ALMA MATER STUDIORUM — UNIVERSITY OF BOLOGNA

School of Engineering
Master's Degree in Computer Engineering

EDGE CLOUD COMPUTING FOR GEOSPATIAL DATA PROCESSING AND APPROXIMATE QUERIES

Thesis in
Mobile Systems M

Advisor
Prof. Dr. Paolo Bellavista

Co-advisors
Dr. Isam Al Jawarneh
Prof. Dr. Luca Foschini

Presented by
Lorenzo Felletti

Third Graduation Session
Academic Year 2021-2022

KEYWORDS

Edge Computing
Geospatial
Geohashing
Kafka
Spark

To all the beautiful people that helped me along the way.

Index

List of Figures

Introduction

Nowadays, huge volumes of data, and particularly geospatial data, are continuously collected for all sorts of reasons, such as traffic patterns analysis, path optimization problems, environmental mapping, and weather forecasting.

With the advent of IoT, the amount of data that are collected will grow even more, due to the ease of deployment of IoT devices, and the use of Internet technologies and protocols for communication.

One of the major challenges of IoT big data collection is posed by the fact that often incoming data need to be cleaned, prepared, or more generally preprocessed, before performing the actual data analysis.

During the past decade, cloud technologies became increasingly utilized by companies all around the world, that moved their on-premise resources and workloads more and more to cloud providers, because of the many benefits that they offer.

Among the workloads that are suitable the most for cloud environments, there are the data analysis workloads, that are usually performed by clusters of computers, and cloud providers enable great scalability of clusters when needed, while also making customer pay only for what they use. Thus, companies may leverage public clouds to both carry out their data analysis tasks faster, and keep costs down by not paying for resources while not using them, limiting the problem of having to over-provision on-premises resources in case they are needed at a later time.

In these contexts, frameworks leveraging clustering shine the most, even more when they are open-source, so to avoid vendor lock-in problems for companies in the long run.

Moreover, analyzing such huge volumes of incoming data sometimes may be either unfeasible, too costly, unnecessary, or a combination of these. To help solve these problems, the introduction of lightweight nodes, capable of performing task-specific data preprocessing and sampling technique, located nearer the data origin source, may be beneficial.

These nodes are known as edge cloud nodes, since they are placed not in the cloud core, a "distant" location in which powerful nodes perform the heavy processing, but "near" data origins, and acts as intermediate nodes, placed between the data source and the core of the cloud, in the pipeline of moving data.

Such nodes can carry out simple tasks only on modest amounts of data, but this is not a problem, since only data coming from nearby sources are received and processed by each of them.

This is an increasingly popular architectural choice, and we wanted to leverage it to offload cloud nodes of part of the workload, optimizing the overall architecture performances.

Geospatial data analysis queries often make use of location approximation techniques to lower the intensive computational work required to establish if a point is located inside an area, a very common task performed by geospatial queries. One of the most used techniques to do so relies on the use of Geohash, which is a small string that uniquely identifies a rectangular area on the Earth's surface. Geohash may be quickly calculated starting from a latitude longitude pair. Since all points inside this rectangle (whose size depends on the hash length) are identified by the same Geohash, a Geohash can be used to aggregate all the points belonging to the same area.

The scenario analyzed in this thesis was the one in which taxis — equipped with an IoT device capable of sending information about the taxi speed and position at a specific time — were moving in an urban scenario, specifically in the city of Shenzhen in China.

Shenzhen is one of the biggest cities in the south of China, with a population of more than seventeen million people. In this kind of scenario, it is easy to imagine that huge volumes of geospatial data are produced daily.

Our idea was to introduce edge nodes capable of calculating the Geohash of incoming data, and using it to perform simple data sampling and aggregation tasks based on it.

Moreover, complex polygons, such as polygons describing the shape of a neighborhood of a city, can be approximated with a list of Geohashes, and locating a point inside a polygon, with a certain precision, becomes as simple as consulting a hash table.

One of our objectives, was to elaborate a strategy to exploit this simple idea to develop a way to distribute incoming data in a spatially-aware manner.

In particular, the edge nodes we developed, were not only capable to efficiently sample data based on their geohash, by leveraging the sampling

technique known as stratified sampling, but also to distribute data to specific message queues, each with the peculiarity of containing only messages coming from the same neighborhood (or area) of a city.

We call this technique spatial-aware data distribution, as data are sent to message queues, and Apache Kafka topics were used for this purpose, that are specific only to data coming from the same area.

Finally, the setup was deployed using containerization platforms, in particular by using Docker, to achieve portability and horizontal scaling capabilities offered by containers, although we did not focus on dynamically scaling the number of deployed edge nodes, as our work was primarily aimed toward prototyping the overall data pipeline architecture: from the data origin, passing through the edge nodes for preprocessing, to the data analysis carried out by cloud nodes, leveraging Apache Spark to perform the analysis.

Chapter 1

Motivations and State of the Art

The purpose of this chapter is to provide a gentle introduction to the motivations and state of the art regarding the topics that were treated in this thesis work.

Huge volumes of geospatial data are collected daily from a variety of different sources and for the most diverse of purposes. Transforming this continuously arriving raw data into timely insights is critical for many modern online services. For such settings, the traditional form of data analytics over the entire dataset could be prohibitively limiting and expensive for supporting real-time stream analytics[12].

This thesis work aims to provide a novel approach, leveraging edge cloud nodes computing capabilities, and the performant spatial queries (e.g. Point-in-Polygon) achievable by using the approximate location encoding technique of Geohash to enhance the traditional setup of a cloud cluster performing both data preprocessing and data analysis.

We propose a solution in which edge nodes are added to the architecture, and acts as a kind of "smart gateway" for incoming data directed to the cloud.

These nodes should be capable of encoding the incoming data location with a hash string, representing the approximate location of origin of the message, calculated from its latitude and longitude. The algorithm we used to encode incoming messages is known as Geohash, and is a common choice when spatial tessellation techniques are needed.

We also developed a novel way — we called it spatially-aware data distribution — to distribute data arriving to message queues having the peculiarity of containing only messages coming from the same neighborhood. This was achievable thanks to the relatively low computational cost of finding the neighborhood of a message obtained with the approximation of the neighborhood

polygon with a set of geohashes composing it, kind of like a composing a mosaic.

These were, in short, the objectives we wanted to achieve with this thesis. The remaining of the chapter, is a gentle introduction of the reader to the topics of geospatial data processing, geohashing, the concepts behind cloud and edge computing, and a brief overview of two techniques widely used in cloud environments, i.e. virtual machines, and containers.

1.1 Geospatial Data Processing

In this section, the topic of *geospatial data processing* will be discussed in depth.

Today's proliferation of ubiquitous positioning devices and technologies has simplified the collection of spatial data at an exponential rate[13]. Also, the large-scale spread of mobile devices, such as smartphones and sensor-enabled devices, has encouraged the participatory collection of a massive amount of geospatial data, specifically in the so-called Smart Cities[14]. Consequently, the demand for analyzing big spatial data volumes has become increasingly important to extract information and knowledge that facilitate better decision-making, which is beneficial for a variety of different fields.

Geospatial data processing involves the collection, analysis, and visualization of data related to geographic locations on the earth's surface. This type of data is often represented in the form of maps, satellite images, or aerial photographs, and can include information about terrain, vegetation, land use, demographics, and more. In particular, geospatial data and geoinformatics technologies are expected to play an essential role in supporting smart city strategies[15].

Geospatial data processing refers to the application of mathematical and computational methods to data that is spatially-referenced. This process involves collecting, organizing, processing, analyzing, visualizing, and interpreting geographic information. Geospatial data processing can be used in a wide range of applications, such as tracking changes in land use patterns, predicting the spread of disease, analyzing social media activity in different locations, or understanding the movement of vehicles in an urban scenario.

In urban scenarios, geospatial data processing can be used to analyze the movement of vehicles (such as taxis), which can provide insights into traffic patterns, demand for transportation services, and the impact of events or construction on mobility. By using geospatial data processing techniques,

researchers can identify hotspots for pick-ups and drop-offs, analyze travel times and distances, and explore the factors that influence taxi usage in different areas of a city.

One way to analyze taxi movement in an urban scenario is by using GPS data, which can be collected from sensors in the vehicles themselves. This data can be processed using geohashes, allowing researchers to aggregate data points and identify patterns in taxi movement across different areas of a city. By visualizing this data on a map, it is possible to identify congestion hotspots and explore the factors that contribute to them.

Geospatial data processing is then a critical tool for analyzing and interpreting spatially-referenced data in a wide range of applications, including understanding the movement of vehicles in an urban scenario, such as taxis. By using geohashes and other geospatial data processing techniques, it is possible to gain valuable insights into transportation patterns, traffic congestion, and other factors that affect urban mobility.

1.1.1 Raster and Vector Data

Geospatial data can be categorized into one of two categories:

- **Vector data** - Vector data represents geographic data symbolized as points, lines, or polygons[16]
- **Raster data** - represents geographic data as a matrix of cells that each contains an attribute value[16].

Vector data can be as simple as GPS coordinates of latitude and longitude, but also more complex geometries, such as the polygon that delimit a district of a metropolis. In the context of this thesis, vector data are used in two different scenarios:

- first, the incoming data are tagged with latitude and longitude coordinates
- then, to categorize each point as coming from a specific neighborhood, the polygons making up each city's district were used.

An example of raster data is, on the other hand, the Geohash (which will be discussed deeply in the next section) of a specific coordinate: in fact, geohashes divides a map into a discrete number of squares, each identified by a hash. Thus, raster data were also used to carry out the thesis' work.

As will be shown in the continuation of this document, vector data can be converted into raster data, and vice-versa (obviously, with the loss of some accuracy in this second case).

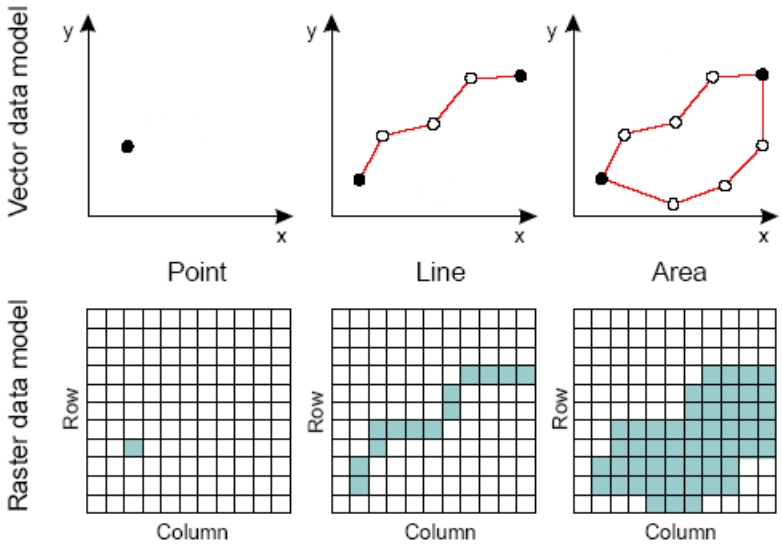

Figure 1.1: The figure shows the difference between vector and raster data models.[1]

GeoJSON Format

GeoJSON is an open standard format designed for representing simple geographical features, along with their non-spatial attributes. It is based on the JSON format[17]. It is a suitable format to represent vector data.

The features that one can represent with GeoJSON includes points, lines, polygons, and multipart collections of these.

In particular, polygons are a suitable feature to describe the topology of a neighborhood or district.

The following code snippet shows an example of a GeoJSON file:

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [122.0, 0.9]
      },
      "properties": {
        "prop0": "value0"
      }
```

```
    },
    {
      "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [122.0, 0.5],
          [123.1, 1.4],
          [124.0, 0.1],
          [125.0, 1.2]
        ]
      },
      "properties": {
        "prop0": "value0",
        "prop1": 1.0
      }
    }
  ]
}
```

Listing 1.1: Example of a GeoJSON file.

Some of the main fields of GeoJSON include:

- Type - this field represents the type of GeoJSON object. It can be "Feature", "FeatureCollection", "Point", "LineString", "Polygon", and so on
- Geometry - this field contains the actual geometrical data in the form of coordinates for different types of objects such as Point, LineString, Polygon, and others
- Properties - this field holds the attributes or properties of the GeoJSON object. It may include fields such as name, population, elevation, and so on, depending on the data requirements
- Crs - this field provides the coordinate reference system used by the GeoJSON object. It includes information about the projection system, the datum, and other spatial parameters
- Bbox - this field specifies the bounding box of the GeoJSON object. It encloses the entire extent of the feature, which helps to optimize the rendering of maps and other visualizations.

Overall, GeoJSON is a useful format for representing geographical data across different web platforms and applications. It is a lightweight, text-based format that can be easily parsed and manipulated using various programming languages and tools.

1.1.2 Geohashing

Geohashing is a spatial tessellation technique used to identify coordinates that are "near" with a unique identifier. The more "similar" two geohash are (i.e. the more prefix they share), the more two coordinates are spatially close to each other. It is a powerful technique, because it enables to make coarse-grained comparison between coordinates in a fast way, certainly faster than calculating the exact distance between two points. Obviously, by using geohashing techniques, one loses on the precision of the results, but oftentimes, the performance gains of geohashing vastly overweight the loss in accuracy.

Geohash is a public domain geocode system invented in 2008 by Gustavo Niemeyer[18]. Geohash encodes a geographic location into a short alphanumeric string (generally 4/6 characters long).

The main idea is to divide a map into a square grid, and have a function that maps each point to the square they are contained into.

One of the guarantees offered by geohashes is that the longer the shared prefix of two geohash is, the spatially closer the points are.
In this sense, geohashes are fundamentally different from the hashes used in cryptography, where two similar input string ("spatially close") need to have a very different hash in order for the hash to be secure.

Geohashes have a wide range of applications, such as location-based search, geographic clustering, and real-time tracking of mobile devices.

How Do Geohashes Work

Geohashes work by dividing the Earth's surface into a grid of cells, each with a unique code (geohash). The grid size is determined by the number of characters in the output string (the geohash). A geohash with more characters represents a smaller cell, thus a more precise location, while a geohash with fewer characters represents a larger cell, thus a less precise location.

To create a geohash for a specific location, the latitude and longitude of that location are first converted into binary format. The binary strings are then interleaved to form a single binary string, which is then converted into base-32 using a predefined set of characters. The resulting string is the geohash for that location.

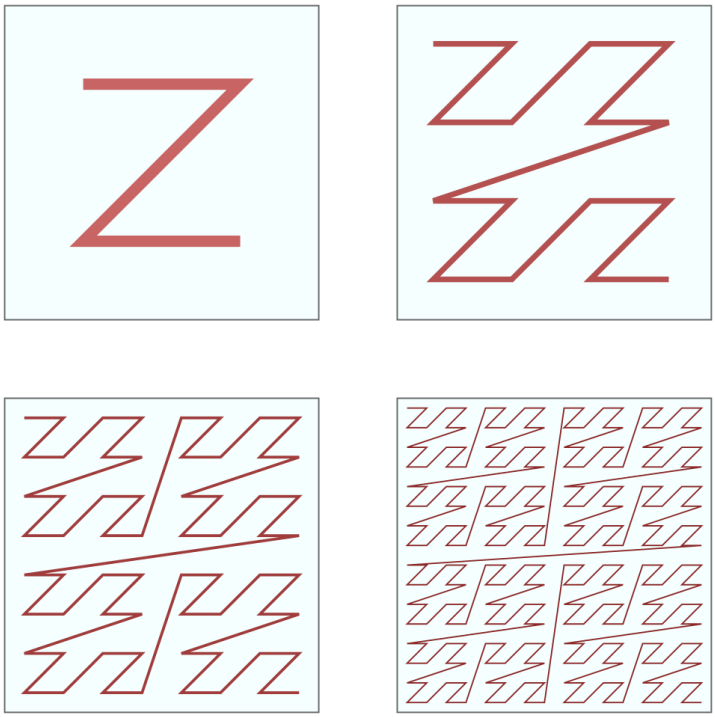

Figure 1.2: Four iterations of the Z-order curve.[2]

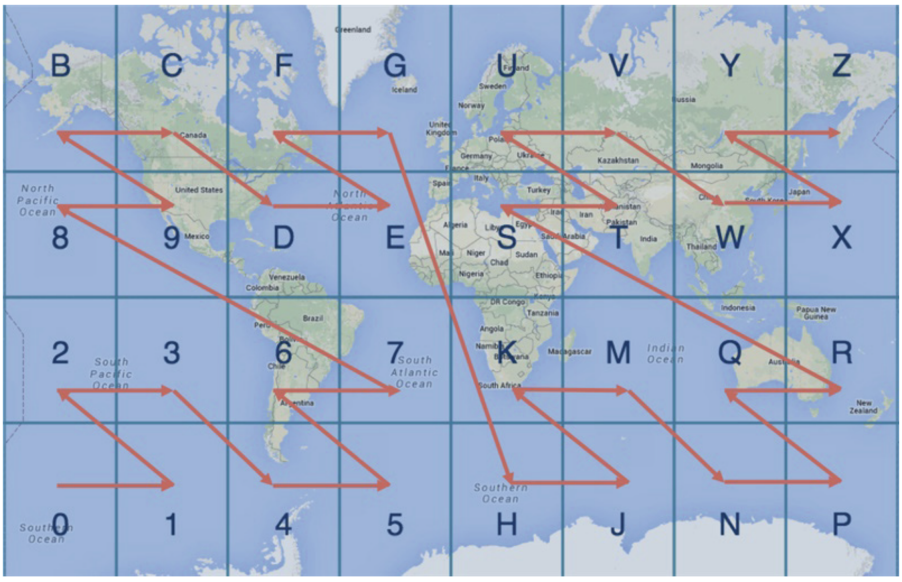

Figure 1.3: Z-order traversal in Geohash.[3]

Base-32 encoding																
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base-32	0	1	2	3	4	5	6	7	8	9	b	c	d	e	f	g
Decimal	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Base-32	h	j	k	m	n	p	q	r	s	t	u	v	w	x	y	z

Table 1.1: Geohash base-32.

Geohash precision table			
geohash length	lat error	lon error	km error
1	±23	±23	±2,500 km
2	±2.8	±5.6	±630 km
3	±0.70	±0.70	±78 km
4	±0.087	±0.18	±20 km
5	±0.022	±0.022	±2.4 km
6	±0.0027	±0.0055	±0.61 km
7	±0.00068	±0.00068	±0.076 km
8	±0.000085	±0.00017	±0.019 km

Table 1.2: Precision for different lengths of a Geohash.

The Geohash encoding algorithm builds a geohash from a pair of coordinates (latitude and longitude) using the base-32 character mapping shown in Table 1.1.

The maximum length of a geohash is twelve (12) characters.[19]

Without delving too deep into the implementation details of Geohash, it is sufficient to know that the computation uses alternately the longitude and the latitude of the coordinate to calculate the geohash and, by applying more iterations of the algorithm, the precision increases.

By increasing the geohash length by one character, since geohash is base-32 encoded you can locate a coordinate (identified by latitude, and longitude) inside a square that is $\frac{1}{32}$ the area of the original one. Table 1.2 shows the precision for each Geohash level.

Despite its numerous advantages, Geohash has some limitations, including:

- **Edge-case problem** - Locations that are located on the opposite sides of the 180-degree meridian, although spatially close, do not share a common geohash prefix (as one can note in Figure 1.3. The same happens with points close together at the North and South poles

- **Z-order problem** - problem related to Z-order traversal. The problem is that, because of the traversal order, some points having a common prefix,

may not be spatially close (such as points R and S in Figure 1.3.

Other Spatial Tessellation Techniques

Geohash is not the only spatial tessellation technique available, nor the only one used nowadays. In fact, there are other techniques available, such as Voronoi, and the choice between a technique or another is often based on the origin and use of geospatial data one wants to do. In particular, the paper *Taxi Demand Forecasting: A HEDGE-Based Tessellation Strategy for Improved Accuracy*[20] compares the geohash and the Voronoi technique, benchmarking the two in various scenarios. The paper findings are that there is not a clear winner between the two techniques, in particular the authors found that:

> "[...] models based on Voronoi had superior performance over models based on Geohash when the demand density was low, and vice versa. While the Voronoi tessellation appeared to be the recommended strategy for tessellating Bengaluru, the Geohash tessellation was the suggested strategy for New York City. We concluded that the performance of the chosen tessellation strategy is dependent on the demand density in each partition, and the geography of the city."[20].

Moreover, the Master's thesis *Partitioning of Spatial Data in Publish-Subscribe Messaging Systems*[21] showed how using pub-sub messaging systems in combination with Geohash often is more beneficial than using pub-sub messaging systems in combination with Voronoi to achieve good performances, and thus a greater ability to scale, in filtering GIS data. In particular, quoting the thesis:

> "Our experiments show that Geohash was able to achieve better results than Voronoi in general."[21]

1.1.3 Processing and Querying GIS Data Efficiently

In general, working with geospatial (or GIS) data is very complex, due to the compute-intensive operations required to, for example, determine if a point is inside or outside a polygon of complex shape, such as the polygon delimiting a district of a city. But, more often than not, we are not as much interested in having the maximum precision as much as we are in analyzing huge amounts of data quickly.

It is right in these scenarios where using Geohash can become useful to our objectives: in fact, by sacrificing some degree of accuracy, we can achieve huge gains in performance.

One use case exploiting both Geohash (and thus raster data) and GeoJSON (i.e. vector data) in combination to achieve significant performance gains is the following: image there is the need to categorize incoming data, spatially referenced by latitude and longitude coordinates, to the neighborhood to which it belongs. To achieve such a result, we could intersect each incoming point with the polygon describing each neighborhood, until we find the one to which the data belongs to. This is an approach that can allow us to achieve our goal, but it is very compute intensive. Instead, we could define a degree of precision that is enough for our use case, and once defined it, find a geohash length level that meet our needs. Once defined the geohash length of our interest, we can precompute the list of all the geohashes that belongs to a neighborhood, and then, for each incoming data, compute its geohash (which is a very fast operation), and then find the neighborhood to which it belongs to (it is sufficient to find the neighborhood whose geohashes list contains the incoming data's geohash).

We could be also more efficient by reversing the map mapping each neighborhood to a list of geohashes, and create a map mapping each geohash to the neighborhood it belongs to. With such a map, it is possible to find the neighborhood from a geohash in $\mathcal{O}(1)$.

Obviously, the gain in performance comes at the expense of precision, to provide an example, by using geohashes 6 characters long, each point is identified with a precision of 610 meters, which, depending on the dimension of the area taken into account, may or may not be a sufficient precision. Although 610 meters may seem like a too big area, the Manhattan borough comprehends 59.1 square kilometers of land[22], thus, in cities as big as New York, 6 characters long geohashes may be sufficient. And anyway, if still 6 character is not enough precise, by just adding one character, the precision becomes of 76 meters.

One interesting point to highlight of the described scenario, is the cooperation of both raster (geohash) and vector (the neighborhood polygon) data that was used to achieve the desired result.

1.2 Introduction to Cloud and Edge Computing

Geospatial big data processing is a critical component in a wide range of applications, such as urban planning, environmental monitoring, and transportation management. These applications require processing huge amounts of data generated from various sources such as satellite imagery, vehicles onboarded with GPS, and sensor networks.

Cloud and edge computing are two prominent paradigms in the field of

distributed computing, suitable for processing huge amounts of data. Both approaches provide different solutions to the challenges of managing and processing large amounts of data in a distributed environment. **Cloud computing** is a *centralized* model where computing resources such as servers, storage, and software applications are provided over the internet. In contrast, **edge computing** is a *decentralized* model where computing resources are located closer to the data source or end-user.

In recent years, the proliferation of the Internet of Things (IoT) and the need for real-time processing has led to an increase in the adoption of edge computing.

This section aims to provide an overview of cloud and edge computing, what they are and what they offer, what are their main advantages and drawbacks, to the reader.

1.2.1 Brief Mention to IoT Devices

The term Internet of Things (IoT) refers to the billions of physical objects, also called "things", that are connected to the internet, collecting and exchanging data with other devices and systems over the internet. IoT devices are physical objects that "sense", in the broadest of meanings, things going on in the physical world, and are able to transmit that information via internet connectivity.

Thus, IoT devices must have some (more or less limited) computing capacity, and the ability to send data on a network at least, in order to be considered such.

IoT devices range from small sensors meant for the general public, such as the smart objects many uses for house automation, to complex and sophisticated industrial tools.

IoT device management helps companies and organizations integrate, organize, monitor and remotely manage internet-enabled devices at scale. But, since for its own nature IoT devices are very different, and communicate with very different services and devices, this poses a challenge on the developers, often of different teams, to keep things working and interacting smoothly. To this purpose, there is a need for standard, or at least well-defined, communication interfaces and frameworks, to help teams and organizations manage IoT scenarios without too much effort being put on configuration and communication, so that they concentrate on the actual business logic of their application. In this sense, the internet protocols come in great help, but alone they are not sufficient: technologies such as Apache Kafka, RabbitMQ, or other Message Oriented Middlewares can further help developers of the different parts of an IoT scenario keep their applications as decoupled as possible from other teams'

choices.

In the context of geospatial data, an IoT device could be as simple as a microcontroller installed on a vehicle that continuously reads the GPS position, and sends this information to the cloud, leveraging the internet.

In the context of this thesis, we can imagine having installed on some taxis an IoT device that can read the vehicle speed and position, and can send this information to a Kafka topic via internet connectivity.

1.2.2 Cloud Computing

Cloud computing is the on-demand availability of computer system resources, especially data storage (cloud storage) and computing power, without direct active management by the user. Large clouds often have functions distributed over multiple locations, each of which is a data center[23].

Through the cloud users can access data, services, software, and hardware remotely leveraging their devices (smartphones, tablets, desktop PCs, etc.) internet connectivity. Instead of being forced to buy increasingly more powerful devices, users can exploit cloud resources, available on-demand, accessible, as already stated, via internet. But individuals are not the only users of the clouds, and arguably not even the one that benefits the most out of it.

Indeed, Cloud computing is increasingly used by organizations of any type and size, because of the many advantages it provides, such as big data analytics, disaster recovery capabilities, scalable deployments, less infrastructural work to be done, load balancing, and so on.

At the present day, there are three main different deployment models for cloud computing:

- **Public cloud** - a set of hardware, networking, storage, computing power, applications, and more, operated by a third party, and offered as a service (with a variable degree of management carried out by the third party offering the service). The three main public cloud providers on the market by share are Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform. One of the big advantages of the public cloud is that the provider takes on its side a part of infrastructure management and scaling that can help people to develop production-grade applications faster. Another great advantage offered by public cloud is that there is little need for a priori provisioning of resources of the right size; with public cloud, one can start small and scale up or down as needed. The main drawback of public cloud providers is the vendor lock-in, i.e. the difficulty of moving the deployments from one cloud provider to another, and this can potentially be leveraged by the cloud provider.

- **Private cloud** - the same set of hardware, networking, and so on, offered by the public cloud, but owned and operated by the organization itself. This gives the maximum flexibility of choice to organizations, but also charges them with the potentially overwhelming burden of having to correctly provision, manage, update, upgrade, keep secure their private cloud.
- **Hybrid cloud** - a combination of public and private cloud that is often the middle ground operated by companies and organizations.

There is another option, that somewhat sits in the middle of the one described above, that is multicloud. A multicloud environment consists of multiple cloud services offered by different providers, both public and private. Noteworthy is that all hybrid clouds are multicloud, but not all multicloud environments are hybrid clouds. Multicloud environments become hybrid only when the various cloud services are connected through an integration or orchestration system. A multicloud environment can be intentionally created to gain more effective control over sensitive data or to have redundant storage space that ensures better recovery in case of emergency. Thus, if services are not interconnected, they remain "separate", for example there can be a private cloud environment, and another public cloud environment, both owned by the same organization, but totally independent of one another. In this example, the organization operates on multiple clouds, and thus leverage multicloud, but this does not configure in a hybrid cloud environment.

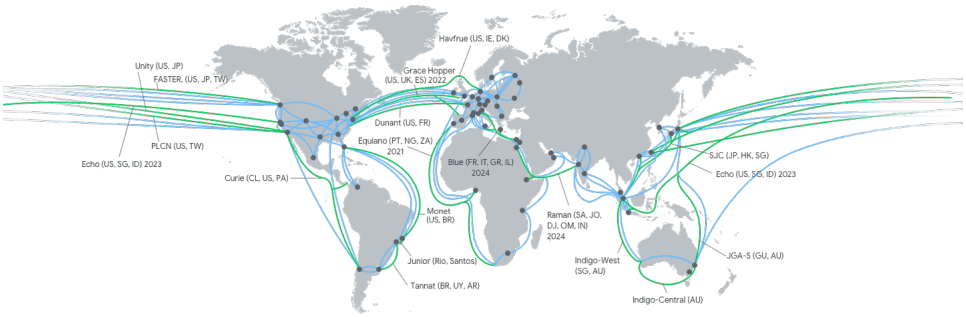

Figure 1.4: Map of Google Cloud network as of 2023.[4]

Different Services Offered by Cloud Computing

The three main models of cloud computing services offered by cloud providers are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Each model represents a different section of the cloud computing stack and is characterized by a specific division of responsibilities

between the user and the service provider, with the Infrastructure as a Service being the one requiring the most management on the user side, and Software as a Service being the other extreme, with most of the management carried out by the cloud provider.

The service model of **Infrastructure as a Service** (IaaS) is the foundation of many cloud technologies. IaaS allows for flexible and cutting-edge hardware (or virtualized) resources that can be scaled to meet the organization's computing and storage needs. This infrastructure can be used to provide the necessary applications, software, and platforms without the hassle of management and maintenance.

A typical example of an IaaS implementation involves the use of virtual machines and storage disks, customized to meet the specific needs of the company, such as the server operating system or storage capacity, offered as a service by a provider.

Platform as a Service (PaaS) is a cloud service model that provides integrated access to hardware and software components through a service provider. PaaS is primarily used for application development. With a PaaS service provider, you can access the cloud infrastructure necessary for application development, including databases, middleware, operating systems, and servers, without having to manage the complexity of configuration and maintenance, or having to manage just part of it. This increases efficiency because you can focus exclusively on developing, executing and managing applications, without worrying about the underlying infrastructure.

Finally, the cloud service model known as **Software as a Service** (SaaS) provides access, via the internet, to a complete software product provided, executed and managed by the service provider. Most SaaS solutions are applications intended for end users.

Using an SaaS service allows one to focus exclusively on using the software, without having to worry about managing the underlying infrastructure at all, as this responsibility falls on the service provider. This means that one does not need to worry about supplying, maintaining and updating the software, for example.

1.2.3 Edge Computing

Edge computing is a distributed computing paradigm that brings computation and data storage closer to the sources of data or the end users. This is expected to improve response times and save bandwidth. Edge computing is an

architecture rather than a specific technology, and a form of location-sensitive distributed computing topology[24].

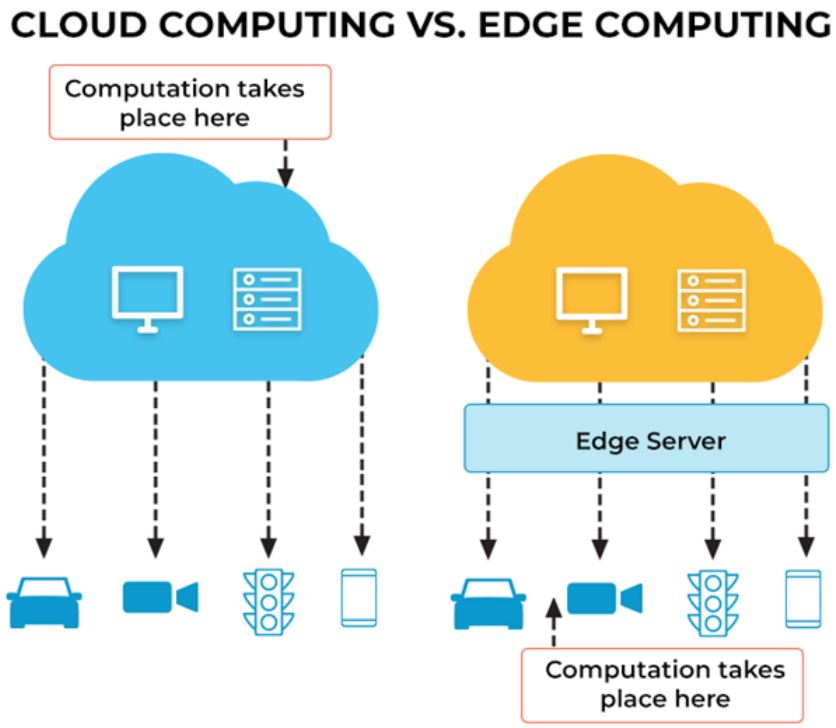

Figure 1.5: Cloud computing vs edge computing.[5]

While cloud embodies core centralized compute and storage resources available via the internet, edge computing, for its nature, is an architecture that decentralize, rather than centralize, the computation.

What is the Edge

In cloud computing, the edge is a colloquial term that refers to the devices of infrastructure away from an organization's core cloud resources. While the edge can be a physical location, the simplest model, the edge can also be demarcated by logical separations. Logical separation may look like a warehouse within a compound of warehouses, where many IoT devices may be deployed.[25]

Edge Cloud Architecture

Edge cloud architecture refers to the combination of core and remote device hardware, and their configurations, that produce a distributed system. In general, edge environments consist of many smaller devices specialized for a

particular task.[25] In the context of this thesis, these devices are both the IoT devices that gather and send cab data, and the intermediate devices that process incoming data before sending it to the "core cloud".

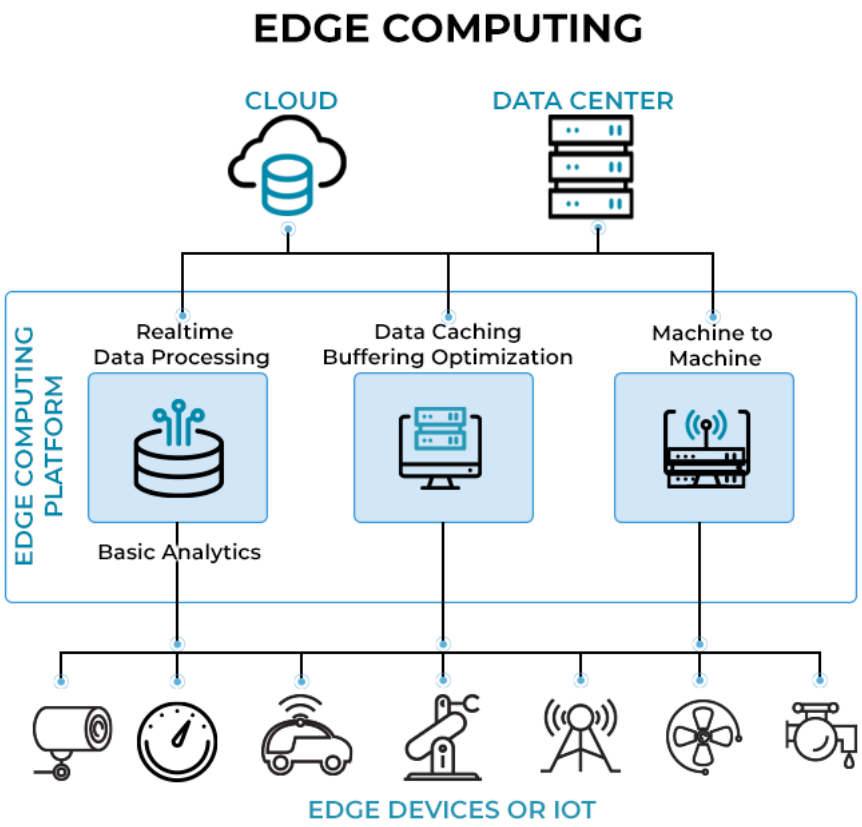

Figure 1.6: Edge computing diagram.[5]

The core cloud on the other hand aims for consistency, homogeneity, and avoids specialization in favor of economies of scale. This is the main reason data is back hauled to core resources, where it is cheaper to analyze huge amounts of data.

When developing an application, cloud architects must evaluate where is the best place to process data. Generally, the right fit means pushing time-sensitive tasks to the edge, and retaining heavy processes in the core.[25]

Edge cloud could be implemented in various, and not mutually exclusive, ways:

- IoT devices can perform telemetry processing on the device itself, instead of sending raw data to the cloud

- regional data centers may replace a single central data center, distributing cloud content in strategic locations

- Micro Modular Data Centers placed in proximity to edge device may provide compute, networking, and storage capabilities, or perform some micro-computations of data received.

Managing Heterogeneity

In edge cloud computing, and IoT, many different devices, onboarded with different hardware and software can cooperate in a larger network with (relative) ease, because the high heterogeneity is masked by the use of the internet protocols (mainly TCP/IP). Indeed, by leveraging the internet, devices just need a way to communicate over the internet, and applications just need to agree on an interface to exchange data, and the communication is then possible; regardless of which hardware is powering the devices, or what software stack the applications are developed upon.

This concept can be pushed even further, in that one does not even need each IoT device to be able to directly communicate using the internet protocols. In fact, it is sufficient for the device to find a chain of devices ending with an internet connected device to be able to participate in the network; for example, one device can only have Bluetooth connectivity, but if it is able to communicate, via Bluetooth, to a gateway device that can connect to the internet, then also such a device can send data through the internet, by leveraging the gateway's connectivity. And it is not even mandatory for the chain to be one hop long as in the example, or using only two different communication protocols; although the longer and more heterogeneous the network is, the more modest the performance will generally be.

1.3 Cloud-Enabling Technologies

Virtual Machines and containers are considered to be cloud-enabling technologies, that is, technologies that greatly simplified and pushed toward the development of cloud environments. They both that improve IT efficiency, application portability, and enhance DevOps, and are used by organization and companies IT departments worldwide on a daily basis.

These two technologies, although similar in the goals and objectives they try to achieve, have distinct characteristics and use cases, and understanding their differences is crucial for developing an agile, cloud-native, and easy to maintain environments and applications.

Virtual Machines (or VMs) provide *hardware-based virtualization*, allowing multiple virtual machines to run their own operating systems and be isolated from each other on the same physical machine. VMs are the key element of Infrastructure as a Service (IaaS) cloud services and can be created on-demand by users.

Containers are *operating system-based virtualization* tools that share a single host OS and the libraries it depends on, drivers, or binaries. They are lightweight and introduce a fraction of the overhead compared to VMs, due to the lack of hardware abstraction. Containers are primarily used for packaging, delivering, and orchestrating software services and applications.

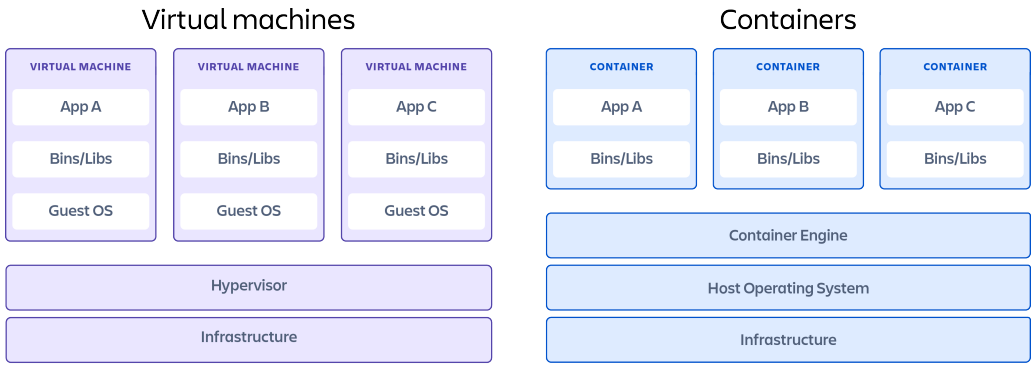

Figure 1.7: Comparison between Virtual Machines and Containers.[6]

Virtual Machine

A Virtual Machine is a software-based abstraction of a computer system that provides a self-contained environment for executing software applications. Put in simple terms, a VM is the virtualization or emulation of a computer system [26]. It is created by partitioning a physical computer into multiple virtual machines, each of which appears to the user as a separate and independent computer system with its own operating system, memory, CPU, and storage resources.

VMs are created and managed using virtualization software that runs on the physical computer hardware, and the virtualization software serves as a layer between the hardware and the virtual machines. The virtualization software provides each VM with a virtual CPU, virtual memory, and virtual storage that are isolated from other VMs and from the host operating system. This allows multiple VMs to run on a single physical computer simultaneously, with each VM having its own set of resources.

A virtual machine can run any operating system that is supported by the virtualization software, regardless of the underlying physical hardware or host operating system. This makes VMs highly portable and flexible, allowing users to easily move VMs between different physical computers or cloud environments.

In summary, a virtual machine is a self-contained software environment that emulates a physical computer system, allowing multiple VMs to run on a single physical computer. VMs are created and managed using virtualization software that provides each VM with its own set of virtual resources, allowing them to run any operating system and be highly portable and flexible.

Containers

Containerization is operating system-level virtualization or application-level virtualization over multiple network resources so that software applications can run in isolated user spaces called containers[27].

Containers are a fully functional and, most importantly portable, cloud or non-cloud computing environment surrounding the application and keeping it independent of other parallelly running environments. Individually, each container simulates a different software application and runs isolated processes by bundling related configuration files, libraries and dependencies.

Since containers are fully functional, portable environments, that are meant to be lightweight, they are not a good fit usually for legacy monolith applications. They are more suitable for running small individual modules of an application. Thus, there is the need to compose multiple containers to create a "complete" application most of the time, and this composition is usually achieved through container orchestration.

Orchestration is the automation of various operational tasks required to run containerized workloads and services. These tasks include provisioning, deployment, scaling, networking, load balancing, and more.

Container orchestration tools help automate the maintenance of containerized applications, enable the replacement of failed containers automatically, and manage the rollout of updates and reconfiguration during their lifecycle.

Containers are widely used in scalable microservices architectures, thanks also to the relative ease with which they can be scaled and also moved around. Microservices, especially if stateless, show a particular affinity to containers, and container orchestration environments, because these technologies allow to move replicate and move microservices across different architectural nodes easily and quickly, and stateless microservices are by nature small, self-contained software modules that are not tied to a particular node, or disk volume, and require little to no coordination between themselves.

Comparison Between VMs and Containers

The main advantage of containers is low performance overhead, whereas VMs offer strong isolation[28]. Containers are more suitable for modern practices and use cases, such as CI/CD in agile, DevOps environments, enhancing portability, and promoting near-limitless scalability. On the other hand, VMs are more suitable for migrating legacy applications to the cloud and hybrid environments, especially for persistent, monolithic enterprise applications with infrequent updates.

It is possible to combine VMs and containers by running containers on top of VMs, which can improve containers' isolation, resource utilization, system management, and functionality. This approach offers benefits but comes at a performance cost[28].

Big cloud provider companies, like Amazon Web Services, Microsoft Azure, and Google Cloud Platform, have already deployed containers on VM instances.

Choosing the right approach depends on the specific application and infrastructure requirements.

1.4 Message Oriented Middlewares

The term *middleware* in Computer Science refers to the set of tools that sit in the middle between higher-level applications and the low-level support (may that be hardware, local Operating System, or else). Middlewares act kind of like a "glue" for software. Usually, middlewares:

- hide heterogeneity
- hide component and resource physical distribution
- provide common interfaces for applications to use.

The term is most commonly used for software that enables communication and management of data in the context of distributed applications.

Middlewares exists of many kinds; in this section we will focus on **Message-Oriented Middleware** (*MOM*), i.e. middlewares supporting sending and receiving messages in distributed environments. A middleware of this kind creates a distributed communications layer that insulates the application developer from the details of the various operating systems and network interfaces.

1.4.1 Characteristics, Benefits, and Disadvantages

Message Oriented Middlewares allow application modules to be distributed over heterogeneous platforms and reduces the complexity of developing applications that span multiple operating systems and network protocols[29].

The main characteristics offered by Message-Oriented Middlewares are:

- asynchronous communication - MOMs provide asynchronous communication, allowing applications to send and receive messages in an asynchronous way, i.e. without waiting for a response
- message queues - many MOM implementations depend on a message queue system, which temporarily stores messages
- routing - MOM can provide routing logic, rely on client applications for routing information, or use a mix of both. Some implementations also use broadcast or multicast distribution paradigms[29].

Given the listed characteristics and benefits that MOMs bring, make them very suitable and widely used in cloud and IoT environments and contexts.

Benefits

The benefits brought to by MOMs are various, notably:

- enabling platform and language agnostic communication, indeed a MOM enables transparent communication between software components developed independently and running on different networked platforms, using various languages
- MOMs simplifies and streamline application development by providing services that enable different applications and services to communicate using common interfaces
- many MOMs can scale and load-balance, more or less seamlessly, to huge amounts of traffic loads varying dynamically across distributed systems
- MOMs can implement security features, thus enabling developers to develop secure applications with ease.

Disadvantages

The main disadvantage of many message-oriented middleware systems is that they require an extra component in the architecture, the message transfer

agent (message broker). Because of this, the increased complexity of the architecture could lead to higher maintenance costs.

Moreover, the asynchronous nature of message-based communication, may not be a good fit for all contexts and applications, especially in cases where applications rely on some kind of intrinsically synchronous communication (such as the sender needing to wait for a reply). However, most MOM systems have facilities to group a request and a response as a single pseudo-synchronous transaction[29].

1.4.2 Apache Kafka and RabbitMQ

Two of the most popular MOMs available today on the market are RabbitMQ and Apache Kafka. They are both popular messaging systems used in distributed computing to handle big data streams, acting as message brokers between applications and services endpoints. However, they have different use cases, capabilities, and trade-offs.

In particular, RabbitMQ uses a bounded data flow, where messages are created and sent by the producer and received by the consumer, and it is best suited for transactional data, such as order formation and placement. On the other hand, Kafka uses an unbounded data flow, with key-value pairs continuously streaming to topics (category or stream name to which records are published by producers, and consumed by consumers).

Both MOMs are open-source and supported by different languages, thus offering various options to developers to choose the language most suited for the particular task they want to perform.

The two messaging systems utilize different messaging models for producers and consumers. Kafka uses a publish-subscribe model, where producers publish messages to topics and consumers subscribe to one or more topics to receive messages. Kafka brokers maintain the current state of each subscribed consumer, allowing them to resume consuming messages from their last known offset in the event of failure or disconnection.

RabbitMQ, on the other hand, uses a message queue model, where producers send messages to a queue and consumers consume messages from the same queue. RabbitMQ brokers use a round-robin approach to distribute messages among consuming clients, and messages are removed from the queue once they have been successfully consumed by a client.

In general, Kafka is better suited for high-throughput, distributed systems that require real-time data streaming and processing, while RabbitMQ is more

suitable for asynchronous, reliable message delivery where message ordering and consumer acknowledgement are important.

While RabbitMQ is a distributed message broker suitable for complex routing scenarios, utilizing what is known as a smart broker/dumb consumer model, Kafka is a distributed event streaming platform designed for raw throughput and direct stream processing using a dumb broker/smart consumer model.

RabbitMQ Use Cases

The scenarios in which RabbitMQ performs the best are:

- *transactional data processing*, such as order formation and placements
- *low-latency* message delivery in complex routing scenarios
- *legacy protocols supporting* applications
- when *acknowledgement-based* message retention is needed.

Apache Kafka Use Cases

The use case and scenarios in which Kafka usually performs the best are:

- *high-throughput scenarios*, such as big data environments where millions of messages per second need to be processed
- *operational data processing*, such as auditing and logging statistics
- *streams* with "at least once" partitioned ordering.

1.5 Cluster Computing

> *"Customers invented clusters, as soon as they couldn't fit all their work on one computer, or needed a backup."*
>
> – Gregory F. Pfister, *In Search of Clusters*[30]

1.5.1 Brief Overview of Clustering

Clustering is the process of grouping together multiple computing nodes to work together, as a single logical unit, in a distributed system. It is a technique used to improve the performance, scalability, and reliability of distributed systems.

Clustering was born out of the need to address the challenges posed by the growing complexity of distributed systems. As distributed systems became larger and more complex, the traditional approach of relying on a single node to handle all the tasks became unsustainable. Clustering provided a solution by allowing multiple servers to work *together* to handle the workload.

A cluster's nodes are usually interconnected through fast LANs (Local Area Networks), and each node runs its own instance of an operating system[31]. The connection between nodes needs to be fast in order to minimize the orchestration and coordination overheads.

Cluster computing is used to achieve different goals, including:

- HPC (High Performance Computing)
- Load-balancing
- HA (High-Availability).

1.5.2 Benefits of Clustering

There are several benefits of clustering. One of the primary benefits is improved *performance*. By distributing the workload across multiple servers, the overall processing capacity of the system is increased, allowing it to handle more requests and data more quickly. Clustering also provides scalability, allowing the system to easily add or remove servers as needed to accommodate changes in demand.

Another benefit of clustering is improved *reliability*. In a distributed system, a single server failure can have a significant impact on the availability and performance of the system. By using clustering, the workload is distributed across multiple servers, reducing the impact of a single server failure. Clustering also allows for automatic failover, where if one server fails, another server in the cluster can take over its tasks without interrupting service.

There are different types of clustering in distributed systems, including load-balancing clusters, failover clusters, and high-performance clusters. Load-balancing clusters distribute the workload evenly across all servers in the cluster to improve performance, while failover clusters provide redundancy and automatic failover to improve reliability. High-performance clusters are designed for high-performance computing tasks, such as scientific simulations or big data analytics.

1.5.3 Drawbacks of Cluster Computing

While clustering provides many benefits, that more often than not greatly outweigh the cons, it is important to list also the drawbacks that cluster computing introduces.

The main drawbacks introduced by cluster computing are:

- cost is high, in fact while having multiple computers working on a task can provide better performances, load-balancing, and failover capabilities, it also increases the cost, with respect to having just one server doing all the work
- many computing units and nodes working together makes monitoring harder, as well as failure analysis, and management
- also, the resulting infrastructure is more complex than having one server only, thus more infrastructure management work is needed with cluster computing.

1.5.4 Notable Examples of Frameworks Leveraging Clustering

Apache Spark and Apache Kafka are two popular distributed frameworks that make use of clustering techniques to achieve their performance and scalability goals.

Apache Spark, that will be also called Spark from now on, is a distributed computing system that is designed for processing large-scale data. It uses a cluster of computers to split up data processing tasks into smaller sub-tasks, which can then be processed in parallel across the cluster. Spark uses a combination of load-balancing and failover clustering techniques to distribute workloads across its cluster of nodes. The load-balancing technique ensures that each node receives an equal amount of work, while the failover technique ensures that if a node fails, another node can take over its tasks without interrupting the computation.

Apache Kafka, that will be also referred to as just Kafka from now on, on the other hand, is a distributed streaming platform designed for processing and storing high-throughput, real-time data streams. It uses a cluster of servers to store and distribute data streams across the system. Thanks to clustering, Kafka can ensure high availability and fault-tolerance.

Both Kafka and Spark will be discussed extensively in the next chapter of this thesis.

1.6 Data Sampling

In data analysis, sampling is the practice of analyzing a subset of all data in order to uncover the meaningful information in the larger data set[32]. Data sampling is necessary in all that scenarios in which there are huge amounts of data, and the necessity to have them processed within strict time constraints.

For example, in an urban scenario, to give up-to-date information about traffic, one may need to process data coming continuously from hundred of thousand or even millions of vehicles, but since traffic patterns change quickly, the information is only useful if processed within tight time constraints, like a minute or so. In a scenario like the one depicted, taking a sample of the data may be a way to achieve the time constraints' goal, at the expense of some accuracy, that, in this scenario, does not represent a great constraint. In fact, when looking for traffic updates, one is not interested in the exact average speed of a given street, i.e. 22.541 Km/h, just knowing that the average speed is between 20 and 24 Km/h is enough to choose which road to travel.

Moreover, using different sampling techniques that may or may not help in retaining the original distribution of values can have an impact on both the performances and the accuracy of a system.

1.6.1 Basic Statistics Definitions

Definition 1. *(Probability distribution). In probability theory and statistics, a probability distribution is the mathematical function that gives the probabilities of occurrence of different possible outcomes for an experiment. It is a mathematical description of a random phenomenon in terms of its sample space and the probabilities of events (subsets of the sample space).[33]*

Definition 2. *(Statistical population). In statistics, a population is a set of similar items or events which is of interest for some question or experiment. A statistical population can be a group of existing objects (e.g. the set of all stars within the Milky Way galaxy) or a hypothetical and potentially infinite group of objects conceived as a generalization from experience (e.g. the set of all possible hands in a game of poker).[34]*

Definition 3. *(Sampling). In statistics, sampling is the selection of a subset (a statistical sample) of individuals from within a statistical population to estimate characteristics of the whole population.[35]*

1.6.2 Sampling Strategies

The simplest way to take a sample from a population is to randomly sample from the population. That is, if, in a population of two million items, one wants to retain 50% of the items, one can randomly choose one million items from the population. In statistics, this is called Simple Random Sampling. In Definition 4, Simple Random Sampling is defined in a more rigorous way.

Definition 4. *(Simple Random Sampling). In statistics, a Simple Random Sample (or SRS) is a subset of individuals (a sample) chosen from a larger set (a population) in which the individuals are chosen randomly, all with the same probability. It is a process of selecting a sample randomly. In SRS, each subset of k individuals has the same probability of being chosen for the sample as any other subset of k individuals. A Simple Random Sample is an unbiased sampling technique. Simple Random Sampling is a basic type of sampling and can be a component of other, more complex, sampling methods.[36]*

Simple Random Sampling is a good choice when one wants a fast sampling technique and does not have any information on the population.

On the other hand, if one want to exploit some knowledge on the population to sample with a better accuracy, one can use Stratified Sampling. In Definition 5, Stratified Sampling is defined in a more rigorous way.

One of the advantages of Stratified Sampling is that it can help to reduce the sampling error, as we mitigate the risk of sampling a non-representative sample (e.g. not sampling enough items from a given subpopulation). A situation in which stratified sampling may be advantageous verifies when it is desirable to have estimates of the population parameters for groups within the population, as Stratified Sampling ensure that enough items from each of the strata are taken. This is especially true if the strata greatly vary in size, as SRS may not sample enough items from the smaller strata.

Definition 5. *(Stratified Sampling). In statistics, stratified sampling is a method of sampling from a population which can be partitioned into subpartitions. Stratification is the process of dividing members of the population into homogeneous subgroups before sampling.[37] Each subpartitions is called a stratum.*

Stratified sampling could be a useful technique to carry out approximate queries in big data scenarios without losing too much accuracy, but nonetheless gaining significant speed, in all those cases in which it is possible to exploit some knowledge on the information carried by the data, and it is possible to find meaningful strata.

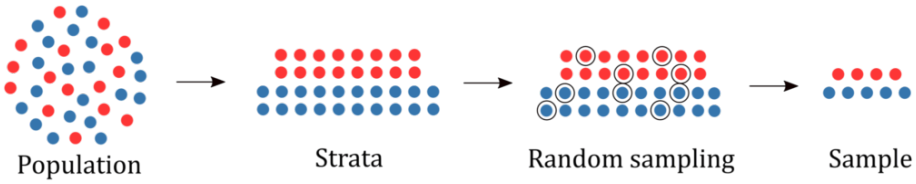

Figure 1.8: Stratified Sampling illustrated.[7]

1.7 Programming Languages

Each programming language has its own unique set of strengths and weaknesses, that make it more suitable in some scenarios than others.

Programming languages could be roughly categorized into compiled languages and interpreted languages. The code written in compiled languages is directly converted into machine code that can be run directly, while code written in interpreted languages needs an interpreter that at runtime interpret the source code (or an intermediate format) to machine instructions.

As a general rule of thumb, compiled languages provides generally better execution speeds compared to interpreted languages, while interpreted languages provides generally better development speed, and are therefore more suitable for rapid prototyping, or when CPU-intensive tasks are delegated "under-the-hood" to more optimized libraries, and the program just glues the different operations together.

I would not delve deeper into this topic as it is outside the scope of the thesis, nonetheless this brief introduction was necessary to justify the language choices that were made developing the different programs this thesis is composed of.

1.7.1 Python

Python is an Object-Oriented, garbage-collected, dynamically typed, interpreted programming language. It integrates some aspects of Functional Programming, such as the lambda expressions, while remaining fundamentally Object-Oriented.
It appeared for the first time in 1991[38], and has since then gained in popularity, becoming one of the most used programming languages as of 2023.

It is one of the most used languages in the fields of Machine Learning and Data Analysis, and is also a fairly common choice for backend and API development.

Overall, Python is a very versatile language that is often used for rapid prototyping, thanks to its relatively simple syntax, memory management, and

the rich, vast, and vital ecosystem of libraries it provides for pretty much every use case.

1.7.2 Rust

Rust is a multi-paradigm, compiled, general-purpose programming language that first appeared in 2015, and has since gained a lot of popularity because of its innovative way to achieve memory-safety and thread-safety, as well as for its advanced features inherited from functional programming.

Its innovative way to achieve memory safety without the need of a garbage collector, and being a compiled language, are two of the reasons that make Rust a generally fast language, with performance comparable to C++.

Graydon Hoare created Rust as a personal project while working at Mozilla Research in 2007. Mozilla officially sponsored the project in 2009. Since the first stable release in May 2015, ever more companies have adopted Rust and implemented it in their development pipelines.

Most notably, in December 2022, it became the second high-level language to be supported in the development of the Linux kernel, the first being C.

Despite being relatively young, Rust already has a vast variety of data analysis libraries (called crates in the Rust world) to choose from.

Some of the most beloved features Rust offers, apart from the compile-time memory and thread safety achieved with the borrowing system and the borrow-checker, are:

- the `Option` monad, which is an effective way to abstract over the side effect of computations that may return a null value
- the `Result` monad, which is an effective way to abstract over the side effect of computations that may fail, and may return an error. This particular abstraction, together with some syntactic sugar offered by the language, provides a powerful tool to manage error, that, in my opinion, provides a solid way to manage possible run-time errors
- the support for proper pattern-matching is another feature of Rust that gives developers more control over a program's control flow.

Relevant to the thesis, Rust offers a good collection of geospatial-focused crates, provided by GeoRust[39], that offers useful set abstractions, types definitions, functions to treat geospatial data, and performs operations such as

encoding or decoding geohashes, reading and writing GeoJSON files, and more.

Moreover, the Rust language comes with the powerful package management tool named cargo, that is, in my opinion, one of the killer features of this language.

With cargo, building binaries, testing both the code and the documentation, and managing the dependencies is relatively easy.

dependencies with cargo are managed through the use of two files:

- *Cargo.toml* which is a file written in TOML language, which is basically an advanced version of an INI file, in which dependencies are described in a "broad sense", and it is thought to be developer-managed and human-readable
- *Cargo.lock* which is the file containing the actual information about dependencies, and it is thought to be managed by cargo.

Rust's documentation suggest that end product repositories committing both *Cargo.toml* and *Cargo.lock* files to their VCS (Version Control System), and committing only *Cargo.toml* for non-end product repositories, such as a rust library that other rust packages will depend on[40]. One of the few pain points I encountered using cargo, is the slow fetch time from *crates.io*, the Rust community's crate (i.e. libraries) registry, but this is one of the points the Rust community is working on to improve. This is especially a problem when building Docker images that need to compile Rust binaries, unless you are able to cache the result of the fetch somehow. But, apart for the slightly longer build time the first time the build is made, this is often not a problem.

Chapter 2

Used Technologies

This chapter will provide to the reader an extensive discussion of the technologies utilized and the motivations behind the choices that were made.

In particular, relevant technologies that will be extensively discussed includes Apache Kafka, Apache Spark, and Docker.

2.1 From Virtual Machines to Containers

The thesis project started with the use of Virtual Machines, using Virtualbox in particular, but later was made a switch to containers, in particular to Docker containers.

The switch to a container-based approach was made for a variety of reasons:

- containers can reduce the overhead on the host machine with respect to the use of several VMs, if used correctly
- less management of the software installation and of the networking is required
- containers provide a solution that is more microservices and cloud oriented; in fact they can often be seamlessly (or with little configuration) run on local machine, on the most popular cloud providers, or on private cloud solutions.

2.2 Docker

The emergence of Docker as a platform for containerization is closely linked to the widespread use of virtualization in recent years. Since the 2010s, containers have exploded in popularity as a technology, and Docker, since its birth in 2013, have imposed as the de facto standard for containerization.

2.2.1 Docker Inc.

The open-source project Docker was started in 2013 from the company *dotCloud, Inc.* (now *Docker, Inc.*), specialized in Platform-as-a-Service. Docker is written in Go language.

Go is a programming language that is greatly optimized for concurrency and with a modern syntax (although inspired by C) developed by Google, and widely used in cloud contexts.

Quickly since its launch, Docker has generated a global interest, due to its ability to streamline the development process and make it easier to move applications between different computing environments, leading the company to enter into an important agreement with Red Hat in September 2013 and collaborate with Microsoft in October 2014. Meanwhile, Google launched the open-source container-orchestration tool Kubernetes, the open-source successor of Borg, which drew the world's attention to Docker.

However, there have been some frictions between Docker and Red Hat over the years, particularly related to the development and adoption of container technologies.

A key turning point was when Docker, in 2017, decided to make the source code of some fundamental components of the technology public, donating it to the Cloud Native Computing Foundation (CNCF). The CNCF is a Linux Foundation project that was launched in 2015 to help advance cloud technologies and tech industries to stay on par with the development of these technologies.

The move guaranteed compatibility between Docker and third party solutions, and further pushed Docker as becoming the industry leader in the containers field.

2.2.2 Docker Engine

The core of Docker is its *Docker Engine*, an architecture that interact with the Linux kernel and allows receiving commands through a client, either by command-line interface or API. The Docker Engine is based on a client-server architecture:

- *server* - the layer on which the individual containers are executed. It has the duty of managing the network layer, the shared resources layer and the images, and exposes its services through a REST API

- *client* - it is a command line interface (CLI) allowing the user to send commands to the server component. Client and server communicate through a REST

- *registry* - a repository from which it is possible to pull or push images. The repository can be both public or private. The public repository contains images that everyone can use, the private repositories contain images accessible only to the organization that created it. The public repository is the Docker Hub[8].

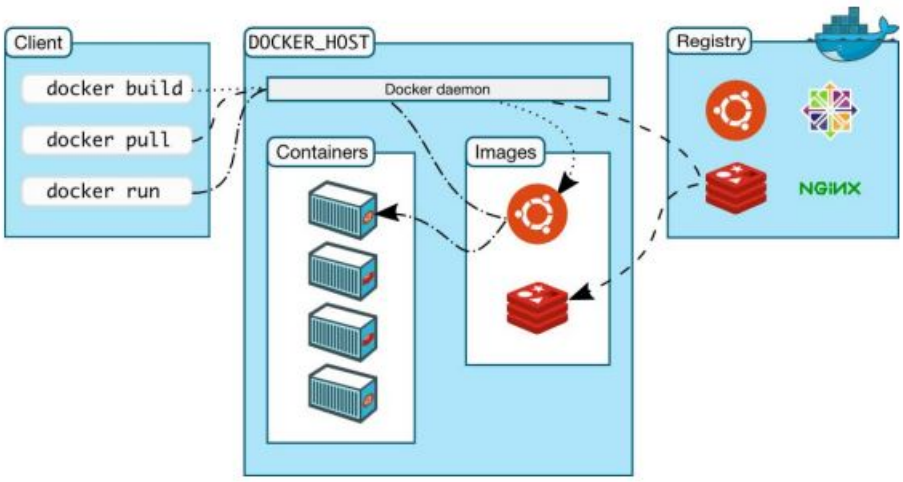

Figure 2.1: The architecture of Docker.[8]

2.2.3 Networking in Docker

Docker containers can communicate with each other and with the outside world, using networking capabilities provided by Docker.

By default, each Docker container gets its own isolated network namespace, which means that it has its own network interfaces, IP address, and routing table.

Docker supports several networking modes, which determine how containers are connected to the network. Some of the most common modes include:

- **bridge networking**. This is the default networking mode in Docker. Containers are connected to a virtual bridge network, which allows them to communicate with each other and with the host system

- **host networking**. In this mode, containers use the host system's network stack, which means they have direct access to the host's network interfaces and IP address. This can be useful for high-performance networking applications, but it can also create security risks

- *overlay networking*. This mode allows containers to communicate with each other across multiple hosts in a distributed environment. It uses a virtual overlay network that spans multiple physical networks and provides seamless communication between containers.

In addition to these networking modes, Docker also provides several network drivers, which allow containers to connect to different types of networks, such as virtual private networks (VPNs), software-defined networks (SDNs), and more.

Overall, Docker networking provides a flexible and powerful way for containers to communicate with each other and with the outside world.

The default networking mode, bridge networking, is enough for most applications, and not by chance it is the default one.

2.2.4 Docker Compose

Docker Compose is an auxiliary component of Docker that allows to define and execute applications spread among more Docker containers. Docker Compose simplifies the process of managing multi-container applications by making use of a declarative way to declare services provided by containers and dependencies (such as if a container needs to start after another) between them. Developers should define a set of containers, their dependencies, and how they should be configured and run together in a YAML file called a Compose file, and usually named *docker-compose.yaml*.

One of the main benefits of using Docker Compose is that it allows developers to define their entire application stack in a single, declarative, configuration file. This makes it easy to start and stop the entire application with a single commands, as well as to manage and scale individual containers of the stack. Additionally, the Compose file is version controlled, providing a way for developers to easily manage changes to the application composition over time.

Another advantage of Docker Compose is that it simplifies the process of managing dependencies between containers. Compose makes it easy to define dependencies between services, and to ensure that those dependencies are started and stopped in the correct order.

However, especially for small applications, Docker Compose may be an overkill, and may only add maintenance overhead, because of the need to maintain the compose file too.

Docker Compose YAML

The YAML used to create an application spread among multiple containers with Docker Compose is usually called *docker-compose.yaml*. This file uses a declarative approach to describe the application.

The language chosen for it, YAML, is a human-readable data serialization language that is used in many contexts, e.g. Kubernetes, with the same goal of describing in a declarative way the goals one want to achieve, rather than how to achieve them.

The main keys of the Docker Compose YAML are:

- The `version` of the Compose file, which specifies the version of the Compose file format being used.
- A list of `services`, which define the containers that make up the application stack. Each service has a name, a set of configuration options, and can be dependent on other services.
- A list of `networks`, which define the networks that are available to the containers in the application stack. Each network can have its own configuration options, such as a custom IP range.
- A list of `volumes`, which define the volumes that are available to the containers in the application stack. Each volume can have its own configuration options, such as the location on the host file system where the volume is stored.
- A set of `environment` variables, which can be used to set environment variables for the containers in the application stack.
- A set of `secrets`, which can be used to store sensitive data, such as passwords, outside of the Compose file.
- A set of `configs`, which can be used to store configuration data, such as a configuration file for a web server, outside of the Compose file.
- A set of `deploy` options, which are used to configure the deployment of the application stack to a swarm cluster.

2.2.5 Dockerfile

The *Dockerfile* is a fundamental element of the Docker ecosystem, because in it are described the necessary steps to create a Docker image.

The Dockerfile is a fundamental element of Docker's ecosystem, because it describes the steps needed to create a Docker image. The image is then used to create a Docker container.

It is important to note that all containers created from the same image are identical, and this makes it easy to have multiple replicas of the same application. This is one of containers' superpowers, but it is also the reason why containers are best suited for stateless applications (not that managing stateful applications is impossible, it is just a bit trickier).

The Dockerfile specifies two main things for an image:

- the base image from which it derives
- the customizations to apply to the base image.

Practically speaking, the Dockerfile is a text file containing a series of instructions, that are executed in sequence to create an image. The Dockerfile defines the steps needed to create a custom Docker image by specifying the base image, installing dependencies, setting environment variables, and copying files to the container.

Dockerfile Structure

The Dockerfile is a simple text file containing instructions for Docker on how to build the image. The main commands are:

- `FROM`: specifies the base image for the Dockerfile. This command must be the first command in the Dockerfile, and it determines the environment in which subsequent commands are executed.
- `RUN`: executes a command in the container. This command is used to install dependencies, set up the environment, and perform other operations needed to build the image.
- `ADD`: copies a file or directory from the host to the container. This command is similar to the `COPY` command, but it has additional functionality, such as the ability to extract compressed files and the ability to fetch remote files.
- `COPY`: copies a file or directory from the host to the container. This command is used to add files and directories to the container, such as application code or configuration files.

- `CMD`: specifies the default command that should be executed when the container starts. This command is optional, but it's a best practice to include it to ensure that the container has a default behavior.

- `WORKDIR`: sets the working directory for the subsequent commands in the Dockerfile. This command is used to ensure that subsequent commands are executed in the correct directory.

2.2.6 Creation of a Docker Image

Docker images are built from a series of layers, where each layer represents a change made to the image. When a Docker image is built, Docker takes each instruction in the Dockerfile and creates a new layer based on the previous layer. Each layer is read-only and represents a specific state of the image.

Layers are an important concept in Docker because they enable Docker to cache and reuse existing layers when building new images.

Using layers effectively is crucial to build lightweigth and efficient Docker images.

To create a Docker image, one need to run the following command of the Docker command line interface:

```
docker build <path/to/dockerfile>
```

Optionally, one could specify a tag to give to the image with the option `-t <tagname>`. The option is very useful in that, by omitting it, the resulting Docker image will have a hard to remember name. Thus, the use of this option is usually advised.

2.2.7 Useful Commands

In this subsection, a list of useful Docker CLI commands is provided. This list is not meant to be comprehensive, nor all the available options of each command is provided. It is just an overview of the most useful commands to manage the lifetime of containers instances.

In Docker, after creating an image, it is as well possible to run it with the following command:

```
docker run <image_name>
```

This command will run a container instance using the specified image.

After running a container, it is possible to execute a series of actions inside it, like opening a shell, and execute commands inside the container.

Listing the currently running containers, it is possible with the following command:

```
docker ps
```

Stopping a container is another operation that one would want to execute often, especially during the development stage, and is achievable with the following command:

```
docker stop <container_name>
```

Finally, removing a container is possible with the command:

```
docker rm <container_name>
```

When, instead of an application made of a single container, you have an application that is spread among multiple containers, using the `docker-compose` command makes the management of these containers as a single unit easier.

To run an application that is made of multiple containers, go inside the directory in which the `docker-compose` YAML is placed and run the following command:

```
docker-compose up
```

It is also often useful to run the above command with the `-d` (or `--detached`), to run the containers in background, and be able to continue using the current shell.

Stopping a Docker compose application is possible running the following command:

```
docker-compose down
```

2.3 Choice of the Message-Oriented Middleware

The choice of the Message-Oriented Middleware to use fell on Apache Kafka. Kafka was chosen instead of other options, such as RabbitMQ, mainly because

of the high-throughput it can support. Indeed, Kafka can support higher throughput than RabbitMQ, reaching also millions of messages per seconds. Moreover, Kafka is a fairly popular choice in IoT scenarios in which modest amounts of data are frequently sent for monitoring purposes by many IoT devices, which is exactly the scenario of this thesis focuses on.

Anyway, Kafka's higher performances comes at the cost of a simpler broker architecture, and thus a little more work to be done by the developer on the business logic side.

2.4 Apache Kafka

Apache Kafka is an open-source system for message exchange developed by the Apache Software Foundation and first released in 2011. Apache Kafka is written in Scala and Java. The project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds[41].

Kafka was originally developed at LinkedIn, and was subsequently open sourced in early 2011. The problem Kafka was originally set out to solve was low-latency ingestion of large amounts of event data from the LinkedIn website and infrastructure into a lambda architecture that harnessed Hadoop and real-time event processing systems.

Apache Kafka is a distributed messaging system, a message-oriented middleware as stated in the previous chapter, that acts as an event streaming platform, with a strong focus on high scalability. Events, in the context of Kafka, are anything (action, change, incident) that is recorded by software. For message exchange, Kafka's servers and clients uses a binary TCP-based protocol that is optimized for efficiency and relies on a "message set" abstraction that naturally groups messages together to reduce the overhead of the network roundtrip.

The interaction model used by Kafka is publish-subscribe, where producers publish data to "channels" (called topics), and consumers subscribe to those channels to receive the data. Kafka can deliver in-order, persistent, scalable messaging. It has publishers, topics, and subscribers. It can also partition topics and enable massively parallel consumption. All messages written to Kafka are persisted and replicated to peer brokers for fault tolerance, and those messages stay around for a configurable period of time[42].

The two main concepts on which Kafka is based are that of a distributed commit log, and that of key-value pairs. Scaling in Kafka is achieved by splitting logs into partitions (e.g. key-based splitting). A file system, or database commit log is designed to provide a persistent record of all transactions, so that by replaying them it is possible to consistently (re)build the state of a system.

Similarly, data within Kafka is stored in a durable fashion, maintaining the message order, and can be read deterministically.

The way Kafka achieves massive scalability and fault tolerance is via a distributed architecture that allows it to be run on multiple nodes. with data replication and partitioning across nodes.

Adding to the high throughput, fault-tolerant, and scalability, there are the stream processing APIs provided by Kafka among the reasons of its success. **Stream processing** APIs enable developers to build complex data processing *pipelines* with ease.

Finally, one of the key strengths of Apache Kafka is the ability to support both *multiple producers and multiple consumers*, writing and reading the topics concurrently, in a seamless way.

2.4.1 Kafka Clients, Servers, and Brokers

Kafka architecture relies on the client/server model, where clients act as normal clients, and servers are divided into brokers and servers running Kafka Connect to continuously import and export data as event streams to integrate Kafka with existing systems such as relational databases as well as other Kafka clusters[9].

Kafka brokers are responsible for maintaining the storage layer, the layer to which Kafka's client consumer and producer applications can respectively read and write data.

Kafka Client

Kafka clients are users of the messaging system, and they divide into producers and consumers (of the messages). *Producers* create new messages to

topics. In general, a message is produced to a specific topic.

By default, the producer does not care what partition a message is written to, and messages are load-balanced over all available partitions evenly. However, there are situations in which writing a message to a specific partition is required for semantics reasons, thus it anyhow is possible to choose what partition to send the message to.

Consumers read messages from topics. The consumer subscribes to one or more topics and read the messages in the order in which they were produced. The consumer keeps track of which messages it has already read and consumed by keeping track of the offset of the messages. The offset is a metadata of the

message (an integer value that continually increases, like a counter) that Kafka adds to each message as it is produced.

Consumers can be part of a *consumer group*, which is one or more consumers that work together to consume a topic. The group assures that each partition is only consumed by one member[10].

2.4.2 Event Streaming

Event streaming is the practice of capturing data in real-time from event sources like sensors, mobile devices, databases, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed. Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.[41]

Event streaming is the digital equivalent of the human body's central nervous system, to use a biological metaphor[9].

Event streaming finds application in a wide variety of use cases in the industry:

- to process payments and financial transactions in real-time, such as in stock exchanges, banks, and insurances
- to track and monitor vehicles, fleets, and shipments in real-time, such as in logistics and the automotive industry
- to continuously capture and analyze sensor data from IoT devices or other equipment, such as in this thesis' scenario
- to collect and immediately react to customer interactions and orders, such as in retail industry, the travel industry, and mobile applications
- to monitor patients in hospital care and to ensure timely treatment in emergencies
- to connect, store, and make available data produced by different divisions of a company, which is an increasingly common situation
- to serve as the foundation for data platforms, event-driven architectures, and microservices.

2.4.3 Topics

Kafka topics are the way Kafka chose to organize events. A topic can be imagined as a table in a relational database, or a log of events, or a channel.

Messages are written to topics in an append-only fashion, and consumed by consumers from beginning to end. Topics are generally divided in partitions, and for this reason there is no guarantee of strict time-ordering across the entire topic, just within each individual partition[10].

In Kafka, topics are append-only and:

- can be configured to expire data after some time
- under the hood, they are files stored on disk
- can only be read by seeking an arbitrary offset, then scanning sequential log entries.

There are several advantages of this approach:

- it is easy to understand, because the log-like semantics is already familiar to developers
- it is relatively easy to manage, and the simple semantics is enabling for high throughput.

Each message in a topic is uniquely identified by an *offset*, allowing consumers to track their position in the stream.

2.4.4 Partitions

If topics were constrained to live only on a single machine, that would pose a strong limit on Kafka scalability. Partitioning is then the mechanism provided by Kafka to split a topic. Each partition is an ordered, immutable, sequence of messages that is assigned to a specific broker in the cluster.

Partitioning enables for parallel message processing, making it possible to scale the number of consumers that can read from a topic. Moreover, partitioning provides fault-tolerance by allowing message replication across different brokers.

The term stream is often used when discussing data within Kafka. Most often, a stream is considered to be a single Kafka topic of data, regardless of the number of partitions it has. This represents a single stream of data moving from the producers to the consumers.

A key feature of Kafka is the retention of messages, which is the durable storage of messages for some period of time. Kafka brokers are configured with a default retention, but custom configurations for retention can be applied.

Messages can be retained for either a specified time frame (e.g. 7 days) or until the topic reaches a certain size in bytes (e.g. 2 GB). Once these limits are reached, messages are expired and deleted. One thing that is not settable using Kafka's retention policies, is to expire a message once all interested subscribers have consumed it, as it is possible in other message-oriented middlewares (such as RabbitMQ). This is due to the Kafka's choice of using a dumb broker, smart consumer model. Also, these policies are simpler than expiring a message once all consumers have consumed it, and they enable the broker to ensure higher throughput, due to their simplicity.

2.4.5 Kafka APIs

In addition to command line tooling ,Kafka offers five core APIs:

- the Admin API to manage and inspect brokers, topics, and Kafka objects in general
- the Producer API to publish a stream of events to one or more topics the Consumer API used to subscribe and read from one or more topics, and to process the stream of events produced
- the Kakfa Streams API to implement stream processing applications. It offers higher-level functions to process streams when compared to the Consumer and Producer APIs. These higher-level functions include functions to perform transformations, stateful operations (such as aggregations and joins), windowing, event-time based processing, and more. Input is read from one or more topics in order to generate output to one or more topics
- the Kafka Connect API to build and run reusable data import and export connectors that consume or produce streams. Connectors connect, for example, to RDBMS like PostgreSQL, and others.

2.4.6 Producers and Consumers

Kafka clients are users of the system, and there are two basic types: producers and consumers. In other MOMs, these may be called publishers and subscribers, respectively.

When a producer publishes a message to a topic, it can specify a key for the massage. That key is then used to determine the partition the message

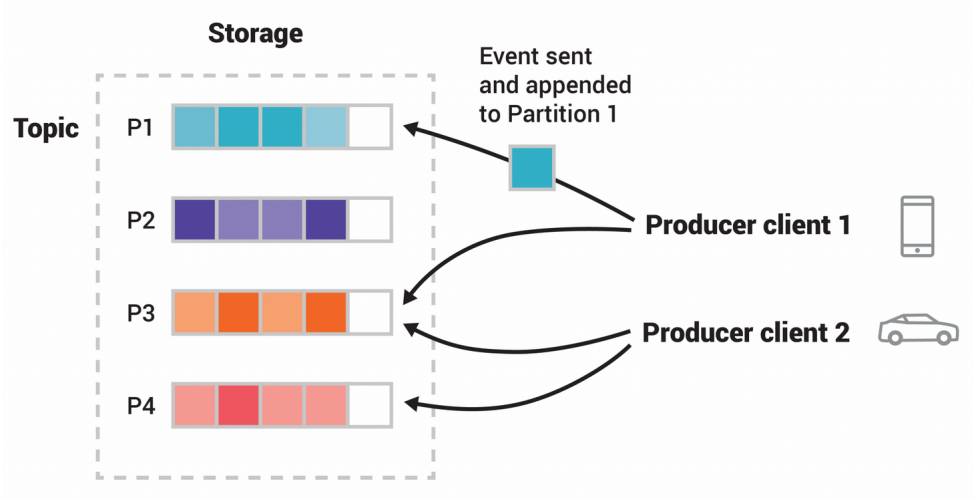

Figure 2.2: Figure showing a partitioned topic to which multiple client producers can send data. It is important to notice that many clients can seamlessly write not only to the same topic, but to the same partition too.[9].

will be assigned to. By default, the partition is chosen based on a hash of the key, ensuring messages with the same key to be assigned always to the same partition.

Consumers can read messages from one or more partitions of the same topic. Each partition can be read by only one consumer at a time (to prevent a message duplication), but a single consumer can read from multiple partition. This is known as the *single-consumer-per-partition* model. This model is an important Kafka concept, enabling for efficient, parallel, processing of messages while avoiding duplicates or conflicts between consumers.

Consumers work as part of a consumer group, which is one or more consumers that works together to consume a topic. The group assures that each partition is only consumed by one member. The mapping of a consumer to a partition is oftentimes referred to as ownership of the partition by the consumer[10].

With this concept of a consumer group, consumers can horizontally scale to consume topics receiving large numbers of messages. Moreover, it provides a way to achieve fault-tolerance. In fact, if a consumer fails, the remaining members of the group will rebalance the partitions being consumed to take over for the failed member. What happens, is that one of the remaining members will start consuming the partition(s) consumed by the failed node.

This mechanism does not only work for fault-tolerance, indeed it works also for scaling as, if a new node joins the group, the partitions will be rebalanced also counting the new member.

2.4.7 Cluster

Kafka brokers are designed to operate in a cluster. Within a cluster of Kafka brokers, one broker will also function as the cluster controller (automatically elected from the members of the cluster). The controller is responsible for administrating the operations, such as assigning partitions to brokers and monitoring the other members for broker failures.

A partition of a topic is owned by one single broker, called the leader of the partition. A partition may also be replicated across multiple brokers, but one and only one remain the leader for that partition. This mechanism provides redundancy of messages, such that another broker can take over the leadership if there is a failure in the leader without message losses[10].

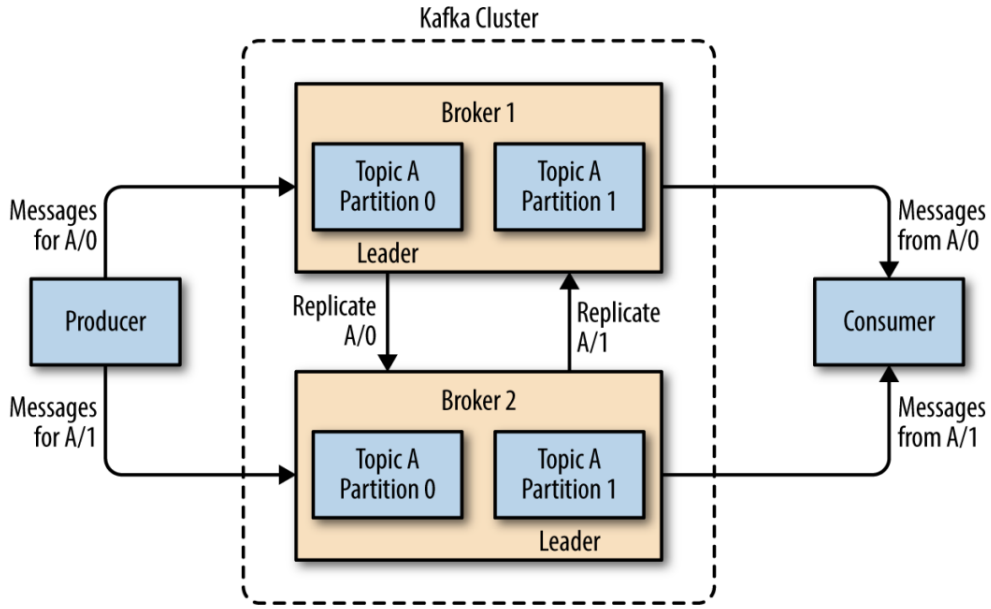

Figure 2.3: Figure showing a Kafka cluster composed of two brokers, with a topic having two partitions and replicated among the two brokers. It is important to note that, for each partition of the topic, only one of the two brokers will be the partition leader.[10].

2.5 Choosing a Big Data Analysis Framework

Apache Spark is a distributed computing system designed for processing large volumes of data quickly and efficiently. It is built on top of Apache Hadoop MapReduce and extends the MapReduce model to support more types of computations, including interactive queries and stream processing.

Apache Spark and Apache Hadoop are two of the main alternatives when choosing a framework to perform big data analytics.

Apache Spark is often considered to be better than Hadoop MapReduce for several reasons. First, it is significantly faster due to its in-memory data processing capability, and also it can handle a variety of workloads including batch processing, real-time processing, machine learning, and graph processing, and is easier to use and more flexible than MapReduce.

Additionally, Spark provides built-in fault tolerance using Resilient Distributed Datasets (RDDs), whereas MapReduce requires more complex code to handle fault tolerance. The Spark ecosystem includes a wide range of libraries and tools for data processing, making it a popular choice for big data processing over Hadoop MapReduce.

Because of this motivations, we quickly decided to focus on Spark rather than Hadoop.

2.6 Spark

Apache Spark is an open-source distributed computing system designed for processing large volumes of data quickly and efficiently, initially released in 2014 and developed by the University of California Berkeley's AMPLab, and later donated to the Apache Software Foundation, which has maintained it since[43].

Spark provides an interface for programming clusters with implicit data parallelism and fault tolerance.

The main components of Apache Spark include Spark Core, Spark SQL, Spark Streaming, Spark MLlib, and Spark GraphX.

At a high level, every Spark application consists of a driver program that runs the user's main function and executes various parallel operations on a cluster. The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel[44].

2.6.1 Spark Core

Apache Spark Core is the base engine for large-scale parallel and distributed data processing. It is responsible for scheduling and dispatching tasks, coordinating input and output (I/O) operations, and providing an underlying execution engine for Spark. Spark Core introduces the concept of Resilient

Distributed Datasets (RDDs), which are immutable fault-tolerant distributed collections of objects that can be operated on in parallel.

RDDs can be created by loading an external dataset or distributing a collection from the driver program. They support two types of operations: transformations and actions. Transformations are operations (such as map, filter, join, union, etc.) that are performed on an RDD and yield a new RDD containing the result. Actions are operations (such as reduce, count, first, etc.) that return a value after running a computation on an RDD.

Transformations in Spark are "lazy", meaning they don't compute their results right away. Instead, they remember the operation to be performed and the dataset to which the operation is to be performed. The transformations are only actually computed when an action is called, and the result is returned to the driver program. This design enables Spark to run more efficiently.

By default, each transformed RDD may be recomputed each time an action on it is run. However, an RDD, may also persist in memory using the persist or cache method, in which case Apache Spark will keep the elements around on the cluster for much faster access the next time it is queried[45].

2.6.2 Spark SQL

Spark SQL is a module in Apache Spark used for structured data processing. It allows developers to query structured data using either SQL or the DataFrame API[44].

One use of Spark SQL is to execute SQL queries in Apache Spark. When running SQL queries from within another programming language (such as Scala or Python), the results will be returned as a `Dataset` or `DataFrame`. It is as well possible to interact with the SQL interface using the command-line or over JDBC/ODBC.

Spark SQL provides the ability to generate logical and physical plans for a given query using the EXPLAIN statement. The cost-based optimizer, columnar storage, and code generation included in Spark SQL make queries fast, while the Spark engine provides full mid-query fault tolerance, allowing it to scale to thousands of nodes and multi-hour queries, without the risk of having to restart from scratch in case of faults, which can never be underestimated, especially when running on big clusters.

2.6.3 Spark Resilient Distributed Dataset

In Apache Spark, the primary user-facing API is the *Resilient Distributed Dataset* (RDD), which is an immutable, distributed, collection of elements that

can be operated on in parallel[44]. RDDs are fault-tolerant and automatically recover from node failures.

RDDs can be created in two ways: by parallelizing an existing collection in the driver program, or by referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes. RDDs are divided into logical partitions, which may be computed on different nodes of the cluster.

Internally, RDDs are characterized by five main properties: a list of partitions, a function for computing each split, a list of dependencies on other RDDs, optionally a partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned), and optionally a list of preferred locations to compute each split on. All the scheduling and execution in Spark is done based on these methods, allowing each RDD to implement its own way of computing itself. Users can also implement custom RDDs (e.g. for reading data from a new storage system)[44].

There are two types of shared variables in Spark: broadcast variables and accumulators. *Broadcast variables* can be used to cache a value in memory on all nodes, while *accumulators* are variables that are only "added" to, such as counters and sums. By default, when Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task. Sometimes, a variable needs to be shared across tasks or between tasks and the driver program[44].

Spark RDD supports a wide range of transformations that can be applied to RDDs, the main ones being:

- map
- filter
- flatMap
- groupByKey
- reduceByKey
- sortByKey
- join
- distinct.

Spark RDD also supports iterative and interactive operations. Iterative operations store intermediate results in a distributed memory instead of persistent storage (disk) and make the system faster. If the distributed memory (RAM) is not sufficient to store intermediate results (state of the job), then Apache Spark will store those results on the disk, an operation that degrades performances. Interactive operations too keep data in memory for better performances.

By default, each transformed RDD may be recomputed each time an action is run on it. However, it is also possible to persist an RDD in memory, in which case Spark will keep the elements around on the cluster for much faster access the next time it is queried. There is also support for persisting RDDs on disk or replicated across multiple nodes, to help horizontal scaling.

Shuffling

Shuffling in Apache Spark is the process of redistributing data across different executors and even across machines. This mechanism is triggered by operations like *groupByKey()*, *reduceByKey()*, *join()*, and *groupBy()*. Shuffling is an expensive operation, as it involves disk I/O, data serialization, and network I/O[44], thus it is good practice to try to limit the amount of shuffling in Spark applications.

Also, shuffle generates a large number of intermediate files on disk, and these files are preserved until the corresponding RDDs are no longer used and are garbage collected (long-running Spark jobs may consume a large amount of disk space because of this).

To optimize shuffle performance, developers can adjust various configuration parameters. For example, one can change the default shuffle partition value using the *conf* method of the *SparkSession* object or using Spark Submit command configurations. Getting the right size of the shuffle partition is never an easy task, and usually takes many runs with different values to achieve the optimized number.

2.6.4 Spark Structured Streaming

Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.[11] It enables to use SQL queries on real-time data streams. The Spark SQL engine will take care of running the computation incrementally and continuously, and updating the final result as streaming data continues to arrive[11]. The incoming data are continuously processed in batches, using a micro-batch processing engine.

The input stream can be though of as an unbounded "Input Table". Every data item that is arriving on the stream is like a new row being appended to it.

A query on the input will generate the "Result Table". Every trigger interval (say, every 1 second), new rows get appended to the Input Table, which eventually updates the Result Table[11].

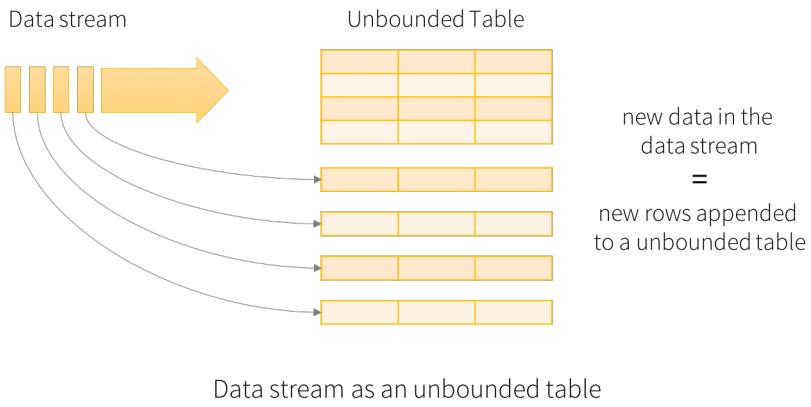

Figure 2.4: The input stream is treated as if it was an unbounded table[11].

The output stream of a structured stream is defined as what gets written out to the external storage (or console). The output can be defined in different modes:

- *complete*, that is the entire updated Result Table will be written at every update
- *append*, that is only the new rows appended in the Result Table since the last trigger will be written in output
- *update*, i.e. only the rows that were updated in the Result Table since the last trigger will be written to the external storage. Note that the rows that were updated do not always coincide to the rows that were appended, thus the distinction between the two modes.

2.6.5 Spark Web UI

The Spark Web UI is a suite of user interfaces that provide information about the status of a Spark application and resource consumption of a Spark cluster. The UIs include Jobs, Stages, Tasks, Storage, Environment, Executors, and SQL. The UI also displays scheduling delay and processing time for each micro-batch in the data stream if the application uses Spark streaming[44].

To access the Spark Web UI, the Spark application must be running. If the application is running locally, the Spark UI can be accessed at the URL *http://localhost:4040/* by using a common browser. The Spark UI runs on port 4040 by default.

The Spark metrics are available in JSON format as well. The JSON is available for both running applications and in the history server. The endpoints are mounted at */api/v1*. The API gives developers an easy way to create custom monitoring and visualization tools for Apache Spark.

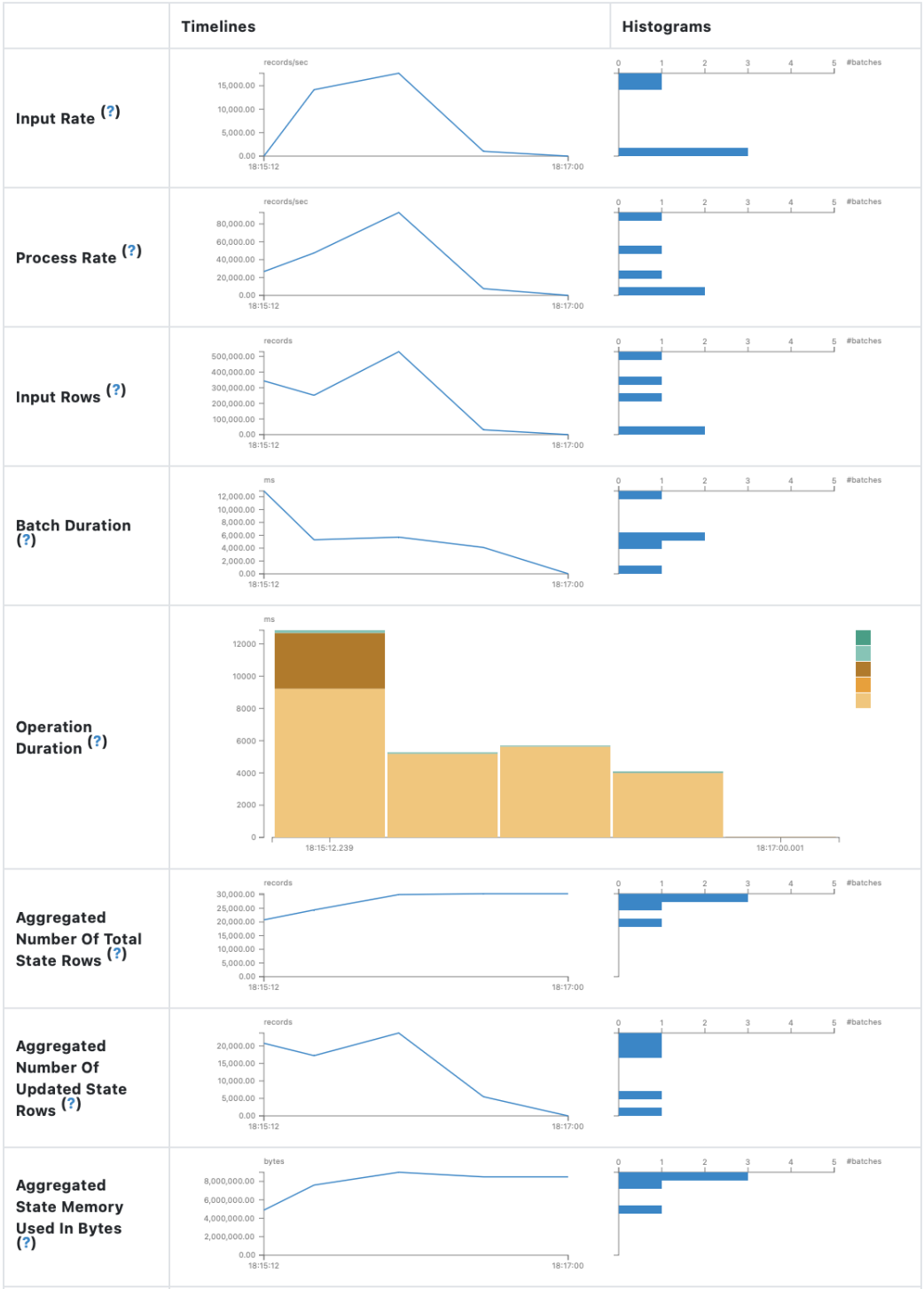

Figure 2.5: Example of the information available in the Spark Web UI Structured Streaming tab.

Chapter 3

Architecture and Features of the Proposed Solution

To allow the reader to better understand the choices that were made, it is important to start by briefly describing the scenario in which we posed ourselves to carry out this thesis' work.

The scenario was that of moving taxis in a city, as already stated a few times, and the collection and pre-analysis and filtering of this data carried out by intermediate edge nodes, before sending them to the cloud where further analysis would be carried on.

Our architecture is based on the introduction of edge nodes, acting as gateways, or a "shield", between the incoming data and the cloud. Incoming data on each node were:

- Geohash encoded, and the hash then used to calculate (approximately) the neighborhood of origin of each message
- optionally, sampled in batches of messages arrived within the same time window
- sent to a Kafka topic that may optionally be "spatial-aware", in the sense that it contains only messages originated from the same neighborhood. Otherwise, sent to a random partition of a non-spatial-aware topic. This was behavior was controlled via a settable configuration.

These are, at a high-level, the responsibilities the edge nodes implemented for this thesis were capable to perform.

The cloud nodes running Apache Spark, on the other hand, continually processed incoming data, by running a query calculating the average speed for

each geohash for each 30-minute time window. Data were then updated and exported to CSV at regular intervals.

As we did not have at our disposal a fleet of actually moving taxis, we needed a way to simulate this scenario. To simulate this, we needed a program able to distribute data to our edge nodes and, since we were already using Kafka for sending data from the edge nodes to the cloud, we chose to leverage yet another Kafka topic specifically for this.

To give an overview of our final setup, our final setup is composed of different programs, running on different containers, and each with a specific purpose. Specifically, it was designed and implemented:

- A program to simulate data arrival in the edge nodes, thus a program that act as a distributor of data to the edge nodes
- A program, running on the edge nodes, that is able to read, preprocess, and send out again incoming data
- A program, running (ideally) in nodes in the cloud, reading preprocessed data and further analyzing them to extract meaningful information about the traffic situation at a given time of day
- A program whose purpose is to visualize in an interactive way the analysis' results, with a map showing visually the traffic situation for a given time window, giving the possibility to resize and move the map, as well as to select another time window.

Figure 3.1 shows what are the three main logical steps into which it was divided the thesis' work, and how information flows among the different programs.

As shown in *step 1*, the program that distributes data to the edge nodes reads data from a file (specifically, it would be a CSV file), and writes the data to a pipe of some kind.

In *step 2*, two edge programs run independently by reading messages to the pipe, and writing them to another pipe. These are the binaries that would ideally run on the edge nodes.

The third step, in which data analysis and visualization are carried out, is divided into two parts. In the first part, *step 3A*, the data are read from the pipe to which step 2 binaries wrote, and are analyzed. The output of the analysis is then written to a file. Finally, in *step 3B* data are visualized in a web application that reads from the file generated at the previous step, and shows the results in an interactive map.

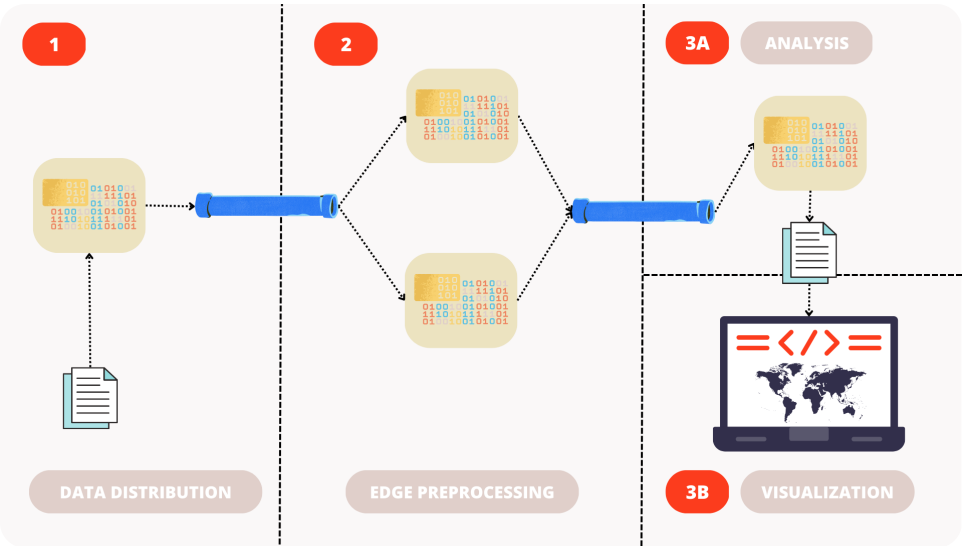

Figure 3.1: Figure showing the various programs that will compose the setup and the information flow between them.

The project was made using different technologies, that were combined to create the final result.
In particular, the project started as a group of several Ubuntu Virtual Machines, and later was switched to a more modern container-based approach. The project in its final form consists of different parts, that implements:

- a **Kafka cluster** with specific topics where nodes produce to and consume data from; specifically, there are two Kafka containers
- a **Zookeeper** container needed by Kafka nodes
- a **Spark cluster** processing incoming data composed of a master and a worker node.

In this chapter, the implementation and choices made for the setup will be discussed extensively.

Figure 3.2 shows an overview of the containers' deployment structure, using dashed lines to indicate that communication between two containers is direct. It is possible to note that the two Kafka nodes do not communicate directly with the Spark master, but Kafka nodes writes on a pipe, and the Spark master reads from it. In particular, the "pipe" are one or more Kafka topics.

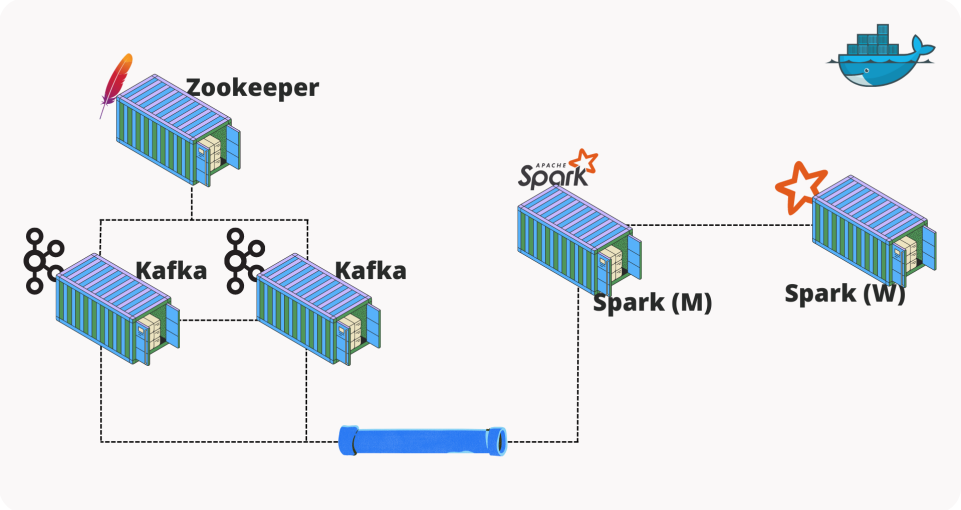

Figure 3.2: Overview of the containers' deployment structure. In the figure, five containers are depicted. Spark(M) indicates that the Spark node act as master, Spark(W) that the node act as a Spark worker.

3.1 Containers Setup

The base images for each of the containers previously described were pulled from the Docker Hub. Specifically, the images pulled were the following:

- *bitnami/zookeeper*[46] image from Bitnami for the Zookeeper node
- *bitnami/kafka*[47] image for the Kafka nodes
- *bitnami/spark*[48] image for the Spark nodes.

These Docker images provided by Bitnami represents a very good starting point to start working with Apache Spark, Apache Kafka, and Apache Zookeeper, and comes with the possibility of customizing the main settings of each of these software programs by default using some environments variables.

Moreover, the Spark and Kafka images were further specialized, via custom Dockerfile, to run the binaries needed for the project.

3.1.1 Custom Image for The Kafka Nodes

To provide some context, the Kafka nodes needed to run two binaries:

- the first, running only on one of the two nodes, that read from the CSV file simulating the data distribution

- the second, running on both nodes that read from the specified partition of the Kafka topic in which data arrives, that preprocess data and sends them to the output Kafka topics.

It was possible to have a completely separate container running the first binary, and then have two additional containers running only the second binary, but this was considered an overkill, because the computational overhead of the data distribution binary is minimal. Thus, it was made the choice to create two images, one with the first and second binary (plus the duty of creating all the needed Kafka topics), and another for the node running only the second binary.

Ideally, if the setup would be deployed in a real production context, the second image only would be needed, and the first image (the one that has also the binary for running the data distribution) would not be deployed.

The Docker image for the Kafka node running both binaries described above is the following:

```
FROM docker.io/bitnami/kafka:3.4

USER root

WORKDIR /

RUN mkdir kafka-producer-rs/
RUN mkdir kafka-edge-bin/

RUN apt-get update
RUN apt-get install -y \
    build-essential \
    libssl-dev \
    pkg-config \
    vim \
    git \
    curl

## Update new packages
RUN apt-get update

# Get Rust
RUN curl https://sh.rustup.rs -sSf | bash -s -- -y
```

```
24
25 RUN echo 'source /.cargo/env' >> /.bashrc
26 ENV PATH="${PATH}:/.cargo/bin"
27
28 COPY data/china /data/china
29 COPY kafka-producer-rs /kafka-producer-rs
30 COPY kafka-edge-bin /kafka-edge-bin
31
32 # Build producer
33 WORKDIR /kafka-producer-rs
34 RUN cargo build --release
35 RUN mv ./target/release/kafka-producer-rs /kafka-producer-rs
36 # Copy config file
37 COPY ./kafka-rs-configs/producer_config.toml
      /kafka-producer-rs/producer_config.toml
38
39 # Build edge
40 WORKDIR /kafka-edge-bin
41 RUN cargo build --release
42 RUN mv ./target/release/kafka-edge-rs /kafka-edge-rs
43 # Copy config file
44 COPY ./kafka-rs-configs/kafka_edge_config-kafka.toml
      /kafka_edge_config.toml
45
46 # Copy startup script
47 WORKDIR /
48 COPY ./docker-startup-scripts/startup-kafka.sh /startup.sh
49 RUN chmod +x /startup.sh
```

Listing 3.1: Dockerfile for the Docker image of the Kafka node running both the data distribution and edge binary.

This Dockerfile changes the user to `root`, which is not a best practice security-wise, but it is very helpful in development environments.

Moreover, multi-stage builds could have been used to keep the final image size down, and this could be a possible improvement for the future. But, as the ultimate goal of this thesis was not about the use of Docker, it was made a choice to concentrate the efforts on other aspects.

This same considerations are valid for the other images that will be exposed in this thesis.

The second Kafka image, the one used by the node (but potentially more

than one) running the edge binary, is created from the following Dockerfile:

```
FROM docker.io/bitnami/kafka:3.4

USER root

WORKDIR /

RUN mkdir kafka-edge-bin/

RUN apt-get update
RUN apt-get install -y \
    build-essential \
    libssl-dev \
    pkg-config \
    vim \
    git \
    curl

## Update new packages
RUN apt-get update

# Get Rust
RUN curl https://sh.rustup.rs -sSf | bash -s -- -y

RUN echo 'source /.cargo/env' >> /.bashrc
ENV PATH="${PATH}:/.cargo/bin"

COPY data/china /data/china
COPY kafka-edge-bin /kafka-edge-bin

# Build edge
WORKDIR /kafka-edge-bin
RUN cargo build --release
RUN mv ./target/release/kafka-edge-rs /kafka-edge-rs
# Copy config file
COPY ./kafka-rs-configs/kafka_edge_config-edge.toml
   /kafka_edge_config.toml

# Copy startup script
WORKDIR /
```

```
39 COPY ./docker-startup-scripts/startup-edge.sh /startup.sh
40 RUN chmod +x /startup.sh
```

Listing 3.2: Dockerfile for the Docker image of the Kafka node running the edge binary only.

Startup Scripts

The startup script for the Kafka node running both binaries is:

```
1 #!/bin/bash
2 ./kafka-producer-rs/kafka-producer-rs create_topic
3 ./kafka-edge-rs topic create out --for-nbw-strat
4
5 ./kafka-producer-rs/kafka-producer-rs &
6 ./kafka-edge-rs -s 0.8
```

Listing 3.3: Startup script of the Kafka container having both data distribution and edge binary.

The startup script for the Kafka node running only the edge binary is:

```
1 #!/bin/bash
2 sleep 10
3
4 ./kafka-edge-rs -s 0.8
```

Listing 3.4: Startup script for the Kafka container with the edge binary only.

Both the startup scripts runs the binaries with options that will be explained in the following sections of this chapter. Anyway, it is important to highlight that these scripts are not automatically executed on container start, so to leave the possibility to customize some options while testing without the need to rebuild the image, and to leave the possibility to the user to decide when to start the execution.

3.1.2 Custom Image Used by the Spark Master

The Spark master also needed a custom image, because its duty is both running the spark script, and the web server for data visualization. Thus, a custom image was created using the following Dockerfile:

```
1 FROM docker.io/bitnami/spark:3.3
2
```

```
USER root

WORKDIR /

RUN apt-get update
RUN apt-get install -y \
    build-essential \
    libssl-dev \
    pkg-config \
    git \
    vim \
    curl

## Update new packages
RUN apt-get update

COPY data/china /data/china
COPY cloud_analytics /cloud_analytics

# Setup flask web app
COPY flask_data_viz /flask_data_viz
WORKDIR /flask_data_viz
RUN pip3 install -r requirements.txt

WORKDIR /
# Setup startup script
COPY ./docker-startup-scripts/startup-spark.sh /startup.sh
RUN chmod +x /startup.sh
```

Listing 3.5: Dockerfile used to build the Spark master node image.

Startup Script

The following is the startup script for the Spark node:

```
#!/bin/bash
sleep 15

# execute flask app in separate terminal
bash -c "cd /flask_data_viz && python3 main.py" &

# Set startup script
```

```
spark-submit --packages
   org.apache.spark:spark-sql-kafka-0-10_2.12:3.3.0
   cloud_analytics/main.py &
```

Listing 3.6: Startup script on the Spark master container.

As for the other startup scripts, this one too does not start automatically, but need to be started manually.

3.1.3 Composing the Containers

Previously in this section, the individual containers were thoroughly described, both in their implementation and in the purpose they serve.

But these containers by themselves are not able to achieve much, they need to be composed together in order for the whole setup to work successfully.

The composition of these containers is achieved by leveraging Docker Compose, and in particular, the following is the *docker-compose.yaml* file that was used:

```yaml
version: '3'

services:

  # ----------------- #
  # Apache Spark    #
  # ----------------- #
  spark:
    build:
      context: .
      dockerfile: Dockerfile_spark
    user: root
    environment:
      - SPARK_MODE=master
    ports:
      - '8080:8080'
      - '4040:4040'
      - '7077:7077'
      - '5000:5000'
    volumes:
      - ./vol:/vol
  spark-worker:
    image: docker.io/bitnami/spark:3.3
```

```
    user: root
    environment:
      - SPARK_MODE=worker
      - SPARK_MASTER_URL=spark://spark:7077
      - SPARK_WORKER_MEMORY=4G
      - SPARK_EXECUTOR_MEMORY=4G
      - SPARK_WORKER_CORES=4
    volumes:
      - ./vol:/vol

  # ----------------- #
  # Apache Kafka    #
  # ----------------- #
  zookeeper:
    image: docker.io/bitnami/zookeeper:3.8
    ports:
      - "2181:2181"
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes
  kafka:
    build:
      context: .
      dockerfile: Dockerfile_kafka
    user: root
    environment:
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - KAFKA_CFG_DELETE_TOPIC_ENABLE=true
      - ALLOW_PLAINTEXT_LISTENER=yes
    depends_on:
      - zookeeper
  edge-1:
    build:
      context: .
      dockerfile: Dockerfile_edge
    user: root
    environment:
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - KAFKA_CFG_DELETE_TOPIC_ENABLE=true
      - ALLOW_PLAINTEXT_LISTENER=yes
    depends_on:
      - zookeeper
```

```
65      - kafka
```

Listing 3.7: Docker Compose YAML composing the various containers of the setup together. The base structure for this file was taken from the article *"A Fast Look at Spark Structured Streaming + Kafka"*[49].

An important thing to notice is that the Spark master container (the one named `spark`) and the Spark worker container (named `spark-worker`) share memory by mounting the same volume (`./vol`). This allows the master to have the worker's results, written to CSV, always available in its local file system.

3.2 Data Distribution

The first step of the setup, was to simulate data arrival at the cloud's edge. The input dataset used in the setup was available as a CSV file. But, while the dataset was one, the nodes that these data needed to be distributed were not necessarily one, therefore there was the need to distribute the single dataset to the numerous nodes. To achieve this, two options were considered:

- a first idea was to divide the dataset in N part with some strategy, and then just mount on a container a volume containing a part of the dataset
- to leverage Kafka, and use a partitioned topic to distribute the data. Each node would then read data from just some of the topic's partition.

The first idea is for sure easier to implement, but present several limitations:

- the solution is rigid, the slightest change in the distribution logic needs the partitioned files to be recreated
- the solution requires manual intervention if the distribution strategy needs to be changed (an operator should create the new files that each node need).

This option was therefore quickly discarded, in favor of the second one, that required more work to get started, but provides much more flexibility.

The language chosen to implement such a program was Rust. Apart from the implementation language, a choice was made into how to organize the program that proved to be very handy once the switch to containers was made. This choice was of using a TOML configuration file together with command line options to customize the program's behavior across runs.

This choice made it possible to just change a couple lines in the configuration file and then rerun the container's startup script, without the need to remember

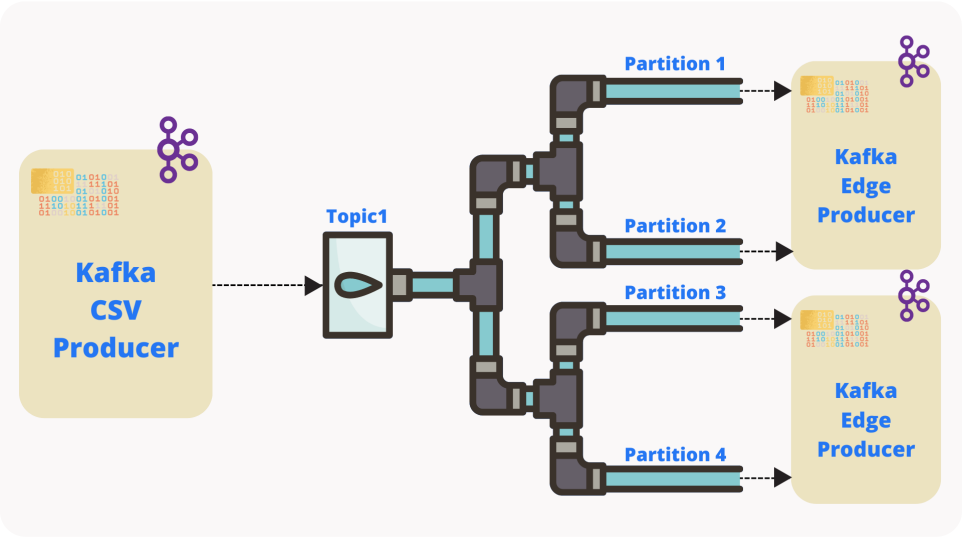

Figure 3.3: This figure shows how, conceptually, data are distributed to the edge nodes. The data distribution program, named "Kafka CSV Producer" in the figure, sends data to a partitioned Kafka topic. Then, the programs running on the edge nodes, in the figure called "Kafka Edge Producer" (producer because it produces the actual messages consumed by the data analysis node) consumes messages from the assigned partition of the topic, elaborates them, and then sends them to other topics, not shown in this Figure.

the CLI options to pass to the program, especially for the parameters that were evaluated as "unlikely" to change often between different runs. Anyway, the possibility to override some of these parameters with a command line option was left in case of necessity, to speedup configuration tuning.

Also, the program was also developed with the ease of use in mind, and thus subcommands to empower the user to easily setup/reset the environment for the program's execution (such as creating or deleting the target topic, or create or swap the configuration file) were provided.

3.2.1 TOML Configuration File Structure

The following is an example of what a configuration file for the data distribution program looks like:

```
[kafka]
zookeeper = [ "zookeeper:2181" ]
brokers = [ "kafka:9092" ]
topic = "datain"
```

```
partitions = 2

[data]
source = "/data/china/mobility/guang.csv"
msg_sleep_in_ms = 1
chunk_size = 1000
chunk_sleep_in_ms = 500
```

Listing 3.8: Example configuration TOML for the binary simulating data distribution.

In this configuration file, the sections and fields shown has the following meaning:

- the `kafka` section, is the section in which the options strictly related to the Kafka behavior are placed
  - the `zookeeper` field is an array of strings containing the information about the zookeeper instances used by Kafka to work properly
  - the `brokers` field is an array of strings of all the Kafka brokers to which to connect to
  - the `topic` field is a simple string containing the name of the Kafka where to write the messages to be distributed among edge nodes
  - the `partitions` field is an integer (greater than zero) representing the number of partitions the topic should have
- the `data` section is the section in which information about how to distribute data is stored
  - the `source` field is a string that must contain the absolute path to the data source CSV file
  - the `msg_sleep_in_ms` field is an integer that initially was used to specify a particular sleep in between sending each individual message, expressed in milliseconds, but in later stages of the development was considered a suitable, and valuable, option to manage traffic load, as the minimum granularity for this option is 1 millisecond, and that was not enough in most testing scenarios (indeed, using the `chunk_size` option together with the `chunk_sleep_in_ms` one provide a much better mean for controlling traffic load). The option was nonetheless maintained, and is available for possible use in the future

  - the `chunk_size` field is an integer value that determines how many messages are sent in a single iteration of the loop sending messages (provided enough messages are left in the file)
  - the `chunk_sleep_in_ms` field is an integer value that determines how often a chunk of messages is sent on the topic.

This configuration file, in order to be effective at run-time, need to be placed in the same directory in which the program's binary is. A couple of utility subcommands for creating a new configuration file using a default template, and to swap the configuration file with a new one were provided, to allow the user to quickly change the configuration without the need to know or remember where exactly the binary is placed.

3.2.2 Messages Structure

The CSV files have the following structure:

- have an ID, which is a string that identifies the taxi driver sending that particular message
- have a latitude and a longitude, that are two floating point numbers
- have a timestamp, which is easier (in this step) to treat as a string to avoid useless conversions during serialization and deserialization, as we are not interested in this step at the actual timestamp (this duty will be carried out by the Spark cloud data analysis script)
- have a speed, which is a non-negative floating point number that is the detected speed by the IoT sensors on the vehicles at the given timestamp.

To deal with messages with this structure, the following Rust structure was defined for this purpose:

```
#[derive(Debug, Deserialize, Serialize, Clone, PartialEq)]
/// Struct holding the message Kafka message that will be sent.
pub struct Message {
    pub id: String,
    pub lat: f64,
    pub lon: f64,
    pub time: String,
    pub speed: f64,
}
```

Listing 3.9: Message structure used by the data distribution program.

It is important to notice that this structure derives several traits (conceptually similar, if not equivalent, to interfaces for people more familiar with Object-Oriented programming) that are helpful in managing instances of this struct.

In particular, the `PartialEq` trait allows the overload of the equality (and inequality) operator. One may wonder why the equality is only "partial" and not complete. The equality can be derived as only "partial" because the "complete" version, called `Eq`, requires the equality to be reflexive, symmetric and transitive, and this is not the case for floating point numbers (because the equality between two NaN numbers returns false). Thus, the `Eq` trait cannot be derived, i.e. automatically generated, for structures whose fields contain at least a field of a type that does not implement `Eq`. Anyway, the `PartialEq` trait is sufficient in our use case.

Two other important derived traits for the message structure are `Serialize` and `Deserialize`, that provides implementations for structures that should support being serialized and deserialized. These traits are provided by the *serde* crate, that allows to easily managing serialization and deserialization capabilities.

The deserialization, in particular, is used when reading an entry from the CSV file, to deserialize the read record into the data structure.

Moreover, the structure implements a method that is used to send messages to Kafka in JSON formatting, which is a format that is a widely popular choice to send data over the network in recent years. In particular, the method uses the `json` macro from the `serde_json` crate to achieve this goal:

```
impl Message {
    /// Serialize the `Message` in JSON format.
    pub fn json_serialize(&self) -> Value {
        json!({
            "id": self.id,
            "lat": self.lat,
            "lon": self.lon,
            "time": self.time,
            "speed": self.speed,
        })
    }
}
```

Listing 3.10: Implementation of the JSON serialization method.

3.2.3 How to Use the Binary

The binary comes with several subcommands that are intended to help to manage the configuration and the interaction of the program with Kafka.

In particular, a production-grade powerful argument parser is obtained using the derive feature capabilities offered by the *clap* (which stands for Command Line Argument Parser) crate. The available subcommands are:

- `edit_config` - a subcommand that allows to easily manage the configuration TOML file. The command has two other subcommands
  - `create` - used to create a new configuration file in the same directory in which the executable is placed from a template
  - `replace` - takes a file path as input, and it is used to replace an existing (or non-existing) configuration with the provided file.
- `create_topic` - a subcommand used for creating the topic (the topic is created as indicated in the config TOML file)
  - the command accepts a command line option (`--replication-factor`) that can be used to specify the desired replication factor of the topic to be created (with a replication factor of one being the default if the option is not provided)
- `delete_topic` - allows deleting the topic specified in the configuration file. It is useful because deleting messages from a Kafka topic is not a supported operation, and when running multiple tests it is often useful to delete a topic to restart from an empty one.

Finally, a useful helper is generated for each command and subcommand, producing the output shown in 3.4. The helper is also automatically generated by *clap*.

3.2.4 The Kafka Producer

The producer implemented using the *kafka-rust* crate[50], which is a crate that implements a Kafka client using only Rust.

The following code is used to initialize the producer:

```
let mut producer = Producer::from_hosts(config.kafka.brokers)
    .with_ack_timeout(Duration::from_secs(1))
    with_required_acks(RequiredAcks::One)
    .create()?;
```

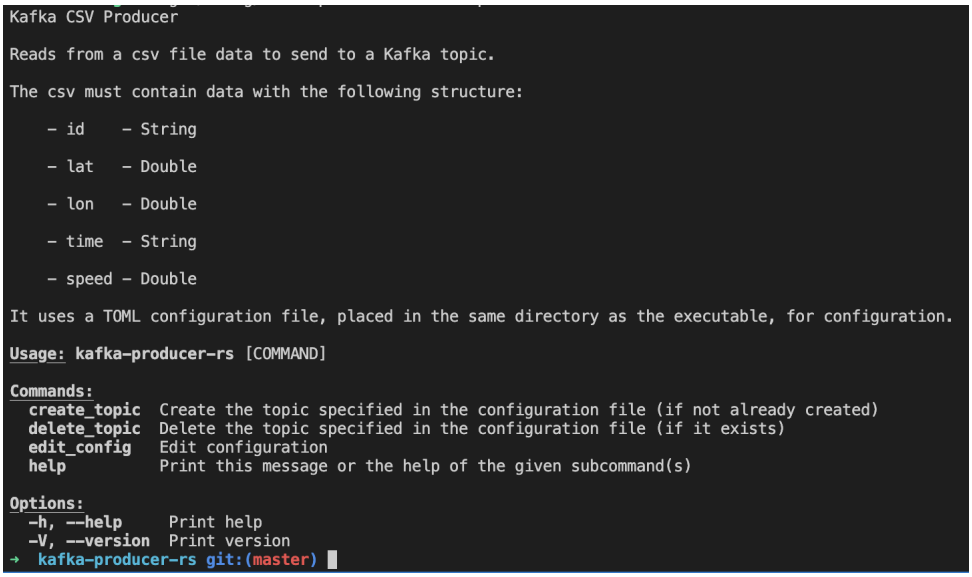

Figure 3.4: Example output of the Kafka data distribution program's helper.

Listing 3.11: Creation of a Kafka producer using *rust-kafka*.

The created producer uses the acknowledgement one, meaning that the producer will consider the write successful when the leader broker receives the record. Also, the acknowledgement would be awaited for a maximum of one second.

3.2.5 Reading From CSV

The CSV reader is implemented using the *csv* crate[51], which is a very good crate to handle with ease CSV files of even considerable size.

The following code shows how the reader is created:

```
let file = File::open(config.data.source)?;
let mut reader =
    ReaderBuilder::new().has_headers(true).from_reader(file);
```

Listing 3.12: Creation of a CSV reader using the *csv* crate.

One important thing to mention when working with files in Rust is that, thanks to its ownership model, one just have to open file, and they are automatically closed as soon as the variable owning it goes out of scope, thus freeing the developer from the burden of remembering to close the file in the appropri-

ate points (which are not always easy to remember, especially in presence of conditional flows and multiple returns). This same principle is applied also to all the resources leaning on the open/close, or acquire/release, semantics, such as network connections and mutexes.

3.2.6 The Data Distribution Loop

Data are distributed using a loop that iterates until there are records to iterate on and, once `chunk_size` records are read (with the chunk size taken from the configuration TOML), they are sent to the output Kafka topic to a partition selected in a round-robin fashion from the first one.

If records to send are prepared before the time between each chunk send has elapsed, a wait for the remaining milliseconds is implemented.

After the messages are sent, the partition for the next chunk is selected, the timer is reset, and the vector of output messages is cleared. Then, the loop starts again processing new CSV records, until chunk size (or last record) is reached.

The code implementing this behavior is the following:

```
let mut chunk = Vec::with_capacity(chunk_size);

let mut start_time = Instant::now();
let mut partition = 0

let records = reader.records().count();
for (i, result) in reader.records().enumerate() {
    let record = result?;
    let data: Message = record.deserialize(None)?;
    let data_json = data.json_serialize();
    let record =
        Record::from_key_value(&config.kafka.topic[..],
   data.id, data_json.to_string())
            .with_partition(partition);
    chunk.push(record);

    if chunk.len() == chunk_size || i == records - 1 {
        // wait for all the chunk_sleep_in_ms to pass
        let elapsed = start_time.elapsed();
        if start_time.elapsed() <
   config.data.chunk_sleep_in_ms {
            sleep(config.data.chunk_sleep_in_ms - elapsed);
```

```
21          }
22
23          // send the chunk
24          for rec_chunk in chunk.chunks(100) {
25              producer.send_all(&rec_chunk)?;
26          }
27
28          println!("Sent {} records", chunk.len());
29
30          // reset for next chunk
31          chunk.clear();
32          partition = (partition + 1) % partitions_number;
33          start_time = Instant::now();
34      }
35  }
```

Listing 3.13: Implementation of the data distribution loop.

One important thing to notice is that, although a chunk is considered as a single batch of messages, in reality the chunk is sent in sub-chunks of at most 100 messages. One may wonder the motivation behind this choice, and the motivation is that it was empirically determined that (probably because of a bug in the crate) sending larger batches using this crate caused unpredictable behavior, with some messages that were lost in a seemingly random fashion. This was the only effective mitigation of this behavior that was found, other mitigation strategies such as tweaking Kafka's or producer's parameters were tried with no luck.

3.3 Edge Data Processing

The edge data processing is indeed the core of this thesis. The program that will be described runs on nodes ideally deployed at the edge of the cloud, that continually receives data from one or more partitions of a Kafka topic, elaborates them (in particular, the geohash of the message is calculated and, from it, also the neighborhood of provenance of the message), and then sends them to some other output topics (from which the message will be read by the node performing data analysis).

The duties of this program are simple:

- read the incoming message stream (which is one or more partitions of a Kafka topic)

- save each incoming message in an in-memory array of messages, and add information about the geohash and neighborhood to each message
- every window milliseconds (with a configurable window) sample the data that were read until that moment, since the end of the previous window
- send the sampled message to an output stream (consisting of one or more Kafka topics).

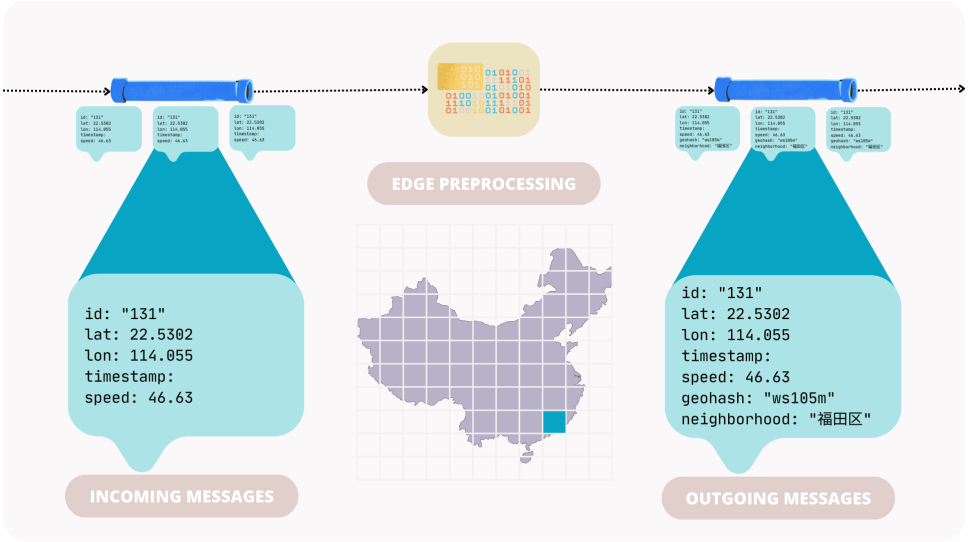

Figure 3.5: Figure shows what are the messages format in the input and output stream. Moreover, it is illustrated that the geohash and some geographical information about the topology of the incoming data's region are used to calculate the neighborhood of the message.

Just like the data distribution program, the program was developed in Rust, and makes use of the kafka-rust crate[50].

This binary too is configured with both a configuration TOML file and some command line options, and comes with various subcommands to manage its life-cycle.

3.3.1 TOML Configuration File Structure

The program is configured with a TOML file placed in the same directory as the binary, and named `kafka_edge_config`. The following is an example of configuration file that can be used to configure the edge program:

```
1 [kafka]
```

```
zookeeper = [ "zookeeper:2181" ]
brokers = [ "kafka:9092" ]

[data_in]
source_topic = "datain"
consumer_group = "datain_cons"
partitions_to_consume = [ "1" ]

[data_out]
target_topic = "dataout"
send_every_ms = 10000
send_strategy = "NeighborhoodWise"
sampling_strategy = "Stratified"
neighborhoods_file =
   "/data/china/neighborhood/shenzhen_converted.geojson"
```

Listing 3.14: Example configuration TOML for the edge program.

In this configuration file, three sections are defined, each with different fields, with the following meaning:

- the `kafka` section is the section in which the options strictly related to Kafka's behavior are placed, and the fields `zookeeper` and `brokers` have the same meaning they had in the data distribution configuration file
- the `data_in` section contains the configurations available for the incoming messages
  - `source_topic` is the field used to configure the Kafka topic to consume in input
  - `consumer_group` is the field to specify the Kafka consumer group to be part of
  - `partitions_to_consume` is the field in which the partitions that the binary need to consume are listed. It is useful when one wants to simulate unbalanced traffic on the networks (it is sufficient to configure a node to consume more partitions than the other)
- the `data_out` section is the section in which the options relative to how to configure the output stream are specified
  - the `target_topic` field contains the name (or the shared prefix) of the output topic (or topics)

- the `send_every_ms` field can be used to set the window of time in which the program just consumes and process messages, before sampling and sending them. To be more clear, the collected input messages will be sampled and sent every this many (`send_every_ms`) milliseconds
- the `send_strategy` field contains the strategy to be used to send the messages. There are two strategies one can choose, and they will be discussed in depth later in the chapter
- the `sampling_strategy` field contains the strategy to be used for sampling. Also in this case, there are two strategies, that will be discussed later too
- the `neighborhood_file` contains the absolute path to the GeoJSON file in which the polygons of the incoming data city (the city of Shenzhen in China, in our case). This file is used to determine the neighborhood of a message, given its geohash.

3.3.2 Messages Structure

The message structure is very similar to the one described in the Listing 3.9, but with the addition of two optional fields:

- `geohash`, where the geohash of a message is saved
- `neighborhood`, where the calculated neighborhood of a message is saved.

Also, the implementation block for the message structure is slightly different from the one provided for the data distribution program. In particular, the method to JSON serialize the message here includes the two newly introduced fields, and a method to JSON deserialize, and another one to calculate the geohash of the message are provided. The following is the updated implementation block for the structure holding the message:

```
impl Message {
    /// Serialize the `Message` in JSON format.
    ///
    /// # Panics
    /// Panics if the message does not have a geohash and a
   neighborhood.
    pub fn json_serialize(&self) -> Value {
        json!({
            "id": self.id,
            "lat": self.lat,
```

```
            "lon": self.lon,
            "time": self.time,
            "speed": self.speed,
            "geohash": self.geohash.as_ref().unwrap(),
            "neighborhood":
    self.neighborhood.as_ref().unwrap_or(&String::from("")),
        })
    }

    /// Deserialize a JSON `Message`.
    pub fn json_deserialize(message: &[u8]) -> Result<Message,
    Error> {
        serde_json::from_slice(message)
    }

    /// Calculate the geohash of the message.
    pub fn geohash(&self) -> Result<String, GeohashError> {
        geohash::encode(
            Coord {
                x: self.lat,
                y: self.lon,
            },
            6,
        )
    }
}
```

Listing 3.15: Message structure used by the data edge processing program.

The geohash encoding was not directly implemented, but a third-party crate was used. The crate is called *geohash*, and it is part of the collection of crates *GeoRust*[39], that comprise many different crates to easily handle geospatial data.

3.3.3 Extracting the Neighborhood From a Message

To extract information about the neighborhood from which a message is coming, three things are necessary:

- the first prerequisite it to have a rough idea about the provenance of messages, for example, knowing that they all come from a particular city or region (in our case, Shenzhen)

- having stored somewhere the information about the various neighborhoods shapes (here we focus on neighborhoods, but this can be generalized to any area of interest)

- knowing the latitude and longitude from which the message come.

With these three prerequisites satisfied, it is in general possible to extract information answering the question: *"from which area this message comes?"*. Or, in our specific case: *"from which neighborhood this message comes?"*.

In computational geometry, this problem is called the point-in-polygon (or PIP) problem, and an algorithm that solves it is the ray casting algorithm[52].

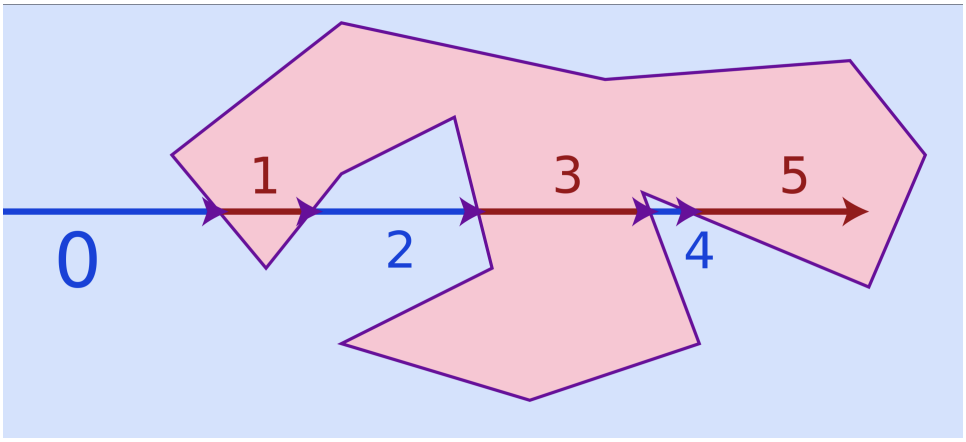

Figure 3.6: The figure shows how intersections are counted for a ray cast from outside the polygon. As visible in the figure, once the ray will cross the last edge, the intersection count will become even, and thus we deduct that the ray originated outside the polygon.

Practically speaking, using the ray casting algorithm to find whether the point is inside or outside a simple polygon consist in testing how many times a ray, starting from the point and going in any fixed direction, intersects the edges of the polygon. If the point is on the outside of the polygon, the ray will intersect its edge an even number of times, if it is inside, the ray will intersect its edge an odd number of times. This algorithm is sometimes also known as the crossing number algorithm or the even–odd rule algorithm, and was known as early as 1962[52].

Apart from the problems originating from the fact that the computer's floating arithmetic has finite precision, the problem with this algorithm is that it is too time-consuming to be deployed on an edge node, often with limited computing resources. Moreover, the degree of precision offered by this

algorithm, is often not even needed for many geospatial applications, and it just gives a computational overhead (meaning increased costs, more or more powerful nodes, more energy consumed) that does not offer a better accuracy on the data analysis we need to perform.

Thus, a smarter approach for our use case, is that of leveraging two things:

- a GeoJSON file with, for each neighborhood we're interested in, the corresponding
- the possibility of encoding each (valid) coordinate with a geohash.

From the GeoJSON file, we could read the polygons of each neighborhood, and precompute a map mapping each neighborhood, with the list of geohashes that are contained in it.

Then, when a new message arrives, we compute the geohash of its coordinates, and we can visit the map to find the neighborhood containing the message's geohash.

To be even more efficient, we could invert the map, and map each geohash, with its corresponding neighborhood. This new map, although less space efficient, can guarantee the retrieval of the neighborhood in which a geohash is located in constant time, $\mathcal{O}(1)$. The concept of behind this optimization is shown in Figure 3.7.

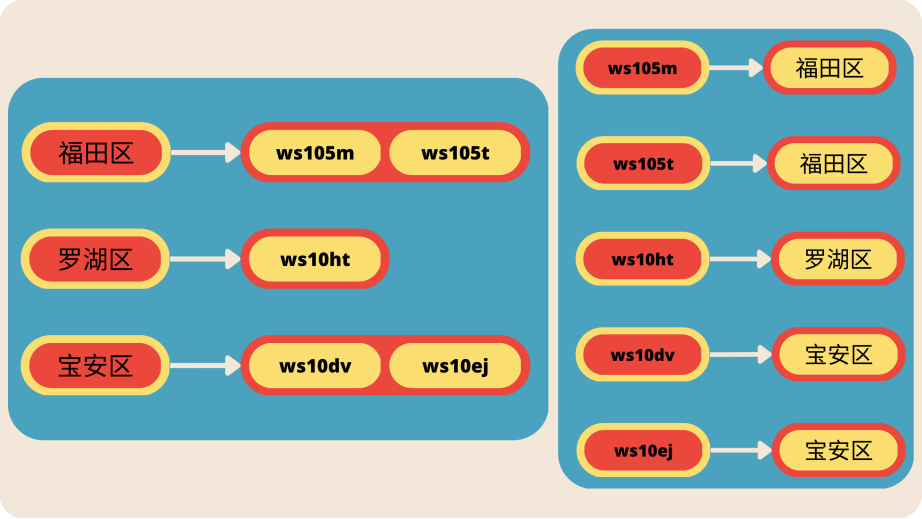

Figure 3.7: The figure shows on the left the map mapping each neighborhood with the vector of geohashes inside that neighborhood, and on the right the reversed map, with each geohash mapped with its neighborhood, is shown.

By using such a technique, it is possible to save a lot of computational power, and achieve a result that is similar in accuracy, for most cases, but that

suffers a little bit more on the edge cases. Indeed, the less precise the geohash is, the less precise the boundaries of the polygon are detected. In fact, it may happen with this technique, that a message is a bit outside the actual polygon, but its geohash is considered as inside the polygon, and thus the point will be considered as if inside the polygon, or vice versa (this depends on whether consider only the geohashes that are completely inscribed in the polygon or the geohashes that circumscribe it).

In a real-world scenario, this becomes a major problem on if the polygon size and the geohash precision cover similar areas, otherwise, if inside a polygon, hundreds of geohashes are present, the problem arises only for a small percentage of the total data.

Implementation

The implementation of an algorithm capable of taking as input a vector of GeoJSON features, and returning a map, mapping each geohash with a neighborhood, is not directly supported by any of the crates of GeoRust. But, the primitives the *geo*, *geohash* and *geojson* crates make available, are sufficient to implement such an algorithm. Moreover, an easy integration between these crates is provided by *geo-types*, a crate that defines geometric types for the GeoRust ecosystem.

The code implementing such a functionality in the edge program is the following:

```
pub const BASE_32: [char; 32] = [
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'b',
   'c', 'd', 'e', 'f', 'g', 'h', 'j', 'k',
    'm', 'n', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
   'y', 'z',
];

pub fn children(gh: &String) -> Vec<String> {
    BASE_32.iter().map(|c| format!("{}{}", gh, c)).collect()
}

pub fn bbox(gh: &str) -> Option<Polygon<f64>> {
    if gh.is_empty() {
        let min = Coord::<f64>::from((-180.0, -90.0));
        let max = Coord::<f64>::from((180.0, 90.0));
        return Some(geo_types::Rect::new(min,
   max).to_polygon());
    }
```

```
    match geohash::decode_bbox(gh) {
        Ok(rect) => {
            let bl = rect.min();
            let tr = rect.max();
            let outer = LineString(vec![
                Coord::from((bl.x, bl.y)),
                Coord::from((tr.x, bl.y)),
                Coord::from((tr.x, tr.y)),
                Coord::from((bl.x, tr.y)),
                Coord::from((bl.x, bl.y)),
            ]);
            Some(Polygon::new(outer, Vec::new()))
        }
        _ => None,
    }
}

pub fn contains(outer: &Polygon<f64>, inner: &Geometry<f64>)
    -> bool {
    match *inner {
        Geometry::Point(ref g) => outer.contains(g),
        Geometry::Line(ref g) => outer.contains(g),
        Geometry::LineString(ref g) => outer.contains(g),
        Geometry::Polygon(ref g) => outer.contains(g),
        Geometry::Rect(ref g) =>
    outer.contains(&g.to_polygon()),
        Geometry::Triangle(ref g) =>
    outer.contains(&g.to_polygon()),
        Geometry::MultiPoint(ref mp) => mp.0.iter().all(|p|
    outer.contains(p)),
        Geometry::MultiLineString(ref mls) =>
    mls.0.iter().all(|ls| outer.contains(ls)),
        Geometry::MultiPolygon(ref mp) =>
    mp.0.iter().all(|poly| outer.contains(poly)),
        Geometry::GeometryCollection(ref gc) =>
    gc.0.iter().all(|geom| contains(outer, geom)),
    }
}

pub fn covering(geom: &Geometry<f64>, level: usize) ->
    Vec<String> {
```

```
    use geo::algorithm::intersects::Intersects;
    let mut ghs: Vec<String> = vec![];
    let mut queue: Vec<String> = vec!["".to_string()];
    while !queue.is_empty() {
        let gh = queue.pop().unwrap();
        if let Some(poly) = bbox(&gh) {
            if contains(&poly, geom) || poly.intersects(geom) {
                if gh.len() < level {
                    queue.extend(children(&gh));
                } else {
                    ghs.push(gh);
                }
            }
        }
    }
    ghs
}

/// Get a map of geohashes for each neighborhood.
///
/// # Panics
/// Panics if a feature has an invalid geometry.
pub fn get_geohashes_map_from_features(features:
    &Vec<Feature>) -> HashMap<String, Vec<String>> {
    let mut geohashes_map: HashMap<String, Vec<String>> =
    HashMap::new();
    for feature in features {
        let geometry =
    Geometry::try_from(&feature.geometry.clone().unwrap())
            .unwrap();
        let covering_geohashes = covering(&geometry, 6);
        geohashes_map.insert(
            feature
                .properties
                .clone()
                .unwrap()
                .get("NAME")
                .unwrap()
                .to_string(),
            covering_geohashes,
        );
```

```
87      }
88      geohashes_map
89  }
90
91  pub fn invert_neighborhood_geohashes_map(
92      neigh_gh_map: &HashMap<String, Vec<String>>,
93  ) -> HashMap<String, String> {
94      let mut inverted_geohashes_map: HashMap<String, String> =
    HashMap::new();
95      for (neighborhood, geohashes) in neigh_gh_map {
96          for geohash in geohashes {
97              inverted_geohashes_map.insert(geohash.clone(),
    neighborhood.clone());
98          }
99      }
100     inverted_geohashes_map
101 }
```

Listing 3.16: Implementation of the data distribution loop.

The implementation was inspired by the *geoq* repository[53], which implement the functionality of getting the list of geohashes that intersects (meaning also that only partially overlaps) the polygon.

The implementation was then slightly adapted to iterate over a list of polygons, so to obtain a map mapping each neighborhood name with the list of geohashes it contains, and the function to invert the resulting map, thus mapping each geohash with its neighborhood, was implemented.

To explain briefly, the purpose of each function:

- `children` is a function that, given a geohash, returns the list of "children" geohashes, that is the list of geohashes that are one character longer and that have the initial geohash as a shared prefix. These children are always 32, since the geohash encoding base is base-32, and because of the properties of the geohash, all the children are located completely inside the "parent" geohash

- `bbox` is a function that given a geohash returns the polygon (a rectangle) that circumscribe all the points having that geohash

- `contains` is a function that given two polygons, outer and inner, returns true if inner is contained in outer, and false otherwise

- `covering` is the core function that given a geometry, return the list of geohashes covering that geometry. It works by continually extending a queue of "candidate" geohashes to test (increasing the geohash precision until the desired precision is reached), and by adding to the results list the geohashes of the desired precision that intersects the given geometry. The procedure is iterated until the queue empties, and at that point the results list is returned

- `get_geohashes_map_from_features` is a function that given a vector of features (extracted from a GeoJSON file) returns a hashmap with the neighborhood name as the key, and the list of geohashes covering the neighborhood as the value. It works by calling `covering` for each feature in input

- `invert_neighborhood_geohashes_map` is a function that, given in input the hashmap returned by `get_geohashes_map_from_features`, inverts the values with the key, and thus returns a hashmap mapping the geohash (used as the key) with the corresponding neighborhood.

3.3.4 Sampling Strategies

The program comes with two sampling strategies one can choose:

- *"Random"* which implements the Simple Random Sampling strategy

- *"Stratified"* which implements the Stratified Sampling strategy.

The Random strategy simply retains a certain percentage of the incoming messages. This is implemented by retaining each input message with a certain probability (if one wants to retain 60% of the input messages, each message would be retained by choosing a random boolean with the 60% of chances of being true).

The Stratified strategy, on the other hand, is a bit more complex. The goal is to divide the input messages into strata (the stratification being done on the message geohash), and then sampling each stratum independently. For example, if one wants to retain the 60% of the incoming messages, one should group each message by geohash, and then retain 60% of the messages in each group.

The following is the code used to implement this strategy:

```
1 let total_size = messages.len();
2 let sample_size = total_size as f64 * sampling_percentage;
```

```
let mut groups: HashMap<&str, Vec<&Message>> = HashMap::new();
for message in messages as &[Message] {
    groups
        .entry(message.geohash.as_ref().unwrap())
        .or_insert_with(Vec::new)
        .push(message);
}

let mut sample_sizes: HashMap<&str, usize> = HashMap::new();
for (geohash, group) in &groups {
    if group.len() == 1 {
        sample_sizes.insert(
            geohash,
            either!(rng.gen_bool(sampling_percentage) => 1; 0)
    as usize,
        );
    } else {
        let proportion = group.len() as f64 / total_size as
    f64;
        sample_sizes.insert(geohash, (proportion *
    sample_size) as usize);
    }
}

let sampled_groups: Vec<Vec<&Message>> = groups
    .par_iter()
    .map(|(geohash, group)| {
        group
            .iter()
            .cloned()
            .choose_multiple(&mut rand::thread_rng(),
    sample_sizes[geohash])
    })
    .collect();
*messages =
    sampled_groups.into_iter().flatten().cloned().collect();
```

Listing 3.17: Implementation of the stratified sampling strategy.

To increase efficiency, the parallel iterator provided by the *rayon*[54] was used. The rayon parallel iterator allows performing operations (such as filtering or mapping) in parallel, by using "behind the scenes" a thread pool, whose size

depends on the available degree of parallelism of the runtime.

3.3.5 Sending Strategies

The strategies to send out data implemented are three:

- sending messages to a single partitioned Kafka topic, with the partition chosen randomly
- sending all messages to a single partitioned Kafka topic, with the partitions selected in a round-robin fashion
- creating a topic for each neighborhood (and a topic for the messages that we were unable to assign a neighborhood to), and sending each message to the Kafka topic dedicated to messages from the same neighborhood.

The three strategies are called respectively: *Random*, *RoundRobin*, and *NeighborhoodWise*.

The first two strategies are indeed quite similar from one another. The most different one of the three is the *NeighborhoodWise* sending strategy. This strategy is the most complex of the three, and the one that is arguably the most interesting to test.

3.3.6 How to Use the Binary

The binary comes with many subcommands that help in managing its entire life-cycle.

Specifically, the main subcommands are:

- *topic* - a subcommand that allows managing the topic-related stuff
- *config* - a subcommand allowing to manage the configuration in an easy way.

The topic command has itself other subcommands, specifically:

- `create` - a command that allows to create topics. When using this command to create topics for the *NeighborhoodWise* strategy, it is important to remember to set the option `--for-nbw-strat` (this is necessary because one may also override the sending strategy via command line option). Moreover, it is possible to set the replication factor for the topic(s) that will be created
- `delete` - a command that allows deleting topics.

The config command, has itself other subcommands too. Specifically:

- `create` - a command to create the configuration from a template. It is also possible to directly specify the source and target topic of the configuration that will be created
- `replace` - a command that takes a file path as input, and it is used to replace an existing (or non-existing) configuration with the provided file
- `show` - a utility command that prints to standard output the current configuration.

Moreover, when running the main program, it is possible to supply the following options through command line:

- `--sampling-percentage` (or `-s`) is the option configuring which percentage of the incoming messages at each window to retain. It must be an integer between *0.0* and *1.0*, and when not supplied it defaults to *0.5*
- `--override-send-strategy` is the option used if one wants to override the sending strategy (for example to switch from *Random* to *NeighborhoodWise*.

3.4 Cloud Data Analysis

For the data analysis task, it was chosen to use Apache Spark, as said in the previous chapter, because of its in-memory data processing capabilities.

Spark allows developers to write data analysis programs in a variety of languages. Notably:

- *Scala* - a modern language that combines object-oriented and functional programming, and that it is the language in which Spark itself is written
- *Java* - one of the most popular object-oriented languages, and among the most used today
- *Python* - one of the most used languages for data analysis, with a simple syntax, and very well suited for rapid prototyping
- *R* - a language specifically suited for statistical computing.

Among these choices, the most used ones are Python and Scala. After experimenting with both languages, it was chosen to focus on Python, because it is easier to use even without advanced IDE support, which is not true for Scala, and because I was also more familiar with it.

The Python API for Spark is called PySpark, and enables to execute Python scripts in Apache Spark. It is possible to execute Python code both by an interactive shell, or by the spark-submit mechanism, which is a mechanism that can be used to launch applications on a cluster.

3.4.1 The Performed Data Analysis

The data analysis performed aimed at calculating the average speed for each geohash in each thirty-minute window. The data are processed in-memory from an input stream consisting of several topics (all the output data topics that may be used by the edge processing nodes), and saved to CSV every minute (by overwriting the previous file).

The following snippet is the source code of the data analysis performed by Apache Spark:

```
import time
from datetime import datetime
from pyspark.sql import SparkSession
import pyspark.sql.functions as F
from pyspark.sql.types import StructType, StructField,
    StringType, DoubleType, TimestampType, TimestampNTZType

KAFKA_BOOTSTRAP_SERVERS = "kafka:9092"
KAFKA_TOPICS = "dataout_0,dataout_1,dataout_2,dataout_3," + \
               "dataout_4,dataout_5,dataout_6,dataout_e"

OUTPUT_PATH = "/vol/"

SCHEMA = StructType([
    StructField("id", StringType(), False),
    StructField("lat", DoubleType(), False),
    StructField("lon", DoubleType(), False),
    StructField("time", TimestampType(), False),
    StructField("speed", DoubleType(), False),
    StructField("geohash", StringType(), False),
    StructField("neighborhood", StringType(), False),
])

spark: SparkSession = SparkSession.builder.appName(
    "write_traffic_sensor_topic").getOrCreate()

```

```
# set log level to WARN
spark.sparkContext.setLogLevel("WARN")


df_stream = spark\
    .readStream.format("kafka")\
    .option("kafka.bootstrap.servers", KAFKA_BOOTSTRAP_SERVERS)\
    .option("subscribe", KAFKA_TOPICS)\
    .option("startingOffsets", "earliest")\
    .load()
# deserialize the data from kafka
df_stream = df_stream.selectExpr("CAST(value AS STRING)")
# json deserialie using schema
df_stream = df_stream.select(F.from_json(
    F.col("value"), SCHEMA).alias("data")).select("data.*")

df_stream = df_stream.groupBy(F.window("time", "30 minutes"),
   F.col("geohash"))\
    .agg(F.avg("speed").alias("avg_speed"))
# creates a write stream with the query name
df_stream.writeStream\
    .queryName("query").format("memory").trigger(
        processingTime="30 seconds")\
    .outputMode("complete").start()

time.sleep(60)
while True:
    df = spark.sql(f"select * from query")
    if df is not None:
        df.show()
        df = df.orderBy(F.col("window.start").desc())
        #stringify the time
        df = df.withColumn("window",
           F.col("window.end").cast(StringType()))
        df = df.withColumnRenamed("window", "time")
        # save the results in a csv file
        filename = "avg_speed"
        df.repartition(1).write\
            .mode("overwrite")\
            .option("header",True)\
            .csv(OUTPUT_PATH + filename)
```

```
65
66      time.sleep(60)
```

Listing 3.18: Pyspark data anlysis script.

As it is possible to notice from the script, the incoming data from the input streams are first deserialized, so that the individual fields are available for querying, then the actual data analysis is performed.

Data in the stream are grouped by time in windows of thirty minutes, and by geohash. Finally, the average speed for each geohash is computed, and the new column is given the alias name `avg_speed`.

A write stream writing in-memory the results of this computation, and triggering each thirty seconds an update of type complete (meaning the whole table is recomputed on each update), is created, and is given the query name *"query"*.

Thanks to the query name, it is possible to use the Spark SQL API to perform SQL a query that selects all the table computed in-memory. From the query results, the window field is replaced with the only window end timestamp, because by knowing that each window is thirty minutes, it is not necessary to save both the extremes of the window, and saving a single timestamp makes it easier to manipulate the field by the data visualization web application. The resulting data frame is then saved in CSV every sixty seconds.

3.5 Data Visualization

Visualizing data is an important task in many scenarios, because graphical representations of data are often more impactful and clear to humans than the numerical datasets from which they are extracted.

Thus, to both monitor and explore the results of the data analysis performed by Apache Spark, a simple web server serving an interactive map visualizing the results of the analysis was implemented.

Among the many options for a web sever available on the market today, it was chosen the Python framework for web applications named Flask.

Flask is a micro web framework written in Python. It is considered a micro-framework because it does not require particular libraries or tools, and it has no database abstraction layer, nor form validation, or any other components where pre-existing third-party libraries provide common functions. Nonetheless, it supports extensions that can add application features as if they were implemented in Flask itself.

Flask was created by Armin Ronacher of Pocoo, an international group of Python enthusiasts formed in 2004. According to its creator, the idea was

originally an April Fool's joke that was popular enough to make into a serious application.

In April 2016, the Pocoo team was disbanded and development of Flask and related libraries passed to the newly formed Pallets project[55]. Flask has become popular among Python enthusiasts, and, in general, it is among the most used web frameworks nowadays.

Flask was chosen among the other choices, such as Django, Express, Next.js, and others, because it combined the following characteristics making it really suited for the particular purpose of this thesis:

- it is very easy to set up (just few lines of Python are needed)
- it uses Python as a language, that, in turn, has many good libraries for visualizing data on a map (notably, Folium[56])
- it offers an easy-to-use template system that is really useful for basic customization of static sites.

3.5.1 Flask Setup

The setup of the web server is the simplest possible, with the only base path route (`/`) implemented for the HTTP GET method only.

The route tries to fetch the latest results from the volume in which the data analysis results are saved, and displays them on an interactive map made using Folium[56]. Folium builds on the data wrangling strengths of the Python ecosystem and the mapping strengths of the leaflet.js library to provide easy to develop interactive maps.

The results are, as showed in the chapters, saved in a CSV containing the average speed for each geohash in each available thirty-minute window.

The returned page presents, other than the map, an interactive list from which it is possible to choose the time window one is interested in. The list items have a clickable link that makes a GET request to the same page, but inserting the window in the request query parameters.

When no query parameters are passed, then the first window appearing in the results dataset is returned by default.

The map is created using both Folium and pandas[57], which is one of the most powerful and popular data analysis libraries available today. Once created, the map is saved in HTML format and embedded in the home page.

The final result of the web app is shown in Figures 3.8 and 3.9. As noticeable, the resulting page is very minimal, as creating a complex and feature-rich web application goes outside the scope of this thesis.

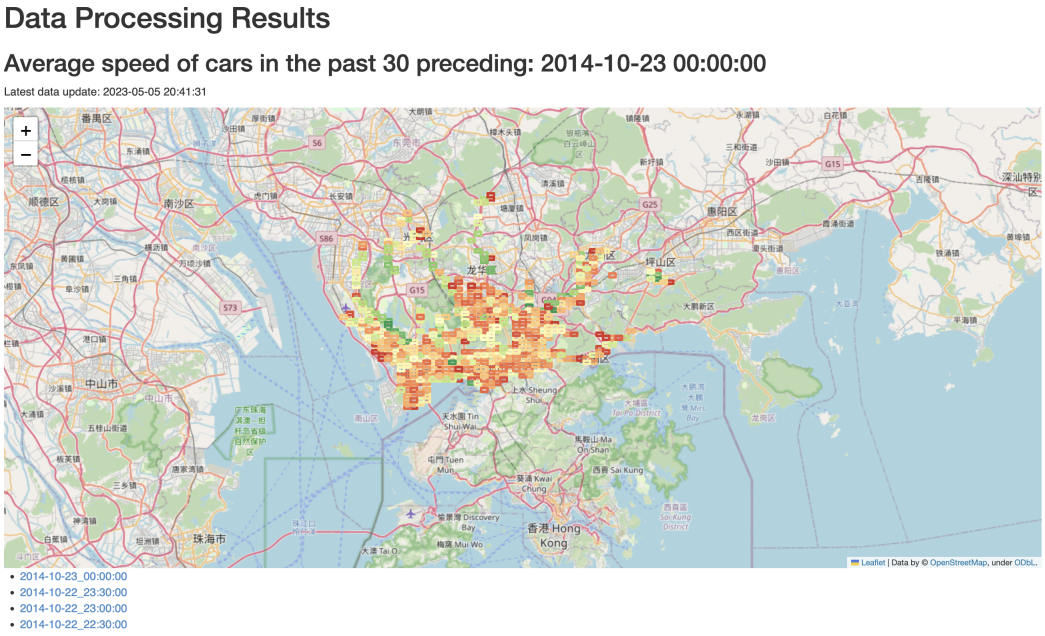

Figure 3.8: The figure shows an example of what the interactive map looks like. Each colored rectangle overlaid on the map is a geohash, and its color is a function of the average speed: the higher the average speed the more the color is toward the green, the lower the speed the more red it is.

3.5.2 Interactive Map Creation

A function was defined in the source code to create and save (in HTML format) the interactive map, starting from a pandas data frame containing the data read from the data analysis output CSV file.

The following is the source code of the function:

```
def process_data(data: pd.DataFrame, time: str):
    # map from red to green based on the avg_speed field
    cmap = linear.RdYlGn_09.scale(0, 60)
    # add color field to the data
    data['color'] = data['avg_speed'].apply(lambda x: cmap(x))
    data['lat'] = data['geohash'].apply(lambda x:
        geohash.decode(x)[0])
```

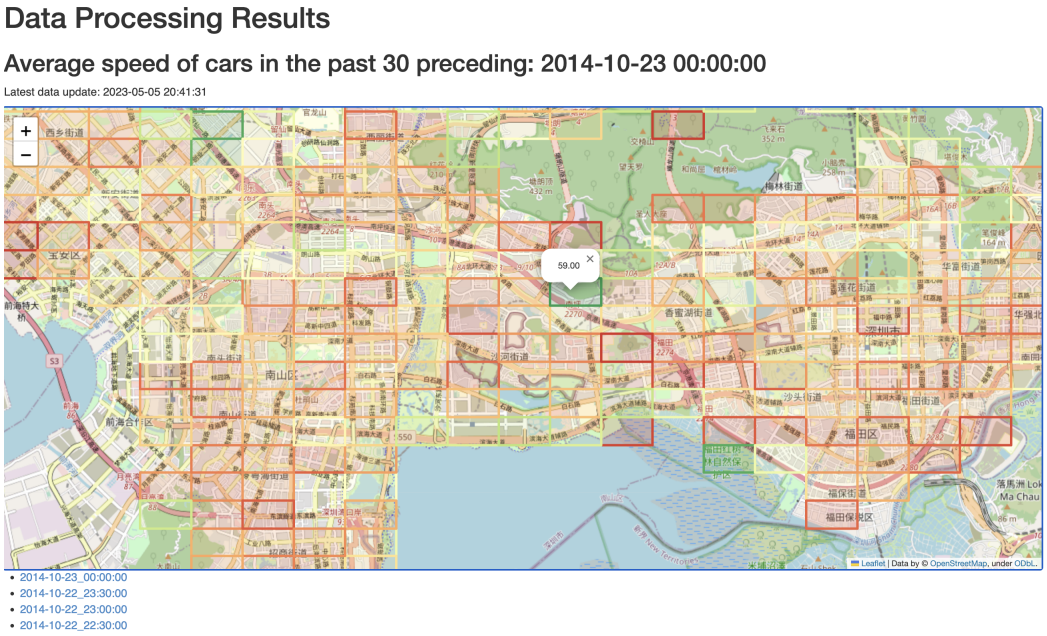

Figure 3.9: The figure shows the interactive map zoomed. Note that, if you click on a particular geohash, a label with its exact average speed for the selected time window is shown.

```
7       data['lon'] = data['geohash'].apply(lambda x:
            geohash.decode(x)[1])
8
9       data = data[data['time'] == time]
10      data.reset_index(inplace=True) # reset index to avoid
            problems (like index 0 not existing)
11
12      # Create a map centered on the first geohash in the dataset
13      #map = folium.Map(location=[data['lat'][0],
            data['lon'][0]], zoom_start=10)
14      map = folium.Map(location=[22.59, 114.11], zoom_start=10)
15
16      for _, row in data.iterrows():
17          bounds = get_rectangles_bounds(row['geohash'])
18          if bounds is None:
19              continue
20
21          folium.Rectangle(bounds=bounds, color=row['color'],
                fill=True, fill_color=row['color'], popup="%.2f" %
```

```
            round(row['avg_speed'], 2)).add_to(map)
22
23      map.save('templates/map.html')
```

Listing 3.19: Function creating and saving the map using a pandas data frame for the geohash data, and Folium to create the interactive map.

Chapter 4

Results and Discussion

In this chapter, the experimental results achieved by testing several aspects of the architecture that was implemented in the previous chapters are presented to the reader, together with a brief discussion about their meaning and relevance.

The setup, as thoroughly described in the previous chapters, is composed of many interacting containers, with different duties each, and many things may have been worth of proper testing.

Anyway, the experiments that were conducted mainly focused on the following aspects:

- measuring the Kafka performances, and the performances of the binaries using Kafka, in various scenarios, and with different traffic loads
- measuring the performances of Apache Spark in performing the data analysis task
- measuring the accuracy of the obtained results in different scenarios, and with different configuration parameters (for example, by varying the sample size, or the sampling strategy).

4.1 Testing Setup

The tests were conducted on a dataset containing data for 664 taxis moving in the city of Shenzhen in China. The dataset contained data for the whole day of 22 October 2014, for the whole 24 hours, and amounted to a total of 1,155,653 entries.

In all the tests that were carried out, unless otherwise stated, the geohash length was set to six characters.

Different configurations were tested, by varying the configuration parameters shown in the previous chapter, such as the sampling window on the edge nodes,

the rate at which data were distributed to the nodes, and the sampling size and strategies.

All the tests were conducted on a 2020 MacBook Air powered by the M1 chip by Apple, with 8 GB of RAM and 256 GB of SSD.

4.2 Edge Nodes Performances

In this section, the performances of the edge nodes observed and measured by carrying out different types of tests are thoroughly exposed and commented.

The edge nodes were tested for the following several different aspects:

- Kafka performances in terms of throughput — that is, the number of messages a node was able to send in a unit of time
- stratified sampling processing time performances — that is, the time needed to sample a certain amount of messages
- accuracy of stratified sampling observed for different sampling percentages, analyzed and discussed by using both a quantitative and qualitative approach.

4.2.1 Kafka Performances

The theoretical throughput of Kafka is around one million writes per seconds. However, as always, when putting things into practice, especially if the computations that need to be done do not have a negligible impact on the performances, the practice may differ significantly from the theory.

The evidence collected suggests that there are two main factors limiting the performances:

- the sampling operation, which, depending on the implementation, may become increasingly time-consuming when there are a lot of data to sample. In our implementation, its impact on performance was reduced by switching to a parallel sampling implementation, which helped to make performance generally good.
- the other limiting factor for performances was the speed of the chosen Kafka library, which proved not to be exceptional, and in the future different implementations may be tried to improve performances.

Also, experimenting with the window size — i.e. the time window the edge nodes wait before preprocessing all data arrived — so that the incoming data in each window are not a huge volume can help in greatly reducing the impact of this computation on performances The tests without sampling, an example run

of which is shown in Figure 4.1, show that, the time to send data is dependent on the number of messages to send, but, experimental results also showed that the optimal number of messages to send in a batch, to optimize average performances, is around *20* thousands messages. Indeed, tests conducted showed that batches of *5* to *20* thousand of messages needs more or less the same time to be delivered, thus sending *20* thousand messages allows for a higher number of messages to be delivered in the same time, i.e. a higher throughput. From the tests conducted, it was noted that the time required to send *20* thousand messages is around *100* milliseconds.

In particular, the average time to send *20* thousand messages over *52* data points is milliseconds.

Anyway, although not completely clear from the obtained results, it is highly likely that these results are highly specific to the particular library (*kafka-rust*[50]) utilized. Probably, with a different library being used, better performances may be achievable, for example *rust-rdkafka*[58] looks like a promising candidate.

4.2.2 Stratified Sampling Compute Time Performances

Performances of sampling are crucial to the architecture that we implemented, as the edge nodes must rely on efficient sampling techniques in order to avoid becoming a bottleneck for the whole data processing pipeline. In fact, the performance gain obtained by moving data sampling to the edge of the cloud may be canceled by sampling time, if this turn out to be too high.

To achieve a good performance of the sampling on the edge nodes, various different implementations of the stratified sampling technique were tried, before finally settling for the one implemented using the rayon crate [54] (whose implementation is shown in Listing 3.17), that exploit multithreaded processing to improve performances.

The first implementation was very similar to the final one, but instead of using a parallel iterator, like the one provided by rayon, it was using a sequential one. The differences in performance were, as expected, huge, especially for larger volumes of data to sample.

Anyway, rigorous data were not collected for the sequential implementation, as an accurate comparison of the two methods was not considered relevant to the thesis objectives.

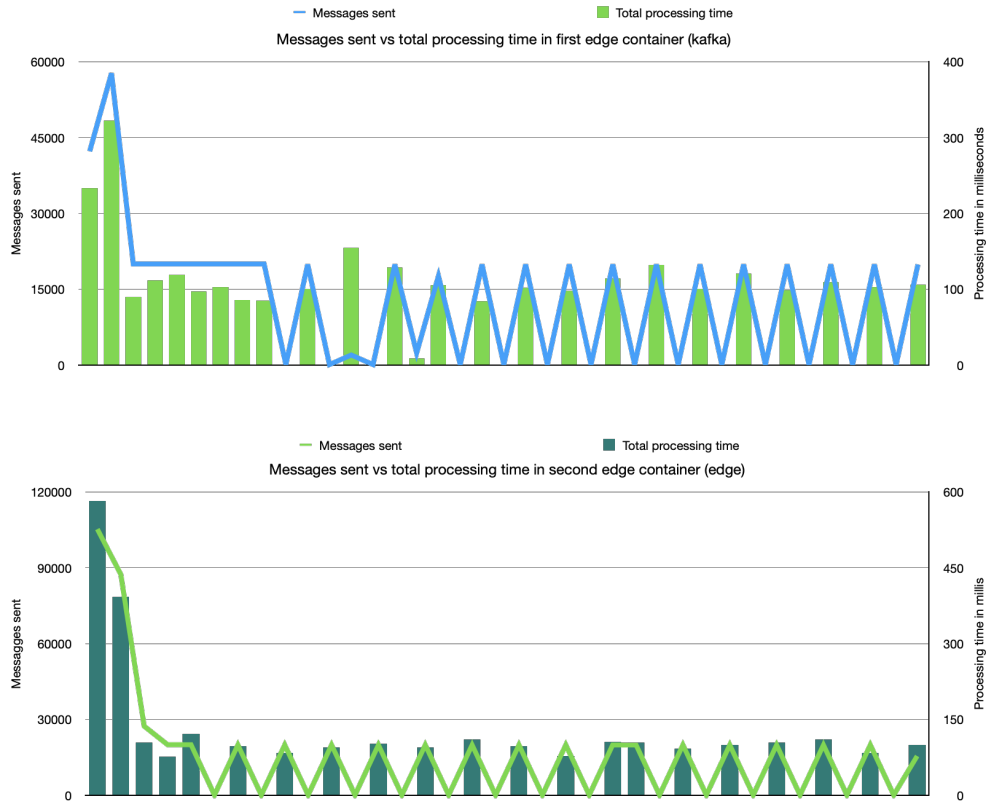

Figure 4.1: The figure shows the Kafka performance of a run of the edge binary on both the edge nodes (named kafka and edge, respectively).

Several test runs were conducted to collect data about the sampling performances. The aggregate results of both edge nodes for all the runs are shown in Figure 4.2. Each data point is a pair of messages to sample and sampling time.

The results of the individual containers are shown in Figure 4.3 for the edge node running both the data distribution and data preprocessing binaries, and Figure 4.4 for the other running only the latter binary.

The results show that the time increase almost linearly with the dimension of the problem, apart from three particularly slow samples of around 100'000 messages that all happened on the same run on the container running both the data distribution and preprocessing binaries, thus the most overloaded of the two. In general, apart from this very sporadic slow samples, the sampling technique shows a very consistent behavior.

Also, the test carried out showed no particular performance difference for different sampling percentages. In the tests, the sampling time appeared to be predominantly driven by the number of message to sample.

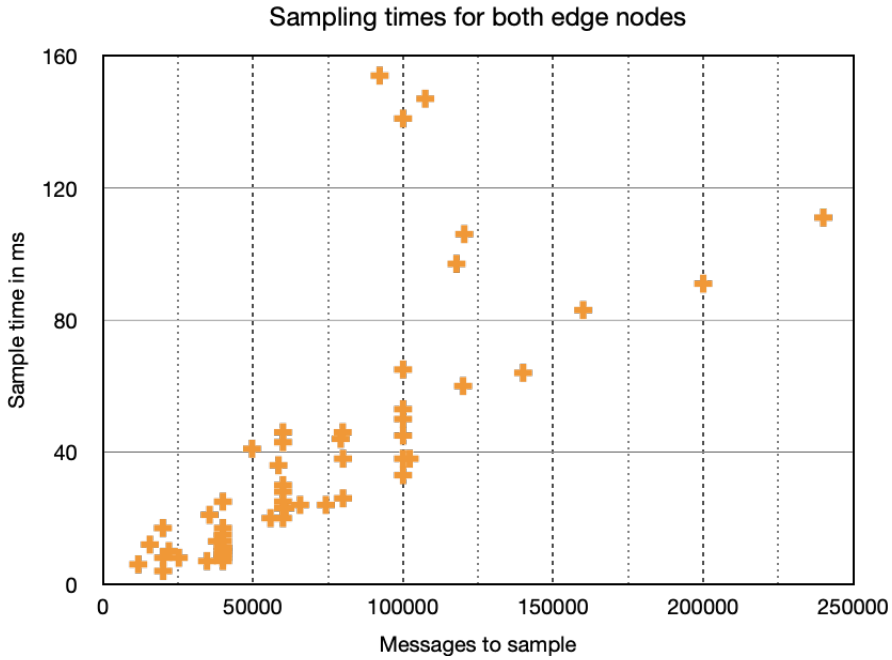

Figure 4.2: The figure shows each pair of messages to sample and sampling time in milliseconds on both the edge nodes of the architecture across all runs.

As a final note about the sampling technique used, the sampling implementation uses hash maps to both group by geohash and determine the size each sampled stratum should have.

Rust's standard hash maps uses SipHash algorithm, which is a state of the art non-cryptographic algorithm that shows consistent performances for various input sizes. However, in our case, we know a priori the length of the keys, and this length is particularly short (it is the length of the geohash), it may be worth it to try using different hashing algorithms and test if a bit of performance more can be squeezed out.

In particular, two particularly promising hashing algorithms for short keys may be the Fowler-Noll-Vo algorithm and the FxHash algorithm. While both algorithms have been proved not to exhibit particularly good performances for long keys, and they also have a higher collision rate than SipHash, they may be good candidates for our scenario, in which we have small keys and relatively short hash tables (at most a few thousands of entries)[59][60].

4.2.3 Stratified Sampling Accuracy Performances

To test the accuracy performances of the data, we decided to compare the Geohash with 6 characters of precision with the Geohash with 5 characters

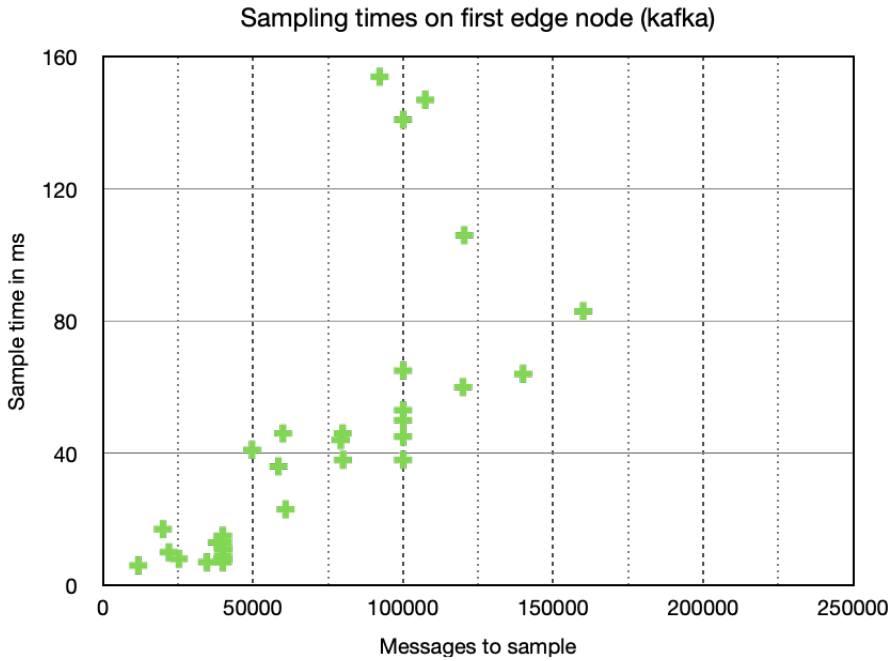

Figure 4.3: The figure shows each pair of messages to sample and sampling time in milliseconds on the container running the data distribution and data preprocessing binary, named *kafka* in the docker-compose file shown in Listing 3.7.

of precision. These two will be reffered to from now on as Geohash-6 and Geohash-5 respectively.

The results of testing stratified sampling performances were aimed at measuring both the MAPE (Mean Absolute Percentage Error) and MAE (Mean Absolute Error) of each run with a certain sampling percentage when compared to a baseline. The baseline was the results of a run in which no data were sampled out (i.e. the whole dataset was used) with Geohash-5 for the Geohash-5 test, and with Geohash-6 for Geohash-6.

Qualitative Analysis of Geohash-6

Since numbers alone are difficult to grasp for humans, the results were also compared graphically by making use of the web application with the interactive map that was created.

Shown below are some figures that exemplify the differences in the results obtainable with different sampling percentages.

As noticeable, on a visual level, the differences between the complete dataset (Figure 4.5, and the stratified sampling applied by retaining 80% of the original

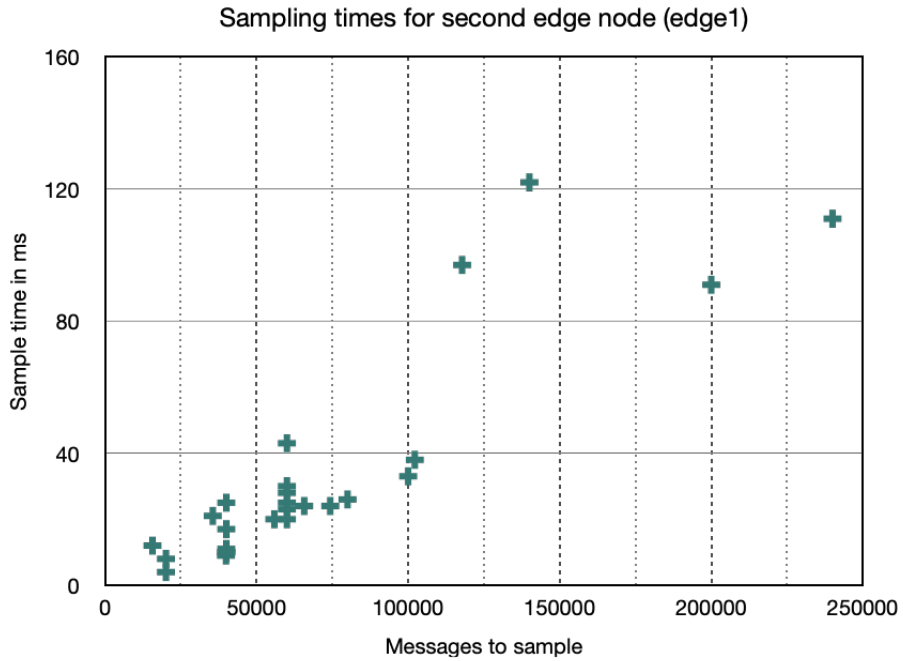

Figure 4.4: The figure shows each pair of messages to sample and sampling time in milliseconds on the container running the data preprocessing binary only, named *edge1* in the docker-compose file shown in Listing 3.7.

dataset (Figure 4.6, shows barely noticeable differences between them. This suggests that, by using stratified sampling together with a sample size of 80% of the original population, it is possible to be fairly confident that the analysis results are not too dissimilar from the actual situation.

Anyway, as more and more data are sampled out, (Figures 4.7, 4.8, and 4.9), it becomes increasingly evident the different with respect to the baseline. Anyway, however small was the tested sample (no sample size smaller than 20% was considered), the resulting image was similar when looked at in its totality, that is when considering the zones with a prevalence of orange to red rectangles, and the zones with a prevalence of yellow to green rectangles, remained more or less the same for all the sample sizes considered.

Also, for some geohashes the data points in the original dataset used for testing were just a few, making it very difficult to have an accurate estimate even for small sample sizes. This is just a hypothesis, but it could be easily tested in a future work with a bigger dataset.

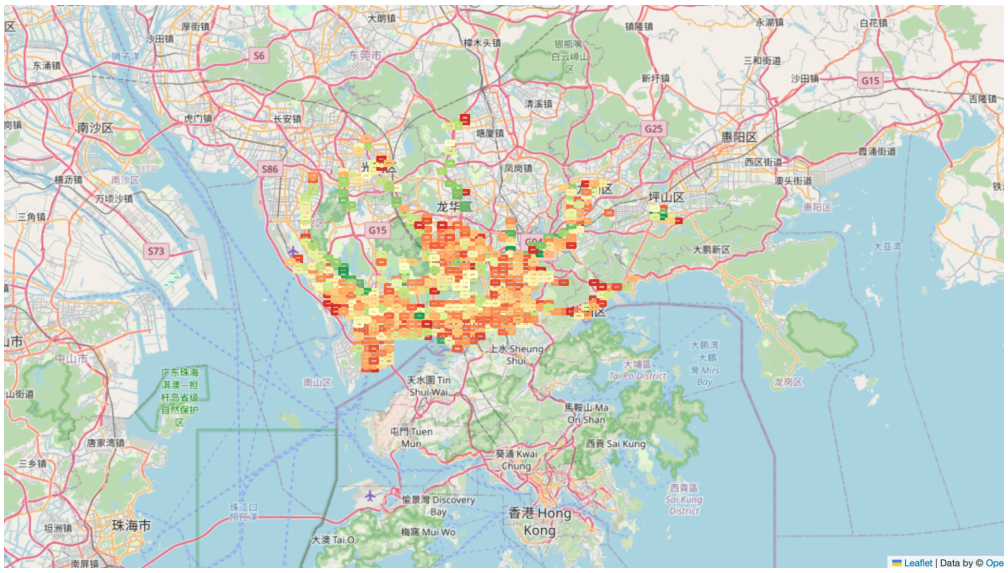

Figure 4.5: The figure shows the data visualization map resulting from a sampling rate of 100% (i.e. all data are retained). The figure results are referred to the traffic situation of taxis moving in the Shenzhen city from 23:30 on 22 October 2014 to 00:00 (midnight) on 23 October 2014.

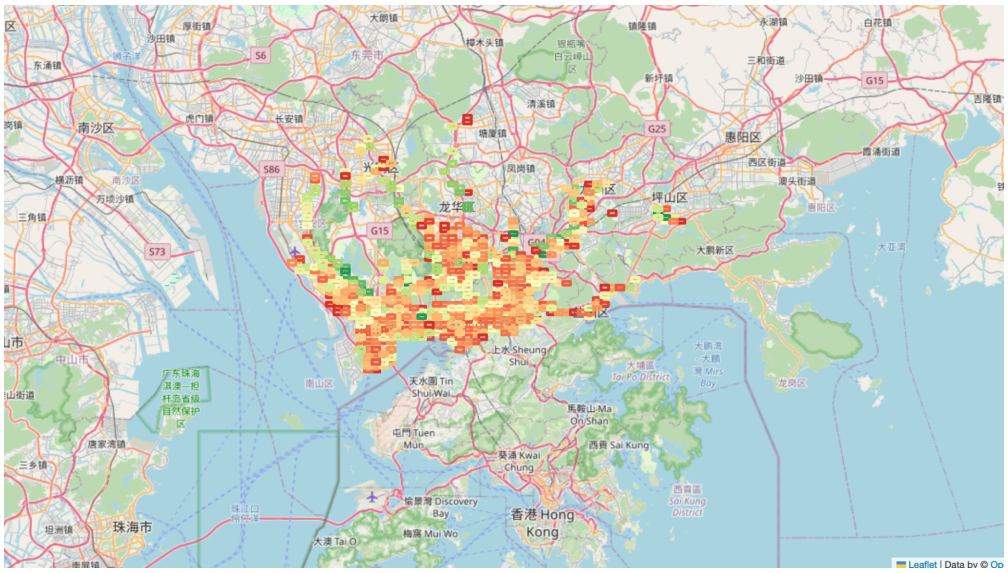

Figure 4.6: The figure shows the data visualization map resulting from a sampling rate of 80% (i.e. all data are retained). The figure results are referred to the traffic situation of taxis moving in the Shenzhen city from 23:30 on 22 October 2014 to 00:00 (midnight) on 23 October 2014.

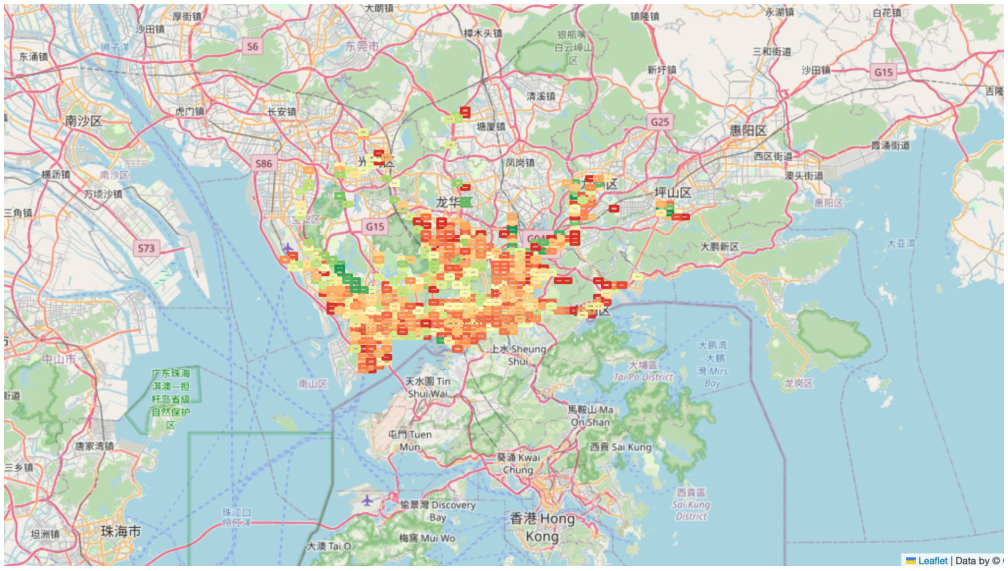

Figure 4.7: The figure shows the data visualization map resulting from a sampling rate of 60% (i.e. all data are retained). The figure results are referred to the traffic situation of taxis moving in the Shenzhen city from 23:30 on 22 October 2014 to 00:00 (midnight) on 23 October 2014.

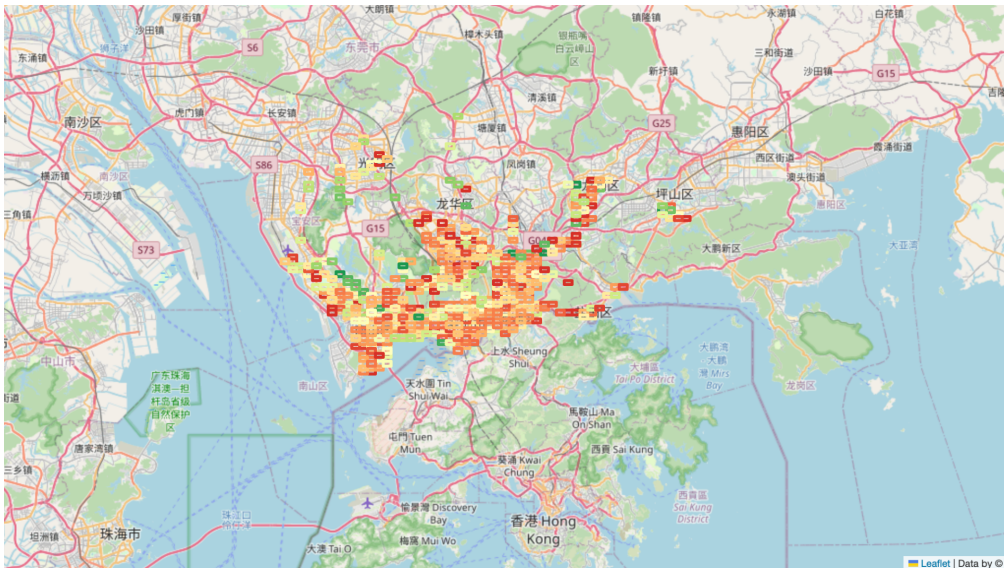

Figure 4.8: The figure shows the data visualization map resulting from a sampling rate of 40% (i.e. all data are retained). The figure results are referred to the traffic situation of taxis moving in the Shenzhen city from 23:30 on 22 October 2014 to 00:00 (midnight) on 23 October 2014.

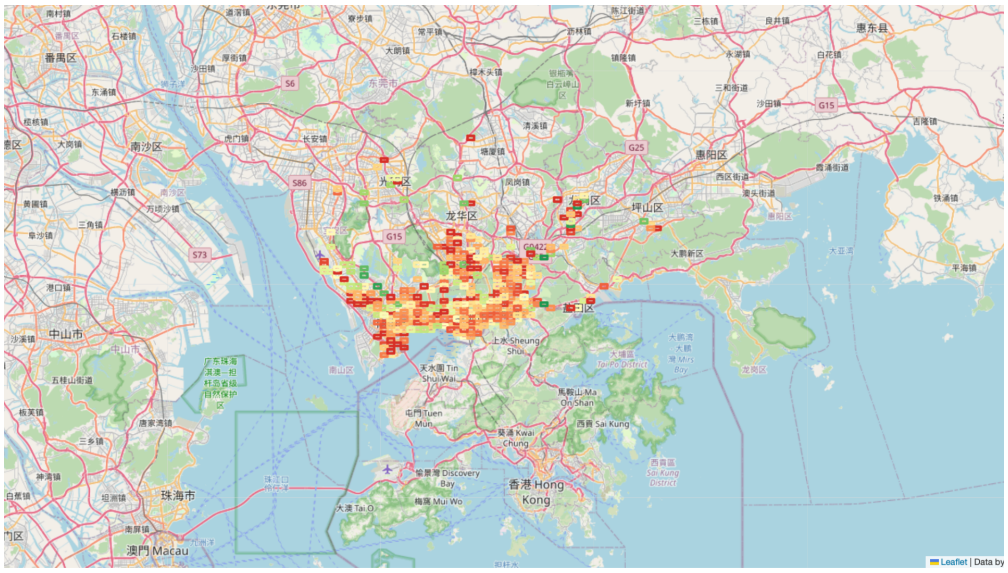

Figure 4.9: The figure shows the data visualization map resulting from a sampling rate of 20% (i.e. all data are retained). The figure results are referred to the traffic situation of taxis moving in the Shenzhen city from 23:30 on 22 October 2014 to 00:00 (midnight) on 23 October 2014.

Quantitative Analysis and Comparison of Geohash-5 and Geohash-6

In the tests we carried out, the performance of stratified sampling for Geohash-5 was evaluated with different sample sizes, namely 20%, 40%, 50%, 60%, 80%, and 100% (baseline), and with sample sizes of 20%, 50%, 80%, and 100% for Geohash-6. Two widely used metrics, namely Mean Absolute Percentage Error (MAPE) and Mean Absolute Error (MAE), were employed to assess the performance of the sampling method. The findings revealed that, for

Geohash-6, and an 80% sample size, the MAPE was observed to be around 10%, indicating a relatively low level of error between the sample and the baseline. On the other hand, when the sample size was reduced to 20%, the MAPE increased significantly to about 38%, suggesting a higher degree of discrepancy between the smaller sample and the baseline.

Geohash-5, on the other hand, have consistently outperformed Geohash-6 in all the runs, showing a minimum MAPE of around 7% with a sample size of 80%, and a maximum MAPE of around 26% with a sample size of 20%.

These findings suggest that, although coarser-grained, the Geohash-5 should be preferred over Geohash-6, at least with the given dataset and for the city of Shenzhen.

As found even in other studies, the Geohash performances are highly dependent on the particular geographical conformation of the analyzed territory and the particular dataset [21], so it is not unlikely that for other datasets and cities the results may be different, and Geohash-5 and Geohash-6 may show opposite trends.

4.2.4 Final Considerations About Edge Nodes Performances

As the primary key factor driving sampling performances was found to be the number of messages to be sampled, and the same factor was also found to be the main factor driving Kafka's performances (with the library used), an important thing to consider, to keep the overall performances as high as possible, is the dimension of the sampling window so that the number of messages to send when the window trigger is not too high. Empirically, it was found that by sizing the window so that around twenty thousand messages were arriving within it was the optimal size.

Another important thing that emerged from our tests was that, although in our implementation the window time is set statically and in milliseconds to wait before re-triggering the processing, in real world scenarios, with variable traffic, unpredictable spikes of traffic, and so on, better choices for window sizing, like dynamically adjusting windows, or incoming messages based windows, may be better choices, that may be worth to test in future works on these topics.

Regarding the accuracy performances of Geohash, on the other hand, the choice of the Geohash precision is an important factor that should be carefully evaluated. The choice should take into account both the particular characteristics of the geographical location of the data, and of the collected data, as supported also by other works[21], but always keeping in mind the granularity constraints that the queries require in order to give meaningful results in the particular scenario.

In the case one cannot find the right balance between granularity of query results, and Geohash based stratified sampling accuracy for its use case, other sampling techniques not analyzed in this work may be evaluated, for example reservoir sampling may be an alternative to stratified sampling.

In conclusion, finding the right balance among computing power requirements, sampling accuracy, and required granularity of results is not an easy task, as changing one parameter may affect the others in a way that is difficult to predict in a consistent way, as it is also highly dependent on the particular case one is considering.

4.3 Spark Performances

Measuring Apache Spark performances is possible in a relatively easy way by using the Spark Web UI. In fact, this UI accessible via web browser offers many different useful metrics about tasks that are being carried out by Spark.

In our scenario, measuring Spark performances was not an easy task to carry out due to the limited size of the dataset, and the fact that Spark can easily handle large amount of data, thus we had a hard time in trying to put it under stress.

Anyhow, the metric that it was chosen to measure was the average duration of a Spark batch (information that can be easily retrieved in the *Structured Streaming* tab of the Spark Web UI) with different sampling percentages. In particular, we carried out tests with sampling percentages set to 20%, 40%, 60%, 80%, and 100%, and always using the stratified sampling technique and the neighborhood-wise sending strategy discussed before (that is, data were distributed to a dedicated topic for each neighborhood).

Although several runs for each sampling percentage were performed, the dataset exhausted within a few batches for each run, and thus overall the data points collected were not much for each sampling percentage.

Moreover, it was decided to exclude from the averages the first batch of each run, as it always took significantly longer to complete with respect to all subsequent batches, but also showed a behavior that was completely independent of the incoming data rate, as confirmed from several "dry-run", that is runs in which no data were sent, and that showed that this batch took about the same time regardless of the fact that data was actually arriving or not. This is likely due to the fact that, in the first batch, Spark "prepares the ground" for the upcoming data.

The results of our tests, showed in Figure 4.10, showed a slight positive correlation between sampling percentage and average batch execution time, but the difference is not as significant as we expected. In fact, the measured difference in performance between analyzing a sample of 20% of the data, as opposed to analyzing the whole dataset, shows a performance increase of just about the 11% in front of a dataset that is 80% smaller in size.

Our guess for a so modest performance increase was that we were not able to adequately stress the Spark node with our dataset, probably also because the computations that were carried out by Spark were relatively simple. Further testing in the future to confirm our hypothesis should be carried out.

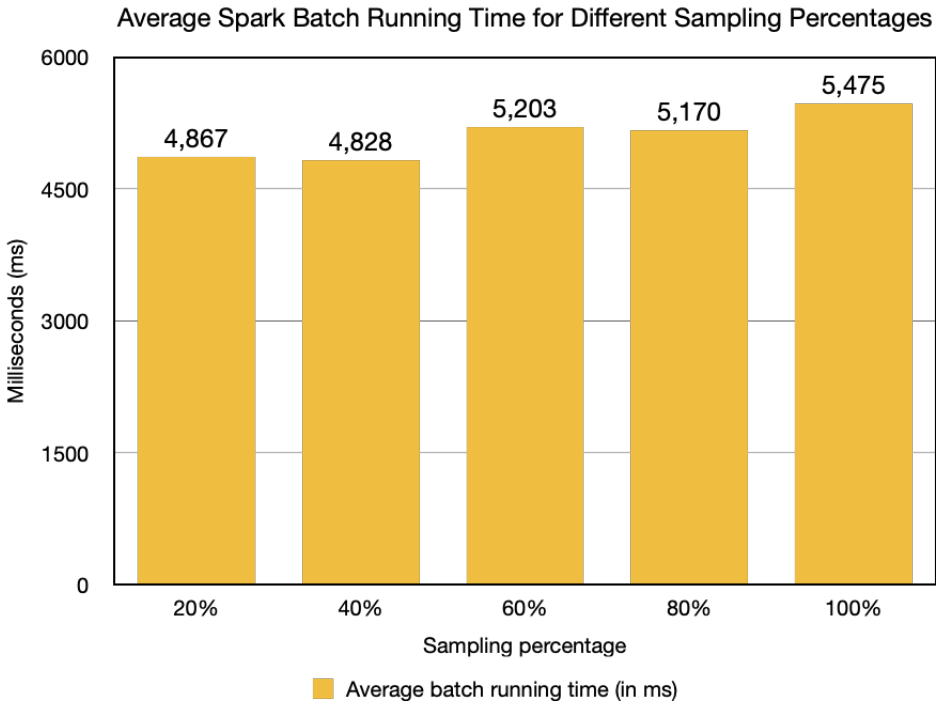

Figure 4.10: The figure shows how the average batch times changed for different sampling percentages in our tests.

Conclusion and Future Developments

The work of this thesis was aimed at evaluating the feasibility and performances of the introduction, in the architecture, of nodes deployed to the edge of the cloud, performing preprocessing of the incoming data, in scenarios in which huge volumes of geospatial data are continuously collected and analyzed. These architectural nodes were found to be very promising as they were capable of offloading the cloud nodes of a considerable part of their duties with limited resource consumption.

The scenario in which we posed ourseleves was that of taxis moving in a city, specifically the city of Shenzhen in China, and continuously sending data about their speed and position to a message queue, and to this particular purpose Apache Kafka was used. The data analysis performed by cloud nodes was a query to found the average speed in different areas of the city, so to locate traffic congestion hotspots.

The work we carried out had the objective of efficiently preprocess data in edge nodes before sending them, in a smart way, to a cluster in the cloud in which Apache Spark was running the heaviest part of the data analysis task.

The smart way we are referring to, is the technique we called neighborhood-aware topic distribution, in which data are not sent all to a single Kafka topic to the nodes in the cloud, but several Kafka topics, one for each neighborhood of the city, are introduced, so that data are sent in a more spatially-aware fashion to cloud nodes.

In particular, what we were able to achieve, was to offload the cloud nodes from the heavy job of locating each message in a specific neighborhood or area of the city performed by using the spatial tessellation technique known as geohashing, and that was used to aggregate data coming from approximately the same area.

Moreover, we wanted to measure the query accuracy loss when data sampling

techniques, such as stratified sampling, were performed by the edge nodes, and we evaluated the loss in accuracy when sampling using Geohash-5 (Geohashes five characters long) and Geohash-6 as stratum selection method.

Geohashing, is a technique that allows to approximate the location of a point in space, at the cost of a quantifiable loss of precision (depending on the Geohash length), but allowing to significantly faster compute if a point belongs to a certain area. This technique is often used in cloud geospatial data processing scenarios, in which the availability (always responding in a timely manner) is often more important than in high precision.

We were particularly interested in data sampling, because in case of sudden spikes in traffic, data sampling helps to avoid overloading nodes downstream in the data processing pipeline. But, to be able to perform effective data sampling, the sample technique chosen must have two fundamental characteristics: it should be fast, otherwise the node performing sampling may itself overload, and so the problem is just moved a step before; and it must not degrade too much the query accuracy, otherwise it is not helping in an effective way.

Our testing shows that, for our particular testing scenario, the loss in accuracy observed by using Geohash-6, was higher than the loss in accuracy observed by using Geohash-5 for strata selection when comparing the query results at different sampling rates. Anyway, research made by other people suggest that the difference in performance is also tightly tied to the particular geographic origin, as well as to the volume, of the incoming data. Thus, the experimental results we obtained should not be considered general, and in different scenarios, independent testing should be carried out when choosing the sizing of this parameter. Moreover, one should always be conscious that different Geohash precision involves different granularity of the query results, thus this is another important factor not to underestimate when making a choice.

Overall, our work demonstrated that it is indeed possible, to offload cloud nodes of simple but costly data preprocessing tasks by using introducing in the general architecture one or more edge nodes, as long as these nodes can perform their tasks with a negligible impact on data throughput.

Both objectives, effectively preprocess data on edge nodes, and not resulting in a bottleneck for performances, may be considered achieved with the work carried out in this thesis. In fact, even if the incoming data rate increase, and start to become overwhelming for the current number of edge nodes deployed, it is sufficient to horizontally scale the deployment, by adding one or more new nodes, to return to a manageable situation.

The scaling of the nodes, although not investigated in this thesis, may be implemented with little to no adjustments to the current implementation of the data preprocessing binary deployed on the edge nodes, and this may be an interesting direction for future works.

As the current implementation of stratified sampling is making use of hash tables having the Geohash as the key, another interesting point that may be investigated in the future, is the comparison between different hashing algorithms for these tables, as the currently used algorithm (SipHash), although very good for the general case, is outperformed, for small hash keys as it is the case of Geohash (which is a few bytes long), by algorithms such as FxHash and FNV-hash.

Finally, Spark performances in our tests showed a small positive correlation between the amount of filtered out data, and the gain in Spark performances, measured as the average time to complete a batch job. Due to the limited size of the dataset at our disposal, we were not able to precisely quantify the gain in performance in longer runs.

Thus, future work could be done towards better quantifying the performance gain derived from data sampling in the runtime of a batch job in Apache Spark, using a larger dataset that can exert greater stress on a Spark cluster. Other metrics may also be introduced, such as the average amount of shuffled data from each of the executor's partitions when using a single Kafka topic to send the data, and when using the technique of sending data to neighborhood-aware topics introduced in this thesis.

To summarize, the work carried out open the way to many different future research scenarios, and while testing focused mainly toward evaluating the feasibility of introducing edge preprocessing nodes in approximate cloud geospatial data analysis scenarios, with positive results obtained in this regard, limitations on the testing dataset posed a limit on the amount of metrics we could collect to investigate the effectiveness of our introduction of a neighborhood-aware (or, more generally, spatial-aware) topics model for data distribution.

Bibliography

[1] NAFI. Gis data. https://firenorth.org.au/nafi3/views/help/Maps_and_Fire_help4.htm, Last visit: 23 April 2023.

[2] Wikipedia. Z-order curve. https://en.wikipedia.org/wiki/Z-order_curve, Last visit: 18 April 2023.

[3] Le Hong Van and Atsuhiro Takasu. An efficient distributed index for geospatial databases. In Qiming Chen, Abdelkader Hameurlain, Farouk Toumani, Roland Wagner, and Hendrik Decker, editors, *Database and Expert Systems Applications*, pages 28–42, Cham, 2015. Springer International Publishing.

[4] Google. Google cloud locations and network. https://cloud.google.com/about/locations#network, Last visit: 22 April 2023.

[5] Hossein Ashtari. What is an edge cloud? https://www.spiceworks.com/tech/cloud/articles/edge-vs-cloud-computing/, Last visit: 22 April 2023.

[6] Ian Buchanan. Containers vs. virtual machines. https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms, Last visit: 28 April 2023.

[7] baeldung. Stratified sampling in machine learning. https://www.baeldung.com/cs/ml-stratified-sampling, Last visit: 25 April 2023.

[8] Andrea Merlin. Docker analisi architetturale. https://amerlin.keantex.com/docker-analisi-architetturale/, Last visit: 1 May 2023.

[9] Apache Software Foundation. Apache kafka. https://kafka.apache.org/, Last visit: 28 April 2023.

[10] Todd Palino Neha Narkhede, Gwen Shapira. *Kafka The Definitive Guide*. O'Reilly, 2017.

[11] Apache Spark. Spark structured streaming. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html, Last visit: 19 April 2023.

[12] Zhenyu Wen, Do Le Quoc, Pramod Bhatotia, Ruichuan Chen, and Myungjin Lee. Approxiot: Approximate analytics for edge computing. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 411–421, 2018.

[13] Isam Mashhour Aljawarneh, Paolo Bellavista, Antonio Corradi, Rebecca Montanari, Luca Foschini, and Andrea Zanotti. Efficient spark-based framework for big geospatial data query processing and analysis. In *2017 IEEE Symposium on Computers and Communications (ISCC)*, pages 851–856, 2017.

[14] G. Cardone. The participact mobile crowd sensing living lab: The testbed for smart cities. *IEEE Communications Magazine*, 52:78–85, 2014.

[15] I Toschi, E Nocerino, F Remondino, A Revolti, G Soria, and S Piffer. Geospatial data processing for 3d city model generation, management and visualization. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 42:527, 2017.

[16] Caitlin Dempsey. Types of gis data explored: Vector and raster. https://www.gislounge.com/geodatabases-explored-vector-and-raster-data/, Last visit: 23 April 2023.

[17] Wikipedia. Geojson. https://en.wikipedia.org/wiki/GeoJSON, Last visit: 23 April 2023.

[18] Wikipedia. Geohash. https://en.wikipedia.org/wiki/Geohash, Last visit: 15 February 2023.

[19] PubNub. What is geohashing? https://www.pubnub.com/guides/what-is-geohashing/, Last visit: 19 April 2023.

[20] Neema Davis, Gaurav Raina, and Krishna Jagannathan. Taxi demand forecasting: A hedge-based tessellation strategy for improved accuracy. *IEEE Transactions on Intelligent Transportation Systems*, 19(11):3686–3697, 2018.

[21] Rik van Outersterp. Partitioning of spatial data in publish-subscribe messaging systems. Master's thesis, University of Twente, 2016.

[22] Wikipedia. Manhattan. `https://en.wikipedia.org/wiki/Manhattan`, Last visit: 23 April 2023.

[23] Wikipedia. Cloud computing. `https://en.wikipedia.org/wiki/Cloud_computing`, Last visit: 23 February 2023.

[24] Wikipedia. Edge computing. `https://en.wikipedia.org/wiki/Edge_computing`, Last visit: 23 February 2023.

[25] Hitachi Vantara. What is an edge cloud? `https://www.hitachivantara.com/en-in/insights/faq/what-is-an-edge-cloud.html`, Last visit: 22 April 2023.

[26] Wikipedia. Virtual machine. `https://en.wikipedia.org/wiki/Virtual_machine`, Last visit: 21 February 2023.

[27] Wikipedia. Containerization. `https://en.wikipedia.org/wiki/Containerization_(computing)`, Last visit: 28 April 2023.

[28] Ilias Mavridis and Helen Karatza. Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. *Future Generation Computer Systems*, 94:674–696, 2019.

[29] Wikipedia. Message-oriented middleware. `https://en.wikipedia.org/wiki/Message-oriented_middleware`, Last visit: 28 April 2023.

[30] Gregory F. Pfister. *In Search of Clusters*. Prentice Hall PTR, 1998.

[31] Wikipedia. Computer cluster. `https://en.wikipedia.org/wiki/Computer_cluster`, Last visit: 24 April 2023.

[32] Google. About data sampling. `https://support.google.com/analytics/answer/2637192`, Last visit: 23 April 2023.

[33] Jeffrey S. Rosenthal Michael Evans. *Probability and statistics: the science of uncertainty*. W.H. Freeman and Co, 2010.

[34] Wikipedia. Statistical population. `https://en.wikipedia.org/wiki/Statistical_population`, Last visit: 25 April 2023.

[35] Wikipedia. Sampling (statistics). `https://en.wikipedia.org/wiki/Sampling_(statistics)`, Last visit: 25 April 2023.

[36] Wikipedia. Simple random sampling. `https://en.wikipedia.org/wiki/Simple_random_sample`, Last visit: 25 April 2023.

[37] Wikipedia. Stratified sampling. https://en.wikipedia.org/wiki/Stratified_sampling, Last visit: 25 April 2023.

[38] Wikipedia. Python. https://en.wikipedia.org/wiki/Python_(programming_language), Last visit: 15 February 2023.

[39] GeoRust. Georust. https://github.com/georust.

[40] Rust Team. Rust. a language empowering everyone to build reliable and efficient software. https://www.rust-lang.org/learn, Last visit: 2 May 2023.

[41] Wikipedia. Apache kafka. https://en.wikipedia.org/wiki/Apache_Kafka, Last visit: 28 April 2023.

[42] Tanvir Ahmed. Kafka's origin story at linkedin. https://www.linkedin.com/pulse/kafkas-origin-story-linkedin-tanvir-ahmed/, Last visit: 28 April 2023.

[43] Wikipedia. Apache spark. https://en.wikipedia.org/wiki/Apache_Spark, Last visit: 28 April 2023.

[44] Apache Software Foundation. Spark documentation. https://spark.apache.org/docs/latest/, Last visit: 7 May 2023.

[45] Radek Ostrowski. Spark documentation. https://www.toptal.com/spark/introduction-to-apache-spark, Last visit: 7 May 2023.

[46] Bitnami. Apache zookeeper packaged by bitnami. https://hub.docker.com/r/bitnami/zookeeper, Last visit: 1 May 2023.

[47] Bitnami. Apache kafka packaged by bitnami. https://hub.docker.com/r/bitnami/kafka, Last visit: 1 May 2023.

[48] Bitnami. Apache spark packaged by bitnami. https://hub.docker.com/r/bitnami/spark, Last visit: 1 May 2023.

[49] João Pedro. A fast look at spark structured streaming + kafka. https://medium.com/towards-data-science/a-fast-look-at-spark-structured-streaming-kafka-f0ff64107325, Last visit: 4 May 2023.

[50] crates.io. Kafka rust client. https://crates.io/crates/kafka, Last visit: 8 May 2023.

[51] crates.io. csv. https://crates.io/crates/csv, Last visit: 5 May 2023.

[52] Wikipedia. Point in polygon. `https://en.wikipedia.org/wiki/Point_in_polygon`, Last visit: 7 May 2023.

[53] Horace Williams. Geospatial utility belt. `https://github.com/worace/geoq`, Last visit: 7 May 2023.

[54] crates.io. rayon. `https://crates.io/crates/rayon`, Last visit: 7 May 2023.

[55] Wikipedia. Flask (web framework). `https://en.wikipedia.org/wiki/Flask_(web_framework)`, Last visit: 23 April 2023.

[56] PyPI. folium. `https://pypi.org/project/folium/`, Last visit: 6 May 2023.

[57] pandas. pandas. `https://pandas.pydata.org/`, Last visit: 6 May 2023.

[58] crates.io. rdkafka. `https://crates.io/crates/rdkafka`, Last visit: 8 May 2023.

[59] Aria Beingessner. Benchmarks of various hashers. `https://github.com/Gankra/hash-rs`, Last visit: 11 May 2023.

[60] crates.io. fxhash. `https://crates.io/crates/fxhash`, Last visit: 11 May 2023.