters when configuring the Kafka consumer [18]. In Table 5.1 we mention some of them with their default values. Here we will only dive deeper into the ones that made a significant difference to our application's performance.

**Batch size:** By default, this value is not limited, meaning we can read as many offsets are available for each trigger interval. The size of our micro-batch can be determined either by defining a number of offsets to read per trigger or by using a time column in the data and setting a time-limit (e.g. TIMESTAMP = 1 day). This parameter is very important, since it determines how long we have to wait to see our results.

For example, in our case we can load our entire dataset as 1 micro-batch. This leads to the execution times we saw in Figure 5.4. If we break the data down to

smaller batches, we can get partial results much faster. By halving the batch size we get results in half the execution time. We tested this for various batch sizes and the batch execution time is analogous to the portion of the dataset we set as the batch size. The time it takes for the entire dataset to be processed shows very small differences.

However, since Kafka is not an idempotent writes sink, we see partial results for each batch. We need to deal with this downstream and take this into account with any application reading this stream. If we write to a key-value store this is not an issue, since each new batch will overwrite old values. With every additional batch we examine, we get a new result state object in the output, which may be the same as the previous one or it may have changed with the additional data.

**Timeout:** This parameter can be increased in order to avoid consumers losses. In some runs, when our network connections were unstable, we saw that jobs failed due to consumer timeouts. We tried increasing the default timeout and we saw a decrease in the number of failed jobs. However, in a smooth running scenario, the default value at 60000 ms works fine and doesn't need to be changed.

**Spark Kafka consumer poll timeout:** This is one of the most important configuration parameters of the Kafka consumer. This parameter is very sensitive, because it is the one that returns to Spark the records requested by a Kafka seek. If after two attempts there is a timeout, the task fails and is sent to another Spark executor which causes a delay. We saw this type of behavior a number of times in our experiments, due to ad connection for some executors.

If the parameter value is too low and the Kafka brokers need more time to answer, there will be many task failures, which causes a delay. However, if this timeout is too high, the executor wastes a lot of time doing nothing, which also causes large delays.

The default value is 512ms. Since we had many failed tasks we needed to raise that value and investigate why the Kafka brokers took so long to send the records to the poll. This was due to unstable network connections that failed at random times. Setting the timeout at 1000ms reduced the number of failed tasks.

**Garbage collection:** This is the final parameter we are going to mention. We noticed long GC pauses in the execution, so we experimented and tried setting the number of GC threads in the configuration. Our findings regarding garbage collection are not conclusive and therefore further work is going to be needed for a more in-depth analysis.

### 5.4.3 Elasticity

Streaming applications are by design long-running. However, the rate of incoming data may differ at different times (e.g. at different times of the day), leading to either wasted resources (if the incoming rate is too small) or long delays (if the incoming rate is too high). Therefore we need to adapt resources as needed, in order to limit resource waste in cases were we share cluster resources with others and to decrease costs if we pay for our executors. Ideally, we would like to be able

to do this dynamically, based on the incoming data rate.

**Dynamic resource allocation** is a feature Spark provides for some of its cluster managers (standalone, Mesos, Yarn); unfortunately it is not supported yet by Spark for the Kubernetes cluster manager. It is expected to be implemented in future versions, but until then a workaround is needed to avoid wasting resources or for managing higher loads, this could be true for any new cluster manager support added in the future. Since dynamic resource allocation is not available, we have to do this manually. Thanks to the fault tolerance feature in Spark Structured Streaming and Kafka's ability to restart from a given offset, we can stop a running query and restart it with more or less resources.

**How delivery semantics and batch size affect execution cost?** Specifically, Spark Structured Streaming guarantees *end-to-end exactly-once* or *at-least-once* delivery (in micro-batch mode) through the semantics applied to state management, data source and data sink. In order for this to occur, there are a few conditions. First of all, our source needs to be replayable. This is true for Kafka thanks to its offsets. We also need to add a checkpoint directory to the query, where Spark stores metadata about the offsets being processed on each batch in write-ahead-logs (WAL). This way, Spark guarantees that all the events in the source will be taken into account for the result. However, the exactly-once or at-least-once semantics depend on the type of sink we have chosen. In order to achieve exactly-once semantics we need an idempotent sink, like a key-value store that will overwrite any partial results. Kafka sinks, which we use, offer at-least once guarantees, since it is append only. Therefore, if after a restart a micro-batch has not been committed it will be rerun and after it is completed, Spark will move on to the next one.

In the *best case* scenario, our last micro-batch was just committed before the crash and the next one will start on the next offset available in the source as normal. In this case, we do not incur any additional latency, other than the cost of the restart. The restart cost is dependent on the system and how long it takes to provide the resources we asked for. In Table 5.2 we show a breakdown of the average cost of each part of the restart. Horizontal scaling takes longer, since we need to incur the startup cost for the container and Spark multiple times, making a slow node critical. The more executors we ask for, the more we increase the startup cost. Vertical scaling has the benefit of only needing to find a worker node with the requested resources and starting our executor there. Here we have the disadvantage that if we ask for a significant number of resources, there may be only a few nodes in the cluster that can accommodate the request, but if they are not available we may not be able to run our application.

So this was the best case scenario cost, where we only need to worry about restarting our application. However, if the last micro-batch was not committed before the crash, on startup we need to rerun it. This will be done automatically thanks to checkpointing. In this scenario our overall cost is the startup cost for the nodes, plus the cost of rerunning our latest micro-batch. The time it takes to run a micro-batch depends fully on how big our micro-batch is. If we have a

Table 5.2: Average cost for each phase of the restart process.

| Restart phase | Average cost |
|---|---|
| `Start driver node` | 4 sec |
| `Start executor nodes` | 4-42sec |
| `Start streaming query` | 23 sec |

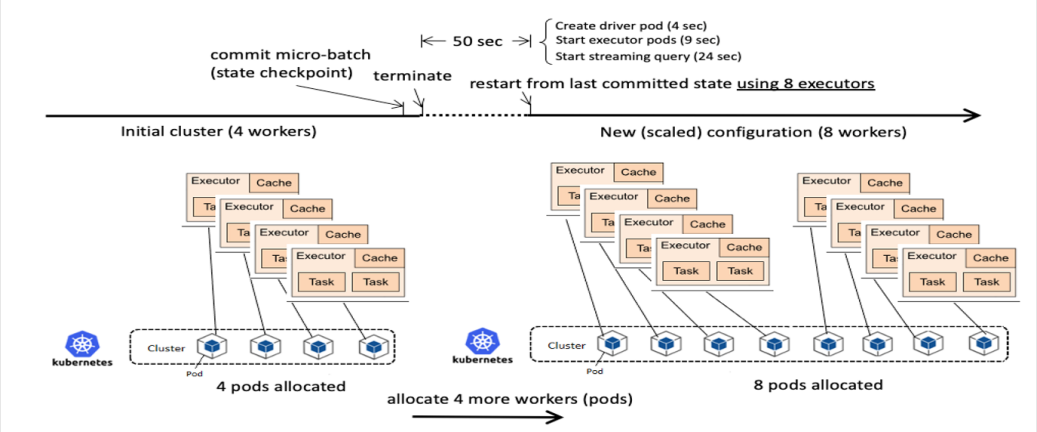



Figure 5.9: Transition from 4 to 8 executors via dynamic elasticity.

very small micro-batch (a few thousand offsets) this can take a few seconds or in the *worst case scenario* it can double our execution time if our micro-batch is the size of our entire dataset. This affects our overall latency and picking the right micro-batch size is a big decision.

**Proposed plan to minimize restart cost:** In order to perform a clean restart we can use action listeners that are provided by the Spark Structured Steaming API. The action listener has access to a number of query metrics like the throughput achieved for the batch, the number of input rows for the trigger, the execution time for various parts of the query, etc. We use the listeners to check if the execution of a micro-batch falls within acceptable parameters and terminate the query if it does not, logging if it needs more or less resources. We use a process that checks periodically (adjustable period) if the streaming application is running and relaunches it if it is not found, with the same or different configuration, depending on what the action listener has logged. This way we can automate somewhat the adaptive process and it can be used in the future for other cluster managers that do not support dynamic resource allocation as a workaround.

In Figure 5.9 we showcase the steps of this process and in Figure 5.10 we have used Grafana [11] to isolate the CPU usage for our executor nodes in the cluster and we can observe the behavior of the system when we try to scale out, with the method we mentioned above. We went from 4 to 8 executors and at 17:29:18 we shut down our application and relaunched with the extra resources. At 17:30:45 our streaming query logged starting a new batch. The time in between was 50

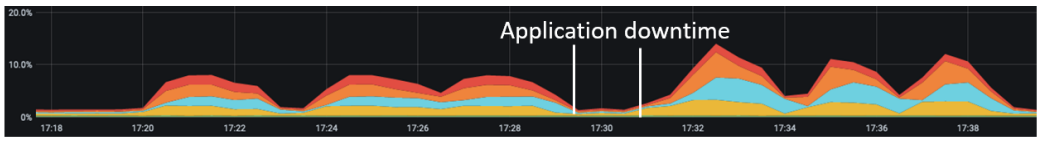



Figure 5.10: CPU usage for the executor nodes before and after a restart. Each color corresponds to 1 worker node.

seconds to detect the application failure and re-submit with double resources, 4 seconds for the driver pod to be created, 9 seconds to start the executor pods and 24 seconds for the application to start the streaming query (this time includes starting up the JVM for each executor, performing all the initial communication between the driver and the executor pods, initializing variables and loading the schema). The delay in the first two parts depends on the load of the cluster manager at the time. The delay in the third phase is fairly consistent (low variation) across different runs. The application submission and execution process has been further explained in section 5.1.

## 5.5 Results before and after tuning

In this section we discuss the final configuration parameters for our final deployment. We presented a number of parameters that we can optimize so far. The parameter values we tested are shown in Table 5.1 and the values we chose are shown in bold.

In summary, we picked 50 partitions for our Kafka source, because we saw a ceiling in the performance gain. If we try to scale our solution out further, we may see that more partitions make sense, but for our load and scalability goals this number made the most sense. We chose to test again on the entire dataset as our batch-size (not good in production) for comparison purposes. In order to reduce task failures due to networking issues, we increased the network threads at the producer to 6, the timeout to 90000(ms), the consumer poll timeout to 1000 and added replication to a factor of 3 replicas per partition. This way we managed to reduce reruns of failed tasks and therefore the overall execution time of each job. Finally, we experimented with garbage collection (GC) parameters and ended up with 1 concurrent GC thread and 5 parallel GC threads. This reduced GC pauses, which caused inexplicable delays in random jobs, by freezing executors.

After all this, we tested our initial application again and we show the side by side comparison of the horizontal scaling we tested in Section 5.2. In Figure 5.11 we show the comparison of the two executions. We can see that the performance is two times faster on average using the initial configuration (4 cores, 8GB memory) and 6 times faster on average, using the best vertical configuration (8 cores, 16GB memory). We also show how the final execution compares to the initial performance we showed using files in Section 5.3.

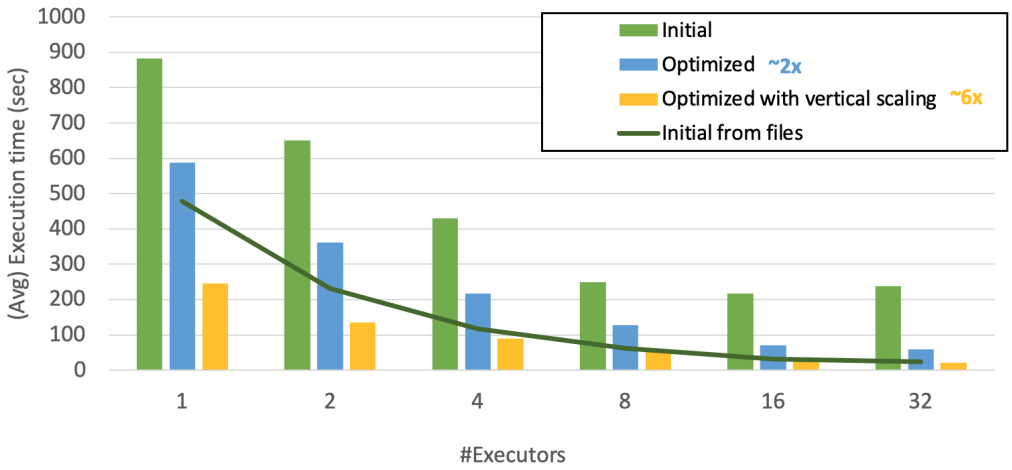



Figure 5.11: Final performance with the original executor setup (4 cores, 8GB memory) and the best executor setup (8 cores, 16GB memory), compared to the initial performance and the performance with ingestion from files (all with the original executor configuration).

## 5.6 Challenges

The main challenges faced in experimenting with scalability and performance tuning had to do with the constant evolution of the infrastructure during our tests, due to software and hardware updates, networking reconfigurations, and so on, straining an already complex performance engineering undertaking. Such challenges are however not unexpected in experimental (and to some extent production) testbeds and should be taken into account in planning a testing and tuning exercise such as undertaken in this chapter. We note that many of the measurements presented in this chapter had to be acquired through Spark logs rather than high-level performance monitoring tools such as Grafana [11], as the latter was only available during the last few months of this thesis.

## 5.7 Conclusions

We are not aware of other work that examines the effect of all these parameters on the performance of Spark structured streaming applications. In this chapter we presented a list of parameters that we consider the most important ones to be considered in the deployment of our applications. We explain which issues were addressed by each one and we compare our final configuration with the initial (out-of-the-box) one. In the end we managed to improve the performance of our applications by a factor of 2 using "thinner" executors and by a factor of 6 using the optimal executor configuration (8 cores and 16 GB memory), which underlines the importance of appropriate performance optimization and tuning.

Future work could extend the investigation presented in this chapter to gauge the impact of additional parameters. For example, the serialization of the events sent to Kafka is one aspect whose performance impact we did not examine in this work and it would be very interesting to see how it affects performance.

# Chapter 6

# Conclusions and future work

In summary, this work examined how we can leverage the real-time response benefits of stream-processing-systems (SPS) for applications that need fast response times. The conclusions and future work for each part of this work is mentioned at the end of each chapter and we are going to summarize it briefly here.

We described a streaming application for building dynamic profiles for users and exhibits, by processing localization and activity information/events in an exhibition space environment. Our experimental evaluation in a controlled testbed demonstrated that classification of users (key to effective personalized recommendations) into categories, utilizing insights from social studies, is possible through an online stream-processing platform using ML libraries. This work is on-going and as part of future work, a larger scale pilot will study the effectiveness of recommendations and further investigate the potential of other ML techniques such as reinforcement learning and artificial neural networks, to more aggressively explore the space of possible recommendations.

In the second streaming application, we used open GTFS-formatted [8, 9] localization data collected from buses, to predict ahead of time if the operator was going to violate their service-level objective. We built dynamic profiles for the trips, route segments and vehicles that were updated in real-time. We then utilized techniques from delay prediction and created a warning system that publishes warnings of anomalous behavior to a live dashboard (e.g. impending SLO violation, bus delay, missed stops, etc). However, the dataset we collected showed behavior that is not easily explained without the input of the provider or someone who knows the workings of the system, therefore such a collaboration, if possible, would be a great step for future work, in order to further test our system.

Finally, we showed how a structured streaming application with a Kafka source performed out-of-the box, trying scaling it out and up. We then presented a list of the most important parameters that need to be tuned for performance benefits and how each one affects the execution. We compared our initial, out-of-the-box execution, with the optimized one and found that with the right configuration we could get it to scale out 6 times faster. We also discuss how elasticity can be

achieved in a system such as ours, where dynamic resource allocation is not supported. We show the cost of downtime and describe a methodology to reallocate resources when needed with relatively small downtime. As part of future evaluation, we would also like to examine the effect of different types of serialization for the messages and do a more in-depth analysis of how to handle garbage collection pauses in more complex applications.

# Bibliography


[1] Apache Kafka. https://kafka.apache.org. [Online].

[2] Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[3] Benchmarking structured streaming on databricks runtime against state-of-the-art streaming systems. https://databricks.com/blog/2017/10/11/benchmarking-structured-streaming-on-databricks-runtime-against-state-of-the-a
html. [Online].

[4] Borealis team: Borealis application programmer's guide. http://www.cs.brown.edu/research/borealis/public/.

[5] A collection of public transport network data sets for 25 cities. https://doi.org/10.5281/zenodo.1186215. [Online].

[6] Complex event processing cql language reference. https://docs.oracle.com/cd/E12839_01/apirefs.1111/e12048/toc.htm. [Online].

[7] Esper documentation. https://www.espertech.com/esper/esper-documentation/. [Online].

[8] General transit feed specification (gtfs). https://gtfs.org/. [Online].

[9] General transit feed specification (gtfs) realtime v2. https://gtfs.org/reference/realtime/v2. [Online].

[10] Google transit data feeds. https://code.google.com/archive/p/googletransitdatafeed/wikis/PublicFeeds.wiki. [Online].

[11] Grafana. https://grafana.com.

[12] Open mobility data. https://transitfeeds.com/. [Online].

[13] Roma mobilita open data. https://romamobilita.it/it/tecnologie/open-data. [Online].

[14] "romamobilita.it: Technology section". https://romamobilita.it/it/tecnologie.

[15] Spark cluster overview. https://spark.apache.org/docs/latest/cluster-overview.html. [Online].

[16] Spark structured streaming programming guide overview. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html. [Online].

[17] Statista. https://www.statista.com/statistics/871513/worldwide-data-created/.

[18] Structured Streaming and Kafka Integration Guide. https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html.

[19] Jawwad Ahmed, Andreas Johnsson, Rerngvit Yanggratoke, John Ardelius, Christofer Flinta, and Rolf Stadler. Predicting sla violations in real time using online machine learning, 2015.

[20] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, August 2015.

[21] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. *STREAM: The Stanford Data Stream Management System*, pages 317–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[22] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 601–613, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Margianna Arvaniti-Bofili, Stelios Gkouskos, Konstantinos Kalampokis, Gerasimos Papaioannou, Georgios Gogolos, Ilias Chaldeakis, Konstantinos Petridis, and Kostas Magoutis. Towards personalized recommendations in an iot-driven tour guide platform. In Schahram Dustdar, editor, *Service-Oriented Computing*, pages 3–11, Cham, 2020. Springer International Publishing.

[24] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez. Recommender systems survey. *Journal of Knowledge-Based Systems (Elsevier)*, 46:109–132, July 2013.

[25] Poonam B.Thorat, R. Goudar, and Sunita Barve. Survey on collaborative filtering, content-based filtering and hybrid recommendation system. *International Journal of Computer Applications*, 110:31–36, 01 2015.

[26] P. Carbone, Asterios Katsifodimos, Stephan Ewen, V. Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[27] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, page 668, New York, NY, USA, 2003. Association for Computing Machinery.

[28] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *In CIDR*, 2003.

[29] Steven I-Jy Chien, Yuqing Ding, and Chienhung Wei. Dynamic bus arrival time prediction with artificial neural networks. *Journal of Transportation Engineering-asce - J TRANSP ENG-ASCE*, 128, 09 2002.

[30] Steven I-Jy Chien and Chandra Mouly Kuchipudi. Dynamic travel time prediction with real-time and historic data. *Journal of transportation engineering*, 129(6):608–616, 2003.

[31] A. Chronarakis, S. Gkouskos, K. Kalampokis, G. Papaioannou, X. Agalliadou, I. Chaldeakis, and K. Magoutis. ProxiTour: A Smart Platform for Personalized Touring. In *Proc. of 13th Symposium and Summer School on Service-Oriented Computing (SummerSOC'19); also IBM Technical Report RC25685*, Hersonissos, Crete, Greece, June 17-23, 2019.

[32] David DEAN. Museum exhibition theory and practice. london: Routledge. 1994.

[33] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[34] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. A survey on the evolution of stream processing systems, 2020.

[35] Howard GARDNER. Dimenze myšlení: Teorie rozmanitých inteligencí. edition 1. praha: Portál. page 398, 1999.

[36] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications)*. Springer-Verlag, Berlin, Heidelberg, 2007.

[37] M. Hatala and R. Wakkary. Ontology-based user modeling in an augmented audio reality system for museums. *User Modeling and User-Adapted Interaction*, 15:339–380, 2005.

[38] Guenter Hesse and Martin Lorenz. Conceptual survey on data stream processing systems. pages 797–802, 12 2015.

[39] Ryan Holeywell. Top Reasons People Stop Using Public Transit. `https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html`, 2013.

[40] Eilean HOOPER-GREENHILL. The educational role of the museum. edition 2. page 346, 1999.

[41] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan. A survey of distributed data stream processing frameworks. *IEEE Access*, 7:154300–154316, 2019.

[42] Ran Hee Jeong. The prediction of bus arrival time using automatic vehicle location systems data. 2004.

[43] Jiann-Shiou Yang. Travel time prediction using the gps test vehicle and kalman filtering techniques. In *Proceedings of the 2005, American Control Conference, 2005.*, pages 2128–2133 vol. 3, 2005.

[44] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518, 2018.

[45] Jonathan Leibiusky, Gabriel Eisbruch, and Dario Simonassi. *Getting Started with Storm*. O'Reilly Media, Inc., 2012.

[46] Edward Leigh. So there you have it: don't depend on buses. `https://www.smartertransport.uk/so-there-you-have-it-dont-depend-on-buses/`, 2013.

[47] Philipp Leitner, Branimir Wetzstein, Florian Rosenberg, Anton Michlmayr, Schahram Dustdar, and Frank Leymann. Runtime prediction of service level agreement violations for composite services. *In: Proceedings of the 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing, co-located with ICSOC 2009*, 11 2009.

[48] Li Weigang, W. Koendjbiharie, R. C. de M Juca, Y. Yamashita, and A. Maciver. Algorithms for estimating bus arrival times using gps data. In *Proceedings. The IEEE 5th International Conference on Intelligent Transportation Systems*, pages 868–873, 2002.

[49] Bernice a Dennis MCCARTHY MCCARTHY. Teaching around the 4mat cycle: designing instruction for diverse learners with diverse learning styles. thousand oaks, calif.: Corwin press. page 102, 2006.

[50] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[51] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful scalable stream processing at linkedin. *Proc. VLDB Endow.*, 10(12):1634–1645, August 2017.

[52] Fernando Ortega, José-Luis Sánchez, Jesús Bobadilla, and Abraham Gutiérrez. Improving collaborative filtering-based recommender systems results using pareto dominance. *Information Sciences*, 239:50 – 61, 2013.

[53] Jayakrishna Patnaik, Steven I-Jy Chien, and Athanassios Bladikas. Estimation of bus arrival times using apc data. *Journal of Public Transportation*, 7, 03 2004.

[54] Md. Golam Rashed, Ryota Suzuki, Takuya Yonezawa, Antony Lam, Yoshinori Kobayashi, and Yoshinori Kuno. Tracking visitors in a real museum for behavioral analysis. pages 80–85, 08 2016.

[55] Burke Robin. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12:331 – 370, 2002.

[56] Amer Shalaby, Ph. D, P. Eng, and Ali Farhan M. A. Sc. Bus travel time prediction model for dynamic operations control and passenger information systems, 2002.

[57] D. Stamatakis, D. Grammenos, and K. Magoutis. Real-Time Analysis of Localization Data Streams for Ambient Intelligence Environments. In *Proc. of 2nd Int. Conf. on Ambient Intelligence (AmI'11)*, pages 92–97, Amsterdam, The Netherlands, 2011.

[58] Dihua Sun, Hong Luo, Liping Fu, Weining Liu, Xiaoyong Liao, and Min Zhao. Predicting bus arrival time on the basis of global positioning system data. *Transportation Research Record*, 2034(1):62–72, 2007.

[59] Fangzhou Sun, Abhishek Dubey, Jules White, and Aniruddha Gokhale. Transit-hub: a smart public transportation decision support system with multi-timescale analytical services. *Cluster Computing*, 22, 01 2019.

[60] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. *SIGMOD Rec.*, 21(2):321–330, June 1992.

[61] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[62] E. Veron and Levasseur. Ethnographie de l'exposition, Paris, Bibliothèque Publique d'Information. *Centre Georges Pompidou*, 1983.

[63] Steven S. Yalowitz and Kerry Bronnenkant. Timing and tracking: Unlocking visitor behavior. *Visitor Studies*, 12(1):47–64, 2009.

[64] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 423–438, New York, NY, USA, 2013. Association for Computing Machinery.