ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

MobilityDB on Google Cloud Platform

Thesis submitted for the award of the degree of
Master of science in computer science and Engineering with focus Professional

Sid-Ahmed BOUZOUIDJA

Supervisor
Professor Esteban ZIMANYI

Service
CoDE

Academic year
2022 - 2023

Abstract

Nowadays, data is generated exponentially due to the newest technologies including IoT, GPS and 5G, etc. In addition, most of the current projects are data driven projects that aim to help organizations making data driven decisions, identifying patterns, trends, correlations and forecasting the future events. This presents numerous challenges concerning data storage and processing, challenges that are crucial for effectively harmonizing the latest technologies with data consumption needs. Traditional ways to store and analyze data is not sufficient to provide high performance and tackle bottleneck issues like extended response time when executing analytical queries on large datasets. Cloud computing is potentially the most effective solution, offering the ability to enhance the performance and cost-efficiency through the scalability and the elasticity.

This thesis focus on the analysis of spatial and spatial-temporal data. Moreover spatial and spatial-temporal queries are complex and heavy to manipulate it on SQL-like expressions. The MobilityDB extension for the PostgreSQL engine plays a pivotal role in this context. It simplifies the manipulation of spatial and spatial-temporal data, facilitates the execution of straightforward analytical queries by providing a set of predefined spatial-temporal functions. In this work, our focus aim in the implementation of a cloud native environment for MobilityDB. This environment is designed to facilitate the distribution of both data and the processing of analytical queries across a cluster. It can be deployed on several cloud providers using Kubernetes system on cloud platforms such as AWS, Azure and GCP with few adjustment in the configuration.

The primary objective of this work is integrate MobilityDB within GCP environment. Additionally, a key focus is on identifying the shared functionalities across all three major cloud providers, in order to ensures that in the case of migrating the MobilityDB instance, re-implementing new functionalities is minimized. Exploring the similarities and differences between the potential approaches, allow us to define the foundation for distributing spatial and spatial-temporal data management on top of MobilityDB extension in the cloud. The Citus extension for PostgreSQL facilitates efficient partitioning and distribution of large tables across a cluster. This capability is further combined with MobilityDB to enable both scaling of storage and distribution of workloads within the cluster environment.

To comprehensively assess and evaluate our solution, a series of experiments have been conducted. We utilized pre-generated trajectories data within Brussels city using the benchmark tool called BerlinMOD. Additionally, we further validated our solution using real-world data provided from the danish maritime vessels tracking with Automatic Identification System (AIS). The experimental results demonstrate that performing large-scale trajectory analysis using distributed environment across multiple PostgreSQL servers outperforms the analysis conducted on a single PostgreSQL server.

`Keywords:` Distributed databases, moving object databases, Google Cloud Platform (GCP), Kubernetes, PostgreSQL server, MobilityDB.

Contents

List of Figures

List of Tables

Acronyms

AET Average Execution Time

AKS Amazon Kubernetes Service

AWS Amazon Web Service

CPU Computer Processing Unit

CTE Common Table Expression

DMS Database Migration Service

EKS Elastic Kubernetes Service

GCP Google Cloud Platform

GKE Google Kubernetes Engine

IaaS Infrastructure as a Service

MIG Managed Instance Group

OSM Open Street Map

PaaS Platform as a Service

PI Percentage Improvement

SaaS Software as a Service

SQL Structured Query Language

SSD Solid State Drive

VM Virtual Machine

YAML Yet Another Markup Language

Chapter 1

Introduction

1.1 Motivation

Trajectory data processing has become more important in recent times, as it plays a vital role in society. Several areas are concerned, particularly in transportation management which allows us to understand the movements of vehicles in order to proceed with trip planning, anomalies detection and route optimization. Even more, it helps to predict the future movements by comparing historical trajectories and allows to visualize movement patterns using spatial-temporal data. Moving object data are collected through advanced technologies like sensors, GPS and WiFi before being stored and processed, where each movement is characterized by position recorded within timestamp.

In General, trajectories data is collected continuously with a collection frequency that varies depending on the type of application. A high frequency implies a large amount of positions recorded in a tiny interval of time. This will directly impact storage, processing and analysis. In addition, executing analytical queries on large trajectory data adds more complexity in terms of response time. There are several technologies to stores, manages and querying data including relational database management systems that are mostly used by organizations through it robustness for many years. Furthermore, manipulating and interacting with the RDBMS database is easier with the standardized and universal SQL language. It allows facilities for writing queries, particularly for the analytical applications. Another potential solution is to use NoSQL technologies which have gained confidence by the organizations over the last few years. Among the benefits of using NoSQL technologies is the storage flexibility that supports unstructured, semi-structured and structured data scalability in distributed environment. Moreover, by using NoSQL technologies, schema are much reduced in terms of design and model. Figure 1.1 we show a ranking of the top ten search engines[1] for databases that brings together SQL and NoSQL technologies. This ranking is updated monthly. The ranking is calculated on several criteria, we note the number of times the database engine is searched on the Google engine for example, the number of relative questions and the number of users interested in the database engine. Also the relevancy in social networks where the number of tweets concerning the database engine is calculated. Figure 1.1 show us the four places that are occupied by the relational database engine of which Oracle and Microsoft SQL server are proprietary while MySQL and PostgreSQL are open source. Compared to

[1] `https://db-engines.com/en/ranking`

[2] DB-Engine ranking criteria :`https://db-engines.com/en/ranking_definition`

Rank			DBMS	Database Model	Score		
Jul 2023	Jun 2023	Jul 2022			Jul 2023	Jun 2023	Jul 2022
1.	1.	1.	Oracle	Relational, Multi-model	1256.01	+24.54	-24.28
2.	2.	2.	MySQL	Relational, Multi-model	1150.35	-13.59	-44.53
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	921.60	-8.47	-20.53
4.	4.	4.	PostgreSQL	Relational, Multi-model	617.83	+5.01	+1.96
5.	5.	5.	MongoDB	Document, Multi-model	435.49	+10.13	-37.49
6.	6.	6.	Redis	Key-value, Multi-model	163.76	-3.59	-9.86
7.	7.	7.	IBM Db2	Relational, Multi-model	139.81	-5.07	-21.40
8.	8.	8.	Elasticsearch	Search engine, Multi-model	139.59	-4.16	-14.74
9.	9.	9.	Microsoft Access	Relational	130.72	-3.73	-14.37
10.	10.	10.	SQLite	Relational	130.20	-1.02	-6.48

Figure 1.1: Top 10 of ranking the database engines including RDMS and NoSQL technologies[2]

NoSQL technologies, we have MongoDB which takes the fifth place. The choice among these database engines is not an easy task, it depends in particular on the nature of the data (structured or unstructured), extensive feature, the frequency of data collection as well as the allocated budget, etc.
In our work we use PostgreSQL as RDBMS for our data, we made this choice for several reasons. A fundamental reason in which we opt for this database engine is to complete the work that has been done in the past years which deals with our subject, moving object data and their trajectories through the extension MobilityDB[3] which is supported only by PostgreSQL engine, it act like a toolbox to better store, manage and analyze spatial and spatial-temporal data in a simple and efficient way. This leads us to complete the past effort which is expanding the integration of MobilityDB on other platforms as well as to increase the performance of applications that deal with the analysis of the movements and trajectories of objects. In addition, PostgreSQL is known for its rich functionality which supports spatial data, in particular via the PostGIS[4] extension which is adopted by many organizations.

Before going into deeply in our motivation for choosing this subject, let's return to our problem concerning the complexity of analytical queries. The query execution time depends entirely on the amount of stored data. There are solutions to optimize queries such as adding indexes for the columns that are used in the WHERE and JOIN clause of the SQL query, redesigning the database schema could also be a solution and the use of partitioning techniques and hash functions could reduce the execution time. These solutions remain limited beyond a certain amount of data. Moreover, a single PostgreSQL server is not enough to maintain the performance for a database that grows rapidly over time. An ideal solution to overcome this problem is to keep the response time of an analytical query acceptable even if the data increases over time. The distribution of data but also the execution of analytical queries within a scalable environment is a potential solution that could overcome this problem. Distributing the data and the queries over a cluster of machines for moving objects data is considered as the main objective of our thesis that could drives MobilityDB in an evolutionary scale. The distributive environment is offered by several providers, including the cloud service AWS, Azure and GCP. Indeed MobilityDB is already integrated on the AWS and Azure platform, the purpose of this memory is to continue the effort to make MobilityDB adapted in Google Cloud Platform

[3] https://github.com/MobilityDB
[4] https://postgis.net

GCP and to implement a native cloud solution which includes the common functionalities between the three cloud providers which is based on an extraction of similarities between the implementation of the three platforms (AWS, Azure, GCP).

Cloud providers have become increasingly in demand over the past few years. Organizations of all sizes in various fields have migrated their data, activities and workloads within the cloud. Especially since the COVID-19 pandemic. Because of several benefits like flexibility, scalability, security and cost effective which is calculated in case of using resources (CPU and memory) only. Figure 1.2 shows a statistical study that describes the adoption rate of cloud services over time. As we can see in Figure 1.2 that the migration

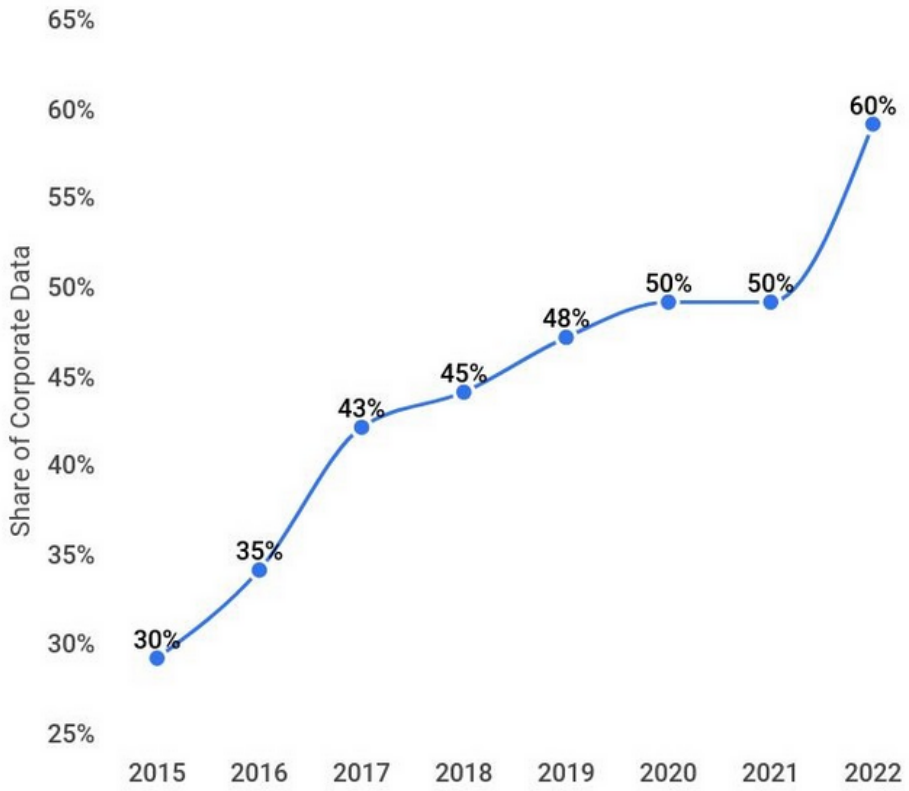

Figure 1.2: Cloud services adoption over time[5]

to cloud services has doubled in the last seven years. This will increase even more in the years to come as the rate of data generation increases more, due to the advanced technologies for data collection, storage and analysis. In [35], chapter migration to cloud, the various reasons and factors that companies rely on to enhance their business through the adoption of cloud computing. In this work our use case for Cloud service is Google Cloud Platform (GCP). This mean our implementation will be experimented in GCP. In the other hand, our database use case will be dedicated for spatial and spatial-temporal databases handled by MobilityDB functions.

1.2 Problem Overview and Challenges

In this part we will give a detailed overview of our problem as well as the challenges encountered. As mentioned above, our job is to integrate the MobilityDB extension for the PostgreSQL server into the GCP in order to distribute data and analytical queries in

[5] `https://www.zippia.com/advice/cloud-adoption-statistics/`

a scalable manner. This work concerns and targets applications that process geo-spatial and spatial-temporal data of moving objects such as vehicles, vessels or even individuals.

The main challenge encountered in this work is the development of a common cloud-native environment for MobilityDB that can operate across cloud provider environment, whether in AWS, Azure or GCP. This implementation is much more complex to unify between the three providers because the APIs and the products offered in each cloud service differ from one environment to another. kubernetes system is a potential approach to unify those features in order to make MobilityDB operate in the three cloud provider.

The second challenge consist to achieving self-scalability using the existing scaling tool like horizontal Pod autoscaler of Kubernetes. A self-scalable system has the capability to autonomously allocate resources when it detects diminished performance for specific complex analytical queries. For instance, it can dynamically increase resources to ensure stable performance in such scenarios. After testing a distributed moving object database in the Kubernetes autoscaler, this is not feasible for the following reasons. Since we will extend the past work that use Citus extension to partition and distributing of moving object tables across a cluster, we will use Citus in our native-cloud environment. The explanation reside that the autoscaler add new worker node by replicating the state of the node that produce bottleneck without considering the shards partition mechanism. As a result, we cannot take advantage from the autoscaler tool. The correct way of scaling when a new node is added is to redistribute the partitions across the cluster in order to send shards to this node.

In the chapter distributed database management on GCP, we will show how to ensure the redistribution of partitions in Citus cluster via an API which allows the user to interact with the coordinator node in order to launch the scale-in or scale-out operation which will be explain in detail latter.

1.3 Objectives

The main objective of this work is to create and deploy a cloud-native version of PostgreSQL MobilityDB extension tailored for compatibility with the Google Cloud Platform (GCP). This work allows us to fully or partially solve another objective which is the factorization of the necessary prerequisites features which are common between the implementation of MobilityDB in the AWS, Azure and GCP platform. The goal of this consolidation effort is to architect a cloud-native MobilityDB solution that boasts portability and is capable of functioning across the three major cloud platforms AWS, Azure, and GCP with few interactions by the cloud administrator.

One important thing to ensure in the case of a scale-in is the integrity of the data. As we have seen previously, the Citus extension of PostgreSQL which makes it possible to distribute the storage over several worker nodes in order to increase the frequency of computation by adding CPUs and memories during the execution of a complex analytical query for a large database. Adding a node to an operational cluster requires a new calculation of the hash table in order to generate new partitions for this node. The scale out operation does not cause a problem since the data in the operational cluster will not be impacted. On the other hand, a scale-in operation causes a deletion of a worker node, this means all the partitions in the desired node to be deleted, will be erased. In order to avoid losing these partitions, we must drain this node in order to send those partitions to

the other nodes before delete the node.
Automation is a crucial element during the initialization of the distributed environment through the Citus cluster inside the Kubernetes cluster provided by the GCP platform. Specifically, the automation concerns two critical aspects: the scale-out and scale-in operations, along with the redistribution of partitions that follows alterations in the cluster size. These processes must be initiated automatically to ensure seamless and efficient management within the cloud environment. Another worthy objective, which is considered as a best practice for users, involves furnishing them with a comprehensive manual. This manual will describes all the necessary steps to successfully deploy MobilityDB within the Google Cloud Platform (GCP). This resource aims to empower users by offering clear and detailed guidance throughout the deployment process.

1.4 Contribution and Thesis Outline

After having detailed the problem of our thesis, in this section we describe our contribution in order to ensure and response to the aforementioned objectives. At the end we give an overview of the outline on the structure of this thesis. Before going further in this section, we want to synthesize in which field or area, our contribution took place. Our contribution evokes practically four main IT topics which are cloud computing, databases, systems and optimization. In each of these subjects, we use specific technological fields which consist in providing a database as a service in cloud computing services, in order to better store, manipulate and analyze spatial and spatial-temporal data through a distributed system, which processes operations in a distributed manner. Regarding the system subject, we use micro service technologies to facilitate the integration, deployment and updating our solution through Docker technology, Kubernetes systems and APIs. The following points detail our contribution in the four areas mentioned above.

- Database as a service in the cloud is a service that provides a database engine without any installation by the user in order to store, manipulate and analyze the desired data in an environment remotely from the workspace of the users. The storage media and calculation units are the responsibility of the cloud provider, in which the user only pays for the time elapsed of his work on these resources provided. Database as a service is an element that we have provided through our work to support moving object databases. This, will be implemented and tested in the Google Cloud platform, more precisely we used the Google Kubernetes Engine (GKE) product which allows us to ensure scalability and distributivity.
- Moving object databases are evolving exponentially, the storage and processing of these data are very complex in terms of data structure such as points, polygons. In addition, the queries to analyze this data are very expensive in terms of CPU and memory, which are linked to operations nested in an analytical SQL query as well as join operations, aggregation and casting functions. Our contribution at this level is to provide a distributed representation for this type of data through the MobilityDB and Citus extension for the PostgreSQL server. A unification of these two extensions has been done in order to have a cloud-native solution with a distributive environment that is specific to support spatial or spatial-temporal databases. A docker container image has been implemented to facilitates the deployment of our solution in cloud service platforms.
- Optimization is crucial for our solution to gain performance. Our contribution concerns three aspects of optimization which are the distribution of the data, the

parallelism of the processing for the analytical queries and the automation during the initialization of the cluster or even during the redistribution of the partitions if we have an addition or deletion of nodes in the cluster. So an alignment is necessary between the evolution of moving object data with the storage and processing resources (CPU, memory). A scalable and distributive environment is designed to allow the allocation or release of nodes within the cluster through commands that have been developed in an API. We have automated several tasks that are repeated during interactions with the pre-deployed cluster in order to allow the user to concentrate on other aspects of the application.

- The Kubernetes system and the Docker engine are among the most used technologies in recent times to ensure maintainability, isolation of the infrastructure and software part. This contributes to have a micro services solution. We have provided a docker image which includes all the necessary prerequisites of our native cloud environment, in particular it includes the PostgreSQL server with the MobilityDB extension and the Citus data extension. In addition we have defined a series of object declarations for the Kubernetes system which facilitates the deployment and installation of our cloud native solution.
- In addition to these technological contributions mentioned above, we have provided a comparative study through the state of the art part that explores the different approaches and systems that could be useful to be adapted for our application, and meets the specifications and objectives. In addition, a tutorial has been prepared for new users who wish to use MobilityDB in the GCP service. This tutorial explains all the steps needed to implement a database as a service in GCP using the Google Kubernetes Engine GKE product. By adding good usage practices in particular, to better take advantage of the scalable and distributed system by choosing the correct method of partitioning tables via Citus for example.

Below is the detail of our thesis structure following this introductory section.

Chapter 2 is devoted to the state of the art of our thesis, which we introduce the different technological possibilities in order to provide a database as a service in the cloud. We start by introducing the different public cloud providers including AWS, Azure and GCP as well as the products and features that they offer in order to know the similarities between them. Then we approach the existing cloud solutions that support moving object databases under the title distributed moving object database on the cloud in order to make a comparison between the different approaches and systems. Then we introduce a benchmark in which the moving object database is tested on the Kubernetes system in a distributed environment. This reveals the advantages and disadvantages when using Kubernetes systems. At the end, we show another approach that ensures the distribution through the Hadoop system, and highlighting the different researches that use such a system to gain workload performance.

Chapter 3 is dedicated to the different technological systems and products that are used in our thesis work. Starting by introducing the services and products of Google Cloud Platform used in our solution, going through the three ways to supply resources in the cloud, in particular software as a service SaaS, platform as a service PaaS and infrastructure as a service PaaS. Then we introduce the Kubernetes GKE system from GCP which is the product used in our solution to host our distributed environment. Thereafter we introduce the PostgreSQL server with the two main extensions needed, MobilityDB and Citus in depth. Finally we talk about the Python

language used in our API as well as the different libraries that are used to interrogate the GCP client APIs, in order to initialize and manipulate the GKE cluster. The visualization tool Dash Plotly used in our thesis work to generate the graphs of our experiments is also introduced at the end of this chapter.

Chapter 4 aim to introduce the development of our native cloud solution which consists in distributing a database in the GKE system of the GCP platform. This part composes the heart of our contribution which consists initially of the initialization and configuration of the necessary infrastructure in terms of resources and type of machine in order to instantiate a GKE cluster in the GCP platform. Then we introduce the partitioning and distribution approach through the initialization and implementation of the Citus cluster on top of GKE cluster in order to setting up our scalable and distributed environment. At the end we explain the implementation of the two main operations scale-in and scale-out and their own role, in order to facilitate the task for the user to increase or decrease the size of the cluster while ensuring the integrity of the database.

Chapter 5 is devoted to experiences and performance evaluation when our cloud-native solution is implemented for an analytical application that processes and analyzes moving object data as a use case. In order to better evaluate our solution, two experiments are made. The first experience is to use the BerlinMOD benchmark model which consists of generating data by the generator with a desired amount of data, in order to launch a series of pre-defined SQL queries where each query has its own behavior in terms of number join, complexity and aggregation functions. The second experience consists in testing our solution with real data, concerning the AIS data provided by the Danish Maritime Authority. This allows us to confront our solution in a real environment with constraints that cannot be taken into account by the data generator of the first experiment. At the end of each experiment, we discuss the results in order to draw conclusions on the performance gained by using our native scalable and distributive cloud solution based on the evolution of the data over time.

Chapter 6 is devoted to the discussion and conclusion after the completion of this work, we also evoke the difficult points that we have confronted during this work as well as the future prospects that we wish to improve or other areas that we can explore.

Chapter 2

State of the Art

In this chapter, we will discuss the state of the art of our thesis which consists in exploring different possible approaches that allow us to meet our objectives mentioned before, as well as to know the different systems and technologies that could support the requirements of our objectives. Moreover we approach the related works which are already carried out in the past that involve the same kind of our problem partially or entirely. In particular, making distributed moving object databases in an evolving environment which requires an alignment between the evolution of data over time and the evolution of resources (storage space, CPU and memory). We will start by exploring the public cloud platforms services, AWS, Azure and GCP by explore their services and products offered that can support our desired solution, as well as determine the similar services offered by these three cloud in order to factorizing the common functionalities between them, to create the foundation of our native cloud solution that can operate on all three platforms. Since our solution must be executed in the PostgreSQL database server, we analyze for for the three cloud providers, its PostgreSQL product as a service to ensure that our solution works in the three platforms. Thereafter we explore the Kubernetes system which could also provide a database as a service in PostgreSQL with their possible advantages in terms of scalability and extensibility. Through this comparative study, by exploring these possibilities, aids us in making an informed decision to design our system that potentially meets our requirements posed by our identified problem. To recall the main requirement, our work consists in deploying a database as a service in the cloud which supports a database of spatial and spatial-temporal nature, which will be manipulated by the MobilityDB extension for the PostgreSQL server in an evolutionary way and distributed through the Citus extension. And as a goal is to guarantee high performance, especially when executing complex analytical queries.

2.1 Database as a Service

Database as a Service (DBaaS) allows us to provide a database operator instance without focusing on the software and physical part of it. It allows us to store, maintain, scale and upgrade our application databases in the background through the cloud provider in an automatic way, without making any effort as well as resource provisioning is done automatically. There are many cloud providers, we will introduce the three well-known providers, GCP, AWS and Azure. A literature review [7] is carried out to identify both the strengths and weaknesses of databases commonly utilized in cloud platforms and to assess the obstacles associated with the development of cloud databases. In addition to

this, we present another approach that could offer a database as a service through a group of virtual machines as a service managed either by the cloud service provider or by the user. Another crucial system is to use a docker container orchestrated by the Kubernetes system which makes the database as a service scales automatically through auto scaling horizontally by adding other machines within the cluster, or vertically by adding more computation power such as CPU cores and memory units. In the following sections, we will present the PostgreSQL database as a service for each possible approaches mentioned above.

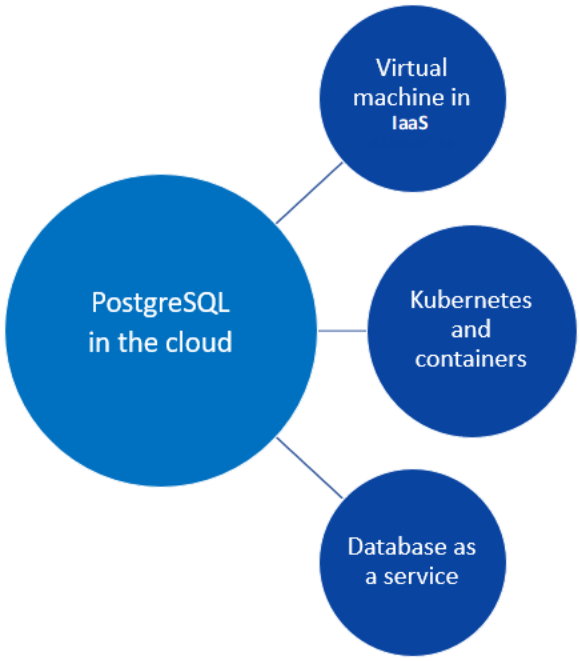

Figure 2.1: Possible systems to deploy PostgreSQL database as a service

Figure 2.1 show the possible system that could provides PostgreSQL database as a service, including Kubernetes system, virtual machine and database as a service fully managed by the cloud provider.

2.1.1 Public Cloud Provider for PostgreSQL

The public cloud is a computing platform where resources are made available over the Internet by a third-party provider, these resources are shared by organizations and individuals who wish to boost their applications and solutions. There are many public cloud providers around the world. The well-known public cloud providers are, AWS, Azure, and GCP. As a cloud user, we can provision storage and computation power by only paying for our usage in terms of the number of operations performed and the amount of data hosted. PostgreSQL it can be deployed on any public cloud provider. In the following subsection, we briefly describe the PostgreSQL service that is integrated in each of the popular cloud providers in order to extract common features between them.

Google Cloud SQL for PostgreSQL

Cloud SQL for PostgreSQL is a fully managed service by offered by GCP provider[1] that enable to manages, configures, and maintains PostgreSQL databases. Cloud SQL offers

[1] `https://cloud.google.com/sql/docs/postgres/features`

several advantages over the standard PostgreSQL, including:

- The possibility of using PostgreSQL in multi zone or multi regions, the localities covered are America, Asia, Europe and Australia.
- Database migration from multiple sources to cloud SQL is supported through the Database Migration Service DMS.
- The data is encrypted by Google's internal network which ensures high security.
- Database replication is guaranteed between regions and zones.
- Backups are made available automatically and restoration is possible from the various backup points.
- Logging and monitoring operations are integrated with the SQL cloud service, this allows the visibility for all activities within the PostgreSQL server, as well as observability of the resources used during the execution of the SQL queries.

Among the disadvantages of cloud SQL is the following:

- Cloud SQL is not compatible with all the standard PostgreSQL extensions, this is a major inconvenience for us because the MobilityDB extension which allows to manipulate spatial and spatial-temporal data is not taken into account in the list of compatible extensions. Also for the Citus extension which allows partitioning and distribution of tables is not yet integrated.
- PostgreSQL's psql client does not support operations that require a reconnecting, such as connecting to another database instance via the \c command.

Cloud Spanner is another product that offers PostgreSQL as a service which is fully managed by GCP as well. Cloud Spanner is considered an infrastructure as a service IaaS that provides scalability at any scale. In other words, it increases hardware resources automatically in the case of a heavy load on the server, without user's interaction. Cloud Spanner is the ideal service for applications that process big data. Despite its robustness, cloud Spanner does not support custom PostgreSQL extensions same as cloud SQL service. A positive point offered by the GCP service is the possibility of opening a request for the development of new functionality which will be supported by the GCP team, after the study of the feasibility and especially the impact and the outcome of this functionality. We opened a request for the integration of the MobilityDB extension in cloud SQL and cloud Spanner in order to use PostgreSQL managed by GCP.

Azure for PostgreSQL

The Azure cloud provider also offers PostgreSQL[2] as a service with functionality similar to GCP in terms of provisioning automatically without going through the phase of deploying, configuring and adjusting resources manually in the case of a heavy workload. PostgreSQL from Azure allows the scalability in an intelligent way which is a remarkable point that allows to reduce the cost of resources material and an efficient use of the resources according to the workload's request. The smart performance feature automatically tunes database performance based on special metrics. Compared to the advantages of using PostgreSQL from Azure, it is almost the same as offered by the GCP counterpart in terms of security, flexibility, fully manageability by the platform and high availability. Regarding the MobilityDB extension, it remains not integrated in the list of Azure

[2] `https://azure.microsoft.com/en-us/products/postgresql#overview`

PostgreSQL extensions. On the other hand, the Citus extension is already integrated with PostgreSQL from Azure,[3] which makes it possible to bring PostgreSQL to any scale. This latest integration of Citus with PostgreSQL from Azure guarantees multi-tenancy application and allow a centralized database architecture in several localities with high level of scale. Figure 2.2 show the advantages when using Azure for PostgreSQL.

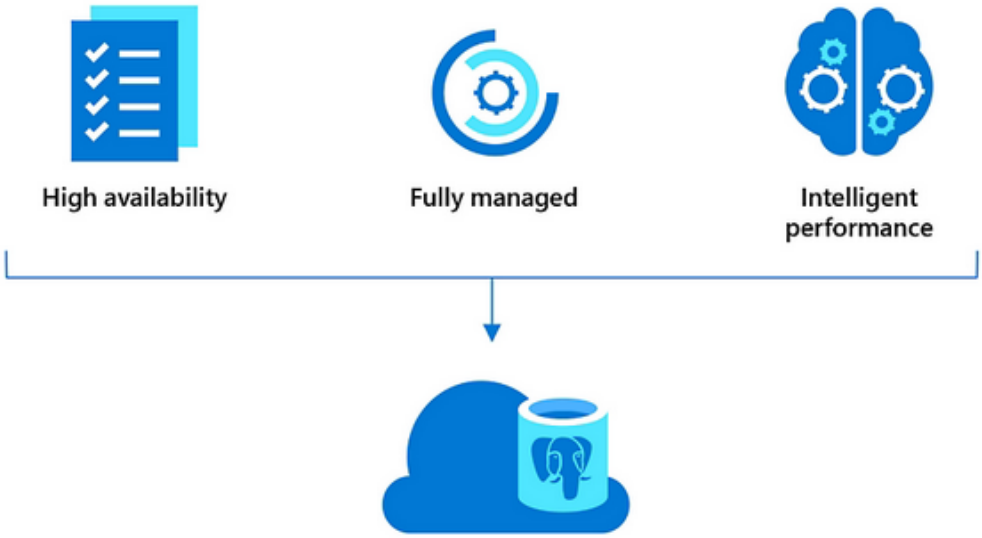

Figure 2.2: Azure for PostgreSQL[4]

Amazon RDS for PostgreSQL

Like GCP and Azure, Relational Database Service (RDS)[5] by Amazon offers a PostgreSQL as a service fully managed by AWS that makes it easy to set up, operates, and scales deployments at lower cost and with resizable hardware capacity. Amazon RDS handles complex and time-consuming administrative tasks such as PostgreSQL software installation and upgrades, storage management, replication for high availability and read throughput, and backups for disaster recovery. As discussed before, GCP and Azure do not support custom extensions for PostgreSQL server. Same thing for AWS, MobilityDB and Citus extensions are not supported by RDS.

EnterpriseDB for PostgreSQL

There are other cloud platforms that offer PostgreSQL as a service, including EnterpriseDB, which offers the PostgreSQL in several ways of implementation depending on the customer's needs and the scenarios of the desired application. The main activity of EnterpriseDB[6] is the contribution and the enhancement of the PostgreSQL engine in order to bring this tool beyond its standard utilisation as a single server. They build new features upon PostgreSQL that support and cover a wide range of business requirements and deployments. Among EnterpriseDB's main contributions to PostgreSQL is:

[3] `https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/service/postgresql`

[4] `https://azure.microsoft.com/en-us/products/postgresql/`

[5] `https://aws.amazon.com/rds/postgresql/`

[6] `https://www.enterprisedb.com/products/biganimal-cloud-postgresql`

- Add advanced features within the server to increase the SQL capability of the engine, make it compatible with a variety of systems and environments and implements custom extensions.
- Make the server suitable within a distributed environment to ensure high availability, centralized between several localities of the application.
- PostgreSQL as a Kubernetes operator: this allows the PostgreSQL server to be a cloud-native solution that can work in cloud platforms, particularly in IBMPower and OpenShift. This is achieved through the developed and implemented APIs that are integrated into the Kubernetes system to add a list of features for PostgreSQL that are not implemented in the Kubernetes API.

Despite these advanced implementations, EnterpriseDB's PostgreSQL remains limited since it is managed entirely by the EnterpriseDB team in order to accommodate the requirements of our objectives, in particular the addition of the custom MobilityDB extension. As well as the distributed environment provided by EnterpriseDB is not suitable for our scenario which is the partitioning of tables into several shards in order to distribute them within the cluster, to guarantee the parallelism of the execution of SQL queries. Unlike the classic way of distribution developed by EnterpriseDB which is the increase of data availability by replicating database in several nodes.

2.1.2 Instance Group of Virtual Machines for PostgreSQL

An instance group is a group of virtual machines that work together as a single unit. It is categorized as IaaS. This type of system could be another approach that serves PostgreSQL as a service. There are two possible ways to configure PostgreSQL in a group of virtual machines. The first approach is to create and manage a group of virtual machines by ourselves as described in [17]. This means that we have to deploy the PostgreSQL on each of the virtual machines and configure the connectivity between them. Using this method is time consuming and later we lose efficiency as well as the maintainability of the infrastructure becomes more complex over time, moreover performing upgrades manually requires an effort by administrators database. Certainly this method provides a high level of flexibility in terms of customization of extensions and functionalities which could satisfy our requirements for our native cloud solution. The other approach is to use the managed instance group service (MIG). Such a service is offered only by the GCP platform among the three cloud providers. The advantages of the managed group of instances are the fully-managed of the hardware and software workload, the scalability and the maintainability is ensured as well as the high availability. Figure 2.3 lists all the features provided, using a VM group managed by GCP, depending on different type of workload including stateless, batch and StateFulset workload.

High Availability

Managed Instance Group (MIG) ensures database high availability which guarantees zero downtime of the workload, which is a mandatory option for a solution energized in the cloud. This means that if a virtual machine in the group fails or stops accidentally, another virtual machine is recreated with the same name and the same application according to the template used, in order to replace the failed VM. In addition, MIG allows the replication of the entire running workload in several zones in order to ensure the high availability.

[7] `https://cloud.google.com/compute/docs/instance-groups`

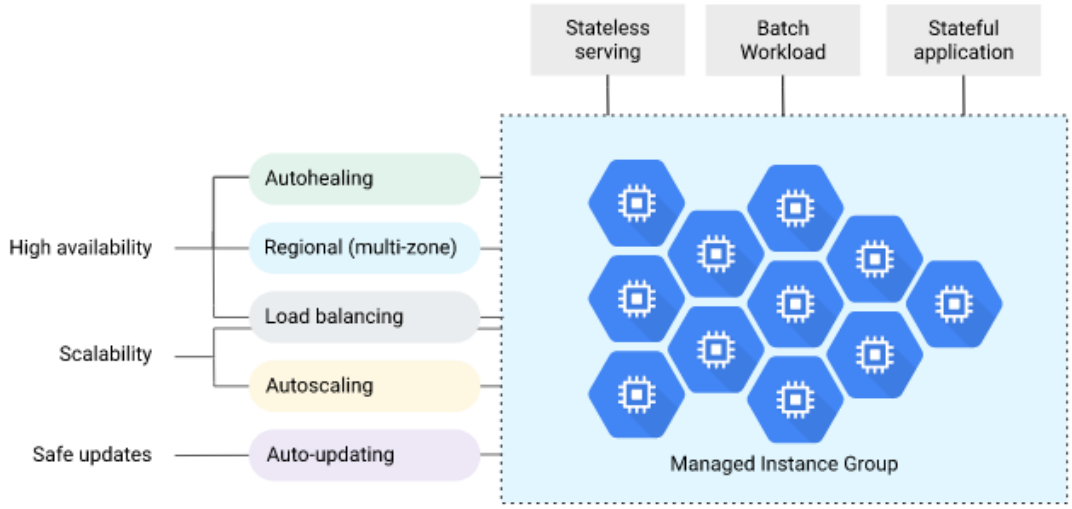

Figure 2.3: GCP Managed instance group capabilities overview[7]

Scalability

Among other benefits of using MIG is the auto-scaling which is a crucial feature that automatically scales the workload according to the configured scaling target, such as the maximum number of virtual machines when we scale (adding more VMs) out, and the minimum virtual machine when we scale (reducing VMs) in, as well as CPU unit and memory threshold to regulate the consumption.

After exploring these two approaches or VM group architectures, we find that the first approach is realistic for our application but it requires effort to maintain the infrastructure in long term. Compared to the second approach, it is less costly in terms of effort and time to achieve. Moreover, it does not allow the customization of virtual machines as well as the scalability of the cluster according to our scenario, is not feasible.

2.1.3 PostgreSQL on Kubernetes

Another possible solution to make PostgreSQL as a service is to use micro-services using a Docker container and a Kubernetes orchestrator. The Kubernetes system allows PostgreSQL to be instantiated in a cluster of machines as well as ensuring the automation and maintainability of the server throughout the life cycle. This type of system allows us to extend and customize the capabilities of the desired application through the container image with the Docker engine, which in turn plays a fundamental role in modeling a cloud-native solution. Figure 2.4 show an overview of PostgreSQL database in the Kubernetes architecture. Among the advantages of using the Kubernetes system are:

- Custom Resource Definition (CRD) which allows the extension of the Kubernetes API to define our custom resource in order to instantiate it as a native object of the Kubernetes system.
- Simplified deployment once the kubernetes system is installed with its necessary resources including the Docker engine in the machine generally named Master node, the declaration and configuration of the application with the desired characteristics within the Kubernetes cluster is declared simply with a few lines of text, this makes the infrastructure as a code. This declaration is written either in YAML

[8] `https://www.percona.com/blog/dbaas-on-kubernetes-under-the-hood/`

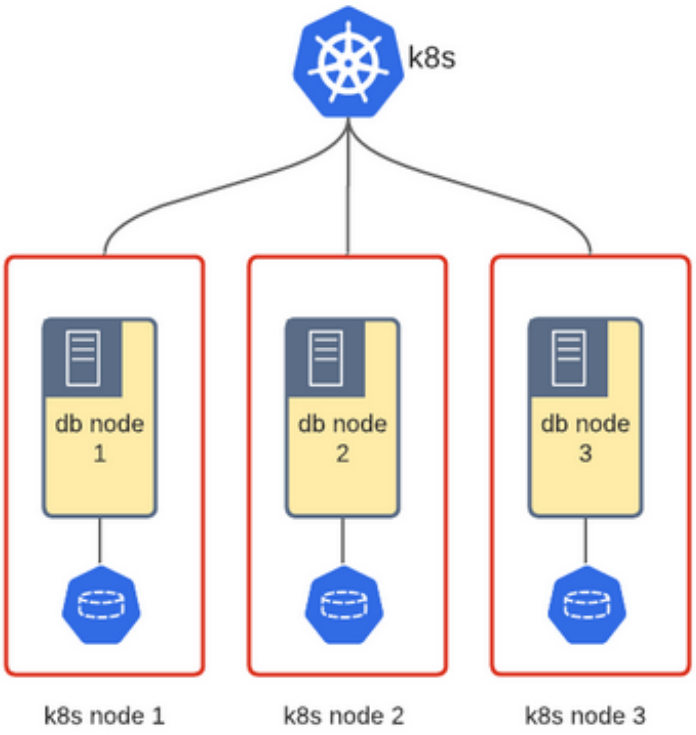

Figure 2.4: Kubernetes insight for PostgreSQL[8]

or JSON language which makes understanding any infrastructure declaration in a user-friendly way.

- Horizontal scalability that allows the Kubernetes system to add replicas and hardware resources to meet the needs of the workload and distributes the work across several nodes.
- The portability of the workload declaration in other cloud platform, whether for the public or private cloud with some adjustments in the workload declaration files.

The flexibility of the Kubernetes system with their rich functionalities that provides long-term maintainability for solutions hosted in cloud platforms. This type of system offers us the possibility of extending the functionalities of the PostgreSQL by adding and customizing our own Docker images. In addition, the horizontal scalability of the Kubernetes cluster is customizable, which leads us to apply the partitioning and distribution of the workload across the cluster according to our scenarios. There are two ways to create a Kubernetes cluster within the cloud provider. Either by first allocating a group of machines, then manually designing our cluster from the control plane, to setting up the worker nodes. Or we can use the cloud provider, it's own Kubernetes engine which is simple to build using a few command lines or through the service provider interfaces. Among those systems, we have Google Kubernetes Engine (GKE) from Google Cloud Platform, Elastic Kubernetes Service (EKS) from Amazon Web Services, and Azure Kubernetes Service (AKS) from Microsoft Azure.

we have explored several approaches to transform PostgreSQL into a service. These approaches include utilizing well-established cloud service providers, AWS, GCP and Azure which offer fully managed services or even unmanaged services. Or we can use the managed or unmanaged group of VM. Additionally, we examined the concept of deploying PostgreSQL as a service within a Kubernetes system, which holds the potential to meet

the prerequisites of our prospective cloud-native solution for MobilityDB on GCP.
In order to choose such a system from aforementioned approaches, it requires careful planning and depends on the project requirements and application scenarios. As the goal of our work, we need to deploy MobilityDB extension for PostgreSQL on cloud provider within GCP use case, in a distributed manner using Citus extension combined with MobilityDB. we performed a comparison between these systems to notice capabilities and limitations, and then determined which features might meet our objectives requirements. In Table 2.1, we summarize the functional coverage of the different systems and architectures mentioned in this part. The main systems that we have explored which provide the PostgreSQL server as a service are the three cloud platforms AWS, Azure and GCP which have PostgreSQL integrated into their services, a second architecture is to use a group of virtual machines fully managed by the cloud provider or even managed by the user. The last system studied is the Kubernetes system.

Features	**DBaaS**	**VMs managed**	**VMs unmanaged**	**Kubernetes**
Provision on demand	Yes	Yes	No	Yes
Customized extensions	No	No	Yes	Yes
Horizontal scaling	Yes	Yes	No	Yes
Data partitioning	No	No	Yes	Yes
Data distributing	No	No	Yes	Yes
Configuration flexibility	Yes	Yes	No	Yes
Stateful workload	Yes	Yes	Yes	Yes
Auto-healing	Yes	Yes	No	Yes

Table 2.1: Comparison between possible system to deploy PostgreSQL in the cloud

By analyzing Table 2.1 which shows the non-exhaustive list of features, we can see very well that the Kubernetes system covers a large functional part, if we compare the database as a service and the group of VMs. Indeed the choice of the cloud system depends entirely on the application and the scenarios to be developed which are determined by the user and the customer. In the next part, we studied the different methods on how to distribute databases through database management systems in general, and then knowing the systems that allow this distribution in cloud platforms.

2.2 Distributed Databases

Until now we have exploring the possible architecture and systems to deploy PostgreSQL in the cloud. Now we focus on the distribution mechanisms of databases. Especially distributing PostgreSQL for moving object databases. A distributed database [49] is a collection of data co-located in a cluster of nodes. Those databases are logically interrelated, meaning that such a table in a distributed database is fragmented horizontally in several nodes. The fragmentation criteria depend on the nature of data and the objectives of the user. The main goal behind the distributed databases is to gain the performance in term of query processing time and fetching data with low latency. A distributed database system it may be resided in the cloud using provider's infrastructures as compute engine and storage nodes or on premise within the data center within enterprise. distributed database is crucial if we have a multi tenant SaaS application. It allow us to centralize a database schema on different zone as shown in Figure 2.5. In addition we could use a

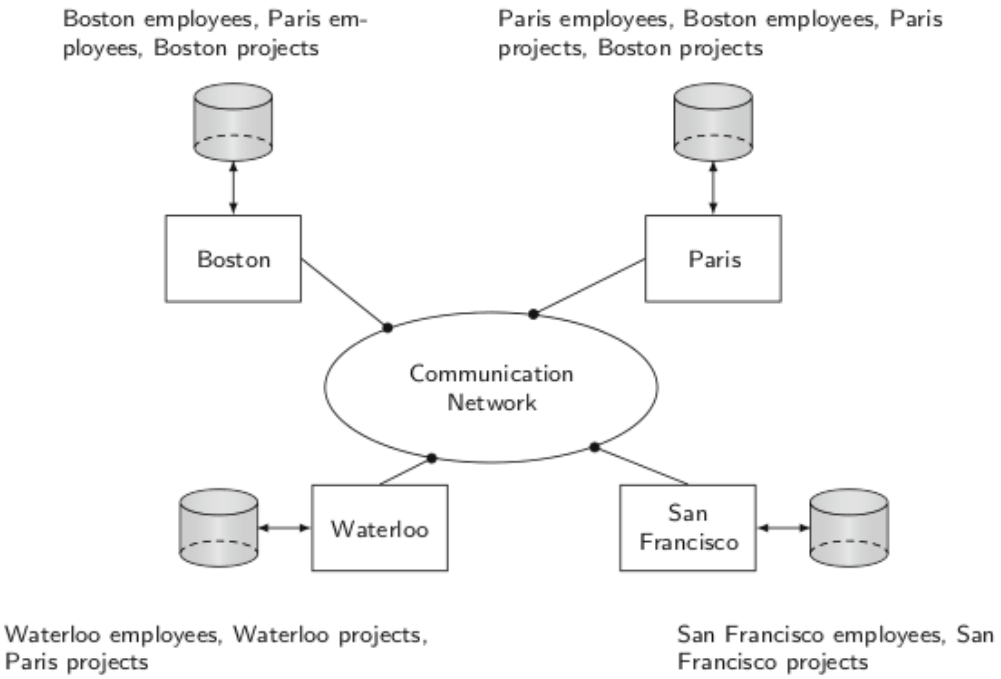

Figure 2.5: Multi-tenant distributed database [48]

distributed database for analytic application that have a complex analytic queries that requests more computation power and storage. Figure 2.2 is an architecture of a distributed database server on multi node cluster.

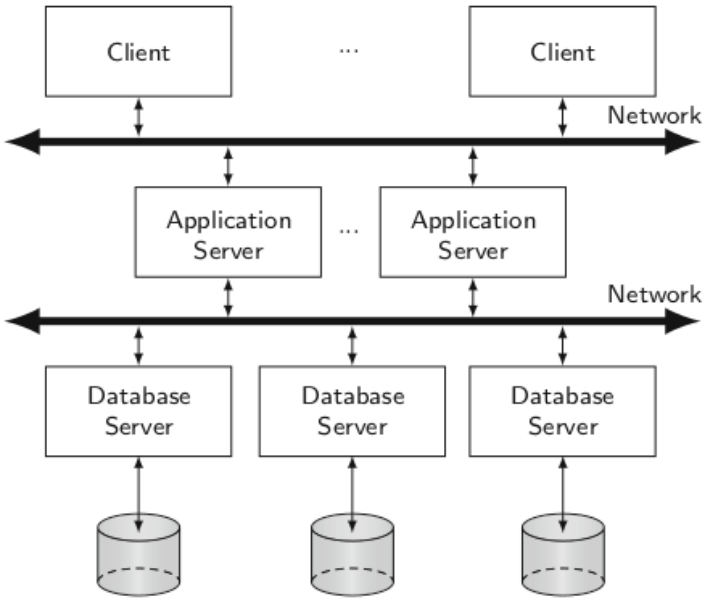

Figure 2.6: Distributed database servers [48]

2.2.1 Distributed Database Design Challenges

Distributing a database in a group of localities reveals several advantages, including scalability, high availability and fault tolerance as well as improving performance. However, several trade-offs must be considered, including data integrity and consistency. For each

trade-off cited, we give a counterpart overview in relation to our prerequisites objectives of our application, in order to make a parallel between the exploration of the possibilities with the needs of our future application.

- Design of a distributed database: in order to benefit from the advantages of the distribution, it is first necessary to choose the design of the distribution architecture which may depend on the nature of the data and the type of application desired to be handled by the end users. There are two main distribution approaches, partitioning and replication. Partitioning consists in fragmenting the database into several fragments in order to distribute them in different sites. For the replication design consists of replicating the database in several sites, it can be fully replicated, this mean the entire database is duplicated in several sites, or even partially replicated where some partitions are duplicated in a few sites only. For such a mechanism chosen, a problem relating to the catalog directory which needs to be centralized between all the sites, in order to manage the meta-data concerning the locality of each partition.

- Distributed Data Control: is an essential element in distributed databases to ensure data integrity and consistency. The distribution of the database adds this type of challenge which is indeed produced especially in the transactional databases, in which the user uses the CRUD operations which could impact the data at the tuple level, CRUD for create, read update and delete within the relational database. Regarding our scenario and the type of our application, the data we store is generally read-only because our SQL queries are totally analytical which interrogates historical data.

- Distributed Query Processing: usually in a classic database server, an SQL query is executed by the database engine algorithms which translates the query into a series of operations to manipulate the data. In a distributive environment, the parameters are totally different so the algorithms need to take in consideration additional factors, like the location of the data, the cost of communication between the different partitions which is affected by the distance between the locations of the data. We will discuss in more detail the different approaches that can overcome this challenge in the distributed query processing section.

- Distributed Concurrency Control: is another challenge in the context of a distributed database to ensure data integrity through access synchronization. In a distributed environment, generally the transactions to the database are synchronized just after the execution in all the replications, this means if we lose the concurrency control for a given replication, the problem will propagate in all the database replications. Approaches like locking-based preserve the mutual consistency of data where all tuples are identical in all replications.

- Reliability of distributed DBMS: in distributed systems, it may happen that some location or data site becomes inaccessible or even non-operational due to a technical problem. It is necessary that the system can bring the site back up-to-date after the possible technical repair. Such a mechanism preserves the consistency of the data.

- Replication: data replication in multiple sites ensures high availability and fault-tolerance. This requires a duplication protocol for transactional databases. There are two protocols to duplicate data, the first is to copy the data once the information is saved in the master location. The second is to copy the data before the transaction completes.

- Parallel DBMS: a distributed database is very similar to a parallel database but they do not have the same objectives. A so-called parallel database, if the data is partitioned and located in a cluster of machines which behaves like a single machine. The main objective of a parallel DBMS is high scalability and performance.
- Database integration: the integration of data makes it possible to collect data from the source in order to host them in the storage media of the RDBMS. In the context of a distributive system, the integration of new information must follow the distributive design or must be saved in multi-database in the case of a database which is duplicated in several sites. Compared to a parallel database, the new information must follow the partitioning protocol, namely the hashing algorithm or other algorithm.

2.2.2 Distributed Database Design

Distributing a database requires a preliminary design which allows to decide on the approach and the methodology of distribution and partitioning, which makes it possible to meet the objectives of the desired application. The distribution begins by designing the global conceptual schema of the database then the two main phases, partitioning and allocation of the data. Each chosen design has its own challenges as mentioned in the previous section. After having modeled the global conceptual schema for a distributed database, data fragmentation can be performed either in horizontal mode where the table will be partitioned by the selection operator (predicate), which will split according to a given condition, or either in vertical mode where the table will be partitioned by the projection operator which allows the table to be divided for certain attributes. Allocation in it turn, plays a crucial role in performance which depends on the distance between data locations which will impact communication between locations.

Data Fragmentation

As mentioned, data fragmentation is possible either horizontally or vertically depending on the overall conceptual scheme and the type of application desired. Horizontal fragmentation is much more used in parallel DBMSs which helps to increase the performance of SQL queries that will be translated into `intra-query` parallelism. Intra-query parallelism consists of dividing a global query into a set of queries that will be executed in relation fragments in all data locations. Moreover, the horizontal division facilitates navigation to the right tuples through the satisfaction of the predicate which will route the operator directly to the right place in order to avoid sequential reading. Vertical fragmentation targets a different kind of application and performance goals. It is used much more in so-called `column-store` DBMSs which allows very fast access for certain number of attributes where these attributes are generally solicited more than other attributes. Nevertheless, it is necessary that the two fragmentation approaches ensure the `completeness` which consists in ensuring that a tuple is present in a relation R, it is mandatory to find it somewhere in the fragments of R, and the `reconstructability` which ensures the obligation to find the relation R from the reconstruction of the fragments of the relation R.

Data Allocation

Data allocation is the phase that follows fragmentation. In practice there are two types of database use cases, we note multi-site or multi-tenant databases as shown in Figure 2.7

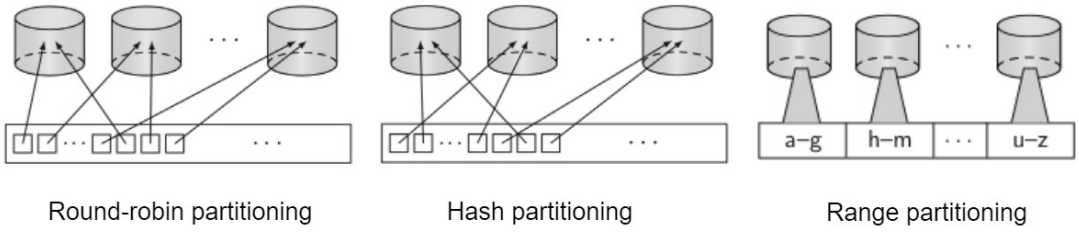

Figure 2.7: Data partitioning methods

where the fragments are distributed in several sites in order to facilitates access and separation of management by site. The data located in several sites are therefore geo-located. Furthermore, the allocation can take place in a data center this time where the fragments are distributed in several nodes, this favors the parallel processing of SQL queries in an efficient way.

2.2.3 Parallel Database Design

Today's application process data on a massive scale. As a result, computational efficiency is a requirement for transactional (OLTP) and analytical (OLAP) database systems. A parallel or multi-processor system consists of a set of nodes (CPU, memory and disk) which are connected to each other by a fast network. This allows the acceleration of the processing for complex queries that manipulates a large database. There are two types of parallel computer, tightly coupled and loosely coupled multi-processor. The strongly coupled processor consists of a set of processors connected to each other in the bus level with a shared memory. This build provides supercomputers. This type of computer is used for targeted tasks that are generally repetitive such as image processing for example. Compared to the weakly coupled type, it is constituted which refers to the group of machines which work at the same time or cluster of computers which are interconnected with each other by a high speed local network. The main advantage of using a computer cluster is the extendability which makes it convenient to add new nodes to boost performance as shown in Figure 2.8 through linear speed-up and linear scale-up. The linear speed-up makes it possible to ensure the speed of execution when the number of nodes is increased for a database, that is stable in terms of data volume, for the linear scale-up we increase the resources in order to stabilize the performance of the database that grows over time in terms of the size of the data. Among other advantages, the ease of maintenance of the equipment in the event of failure as well as the cost is advantageous, comparing the strongly coupled multi-processor. In this part, we discuss the parallel architecture in general then we briefly introduce the parallel algorithms which ensure the parallel computation of relational queries. Finally we introduce the load-balancing technique which is used to boost high availability and fault-tolerance.

Data Placement

The location of the data for a distributed RDBMS in a cluster of nodes or even multi-site architecture is realized by a physical server for each site which makes it possible to host the partitions of the horizontally distributed global database. The organization of the partitions within the nodes is based on two main indexes, a global index and a local index. The global index allows us to locate the location of the partition in the cluster by identifying the node that contains this partition, this type of index somehow maps the global relationship into a series of partitions distributed evenly over the number of nodes

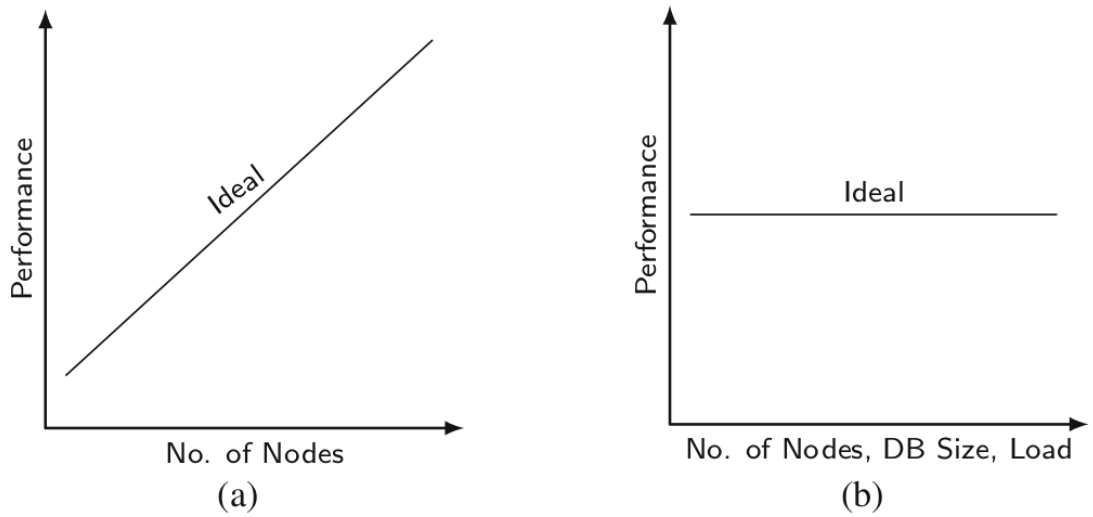

Figure 2.8: Extensibility metrics. (a) Linear speed-up. (b) Linear scale-up [49]

that exist in the cluster. The local index allows us to locate the partition at the level of a disk pages, this index is carried out by the B-tree index generally which makes a map of the relation (partition in a parallel RDBMS) in a set of disk pages. Figure 2.9 gives us an example of these two types of indexes for the employee relation with it different indices which allows quick access to the desired locality. The location of the data requires

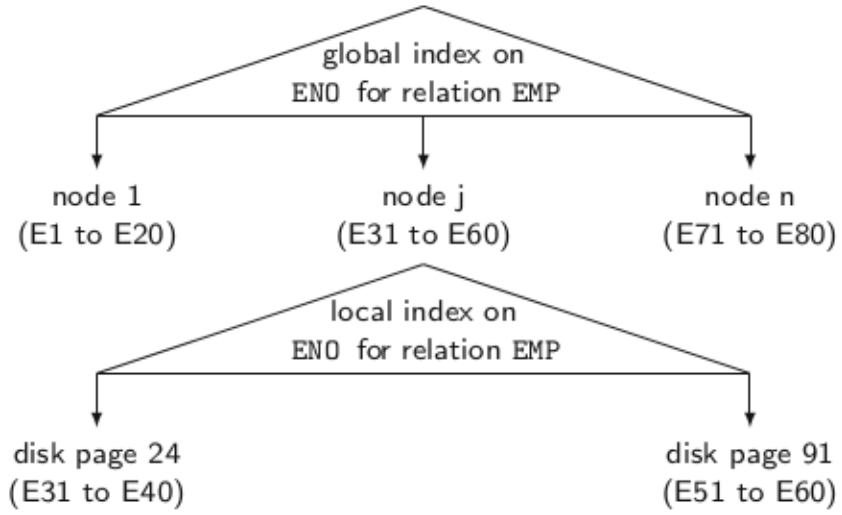

Figure 2.9: Example of global indexes and local indexes [49]

a reorganization of the partitions in the case of a new partition insertion, in order to recalculate the global index to ensure an equitable sharing of partitions within the cluster.

Parallel Architecture

A perfect parallel architecture depends on the choice of design in terms of hardware including the CPU, memory and disk as well as the connectivity between the different nodes of the cluster. In practice, there are three types of distributive architectures including:

- **shared memory architecture:** where all processors share a single memory and behave like a single machine, connectivity for this type of architecture is ensured by a single buffer bus. The advantage of shared-memory is the simplicity to program the algorithms since we are in a single environment, moreover the intra-query

parallelism becomes easy with only a few adjustments at the level of the algorithm. Shared-memory uses either Uniform Memory Access (UMA) where all the processors share a single memory that can be upgraded by adding additional memory. Unlike Non-Uniform Memory Access (NUMA) which provides each processor with its own memory, this means that we will have more cache memory compared to UMA. Shared-memory guarantees vertical scalability (`scale-up`), as shown in Figure 2.10.

- **Shared-disk architecture:** allows processors to have access to all disks through interconnection between processor memory. So the processor can recover data from all the disks that exist in order to hide them in its calculation memory. In order to allow this access, shared-disk needs to be globally accessible in the cluster through Network-Attached Storage (NAS) which uses the TCP/IP protocol to communicate with the distributed file system. The NAS type of connectivity is useful for applications that do not require high performance, typically used for backing up and archiving data, and Storage-Area Network (SAN) which has the same characteristics as the NAS but with efficient protocols that facilitates the management of cached memories. The advantage of using shared-disk is the simplicity and the reduced cost of implementations because the database administrator does not worry about the fragmentation and the distribution of the partitions as well as if a node cause an issue, the lost data is located only in this failing node.
- **Shared-nothing architecture:** in this type of architecture, neither the memory or the disk is shared between the processors, where each processor has access to its own memory space through the Directly Attached Storage (DAS). This type of architecture is widely used in practice, especially for multi-tenant applications where the global database is distributed and replicated across multiple sites. Cluster processor-memory sharing is the perfect approach for this type of distribution where each node represents a site. Same thing for OLAP queries which can be executed as intra-query processing where each node of the cluster takes a division of the global query before the merge of the results is carried out. This boosts performance and guarantees horizontal scalability (scale-out).

Parallel Query Processing

Parallel computation of relational queries is performed through the mechanisms that transform the query processing plan into its parallel query processing plan counterpart that can be executed in a distributed environment. The response time of a query executed in parallel in a distributed environment depends not only on the number of partitions that exist in the cluster, but also the location of these partitions which characterizes the time taken to fetch the data from the disk into the memory for a given node in the cluster. The execution of a relational query in a single server is usually handled by query plan algorithms or operators such as the sort algorithm, join and hash join algorithm. These latter executors require adjustment if they are needed to be used in distributed DBMSs. In the following, we give an overview of the parallel version of these algorithms.

- **Sort algorithm:** is widely used to order tuples or it's used in aggregations and group by expressions. The Quicksort algorithm is one of the quickest single-processor sort algorithms, but because it is highly sequential, it cannot be adapted to parallel processing. Other centralized sorting techniques can be parallelized. The parallel merge sort algorithm, which is simple to build and need few adjustment requirements for the parallel system architecture, is one of the most often used algorithm. It has been applied to shared-disk and shared-nothing clusters, respectively. In order

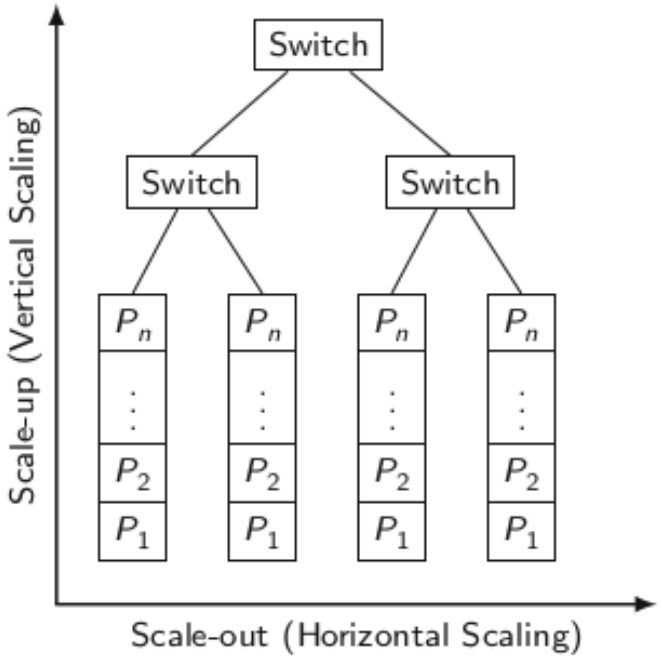

Figure 2.10: Scale-up versus Scale-out [49]

to understand how the merge sort algorithm works, we assume that we have a distributed database system with partitions located in several nodes. And assuming the use of the well-known master-worker node model for carrying out parallel tasks, in which one master node manages the actions of the worker nodes by distributing tasks and data to them, and requesting results on tasks completed. The algorithm proceeds with two main stages, the first consists in doing the sort by each one of the nodes of the cluster for these own fragments which exist in the local space of this node with the classic quicksort algorithm, and the second stage consists in doing the merge of all the sort results which will be redirected and merged at the level of the master node.

- **Parallel join algorithm:** is the most used, especially in analytical queries. The parallel merge sort join algorithm, the parallel nested loop (PNL) algorithm, and the parallel hash join (PHJ) algorithm are the three fundamental parallel algorithms. These algorithms are versions of the ones used in centralized systems. The parallel merge sort join algorithm merely performs a parallel merge sort on both relations. Despite the fact that the previous action was sequential, the joined relation's result is sorted according to the join attribute, which is important for the subsequent operation. The remaining two algorithms run entirely in parallel. Regarding the parallel hash join is fully parallel algorithm, it takes place in two phases. The build phase and the probe phase. The build phase hashes the join attribute of the R used as an inner relation before sending it to the target nodes, which create a hash table for the incoming tuples. The target nodes that probe the hash table for each incoming tuple are sent S, the outer relation, associatively during the probe phase. Therefore, the S tuples can be delivered and processed in pipeline, by probing the hash tables as soon as the hash tables have been constructed for R.

2.3 Distributed Moving Object Databases

The moving object databases aims to store real-world mapped geographic objects in order to manipulate the trajectories and travels of objects in time and space. Moving object databases have complex data types such as points, lines, or geometries that are typically difficult to process as well as its grows exponentially over time through advanced data capture and integration technologies such as the 5G era, sensor and streaming data, and IoT technologies. Among the objectives desired to be achieved by transport and logistics organizations or even the government, are transport problems such as road or maritime traffic, traffic jams or even the elimination of fraud and organized crime. The manipulation and analysis of moving data of objects whether vehicle, plane, ship or individuals in space and in real time through analytical applications is a lever that allows to overcome the problems indicated above. Generally, moving object databases are also called spatial and spatial-temporal databases. Among the challenges faced by moving object databases, the rapid evolution of data which poses storage and performance issues. The horizontal distribution of this type of database within a cluster of machines could be a real solution in order to keep the ideal performance through the alignment between the number of nodes in the cluster, the size of the database and the existing load. In this section, we will define spatial and spatial-temporal databases with their complexity, then we present past research and solutions that deal with the same type of database, in order to have insight and inspiration to meet the requirements of our application objectives.

2.3.1 Spatial Databases

In order to represent and query geometries in a natural way, spatial database research aims to extend DBMSs data models and query languages. A DBMS implementation must be expanded to include corresponding data structures for geometric shapes, algorithms and indexes for performing geometric computations, multi-dimensional space indexing techniques, and extensions to the optimizer including translation rules, cost functions and query execution plan choices to map from the query language to the new geometry-related components. Geographical Information Systems (GIS) are the primary driving force behind the use of spatial databases. Early GIS systems used DBMS technology sparingly, maintaining geometries independently in files while keeping non-spatial data in an DBMS system. The development of spatial database technology however, has led to the availability of spatial extensions from all the main DBMS providers, including Oracle, IBM DB2, and PostgreSQL server. As a result, it is now simpler to completely develop GIS as a layer on top of a RDBMSs, that allow to store all the spatial data in the RDBMS. Regarding image databases and spatial databases, there are certain key differences. The goal of spatial DBMS is to represent objects in the space with a clearly defined and accurate location provided by the GIS system. In addition the geographic space can be represented by images collected from aerial photography or satellites. As such, image databases manage images. Geographic objects this time may be extracted using feature extraction techniques that make it possible to segments geographical objects in a picture that can be kept in a spatial database.

In order to record geo-spatial objects, modeling and abstraction of these objects is necessary to allow easy manipulation. The three primary abstractions for modeling single objects are `point`, `line` , and `region`. The geometric feature of an object is represented by a point, where just the object's placement in space but not its size. Cities on a big scale map, landmarks, hospitals, or subway stations are examples of point objects. The

simplest metaphor for traveling through space or making connections in space is a line, which is in this context always refers to a curve in space. Finally, a region is an abstraction for an object with a 2D spatial extent which is similar to a polygon. In general, a region might have holes and be made up of multiple fragments. Countries, woods, and lakes are a few examples of region objects. Figure 2.11 provides illustrations of the three fundamental abstractions [21]. The relationship between these three main abstractions

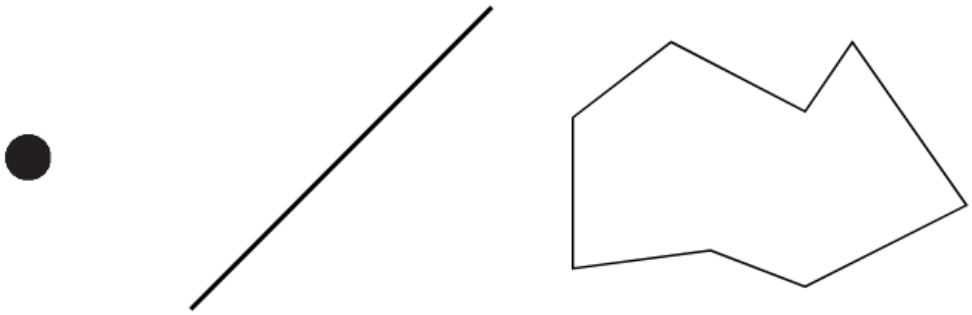

Figure 2.11: Basic abstraction for geo-object, point, line and region

for geo-spatial objects allow us to represent other objects as a partition or network. A partition is composed of several disjoint regions between them where each region has one or more boundaries. The partition consists of representing a map such as a map of a country with its different cities. A network as a graph embedded in the map, it composed with a set of point objects for its nodes and a set of line objects for its edges. In geography, such a networks we have roads, rivers, transportation hubs, and electricity lines. Figure 2.12 show the abstraction of partitions that represent regions and network which

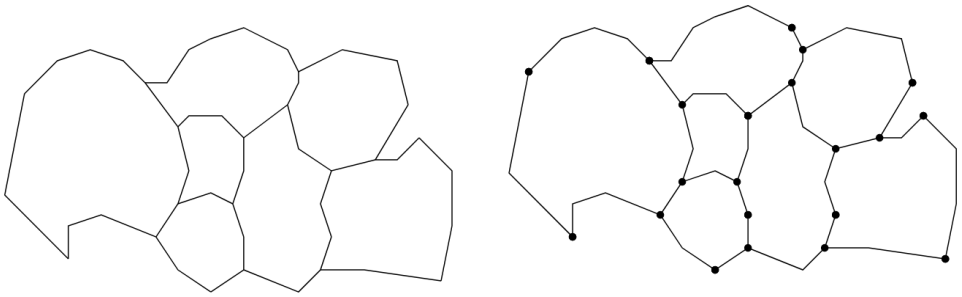

Figure 2.12: Partition and network abstraction

is composed from a regions highlighted by a set of point connected through line object.

2.3.2 Temporal Databases

Standard relational databases generally record the status of objects in order to query, modify or analyze them. Many applications are interested in the change of these status over time in order to keep the history of those states of the objects, to make possible the analysis including comparisons between the past and present states or even to predict the future states of the objects. The data types supported by RDBMSs which characterize time as date or time type are limited which lead to complex query formulation.

Therefore, the objective of research on temporal databases [21] has been to fully incorporate temporal ideas into the RDBMS query language and data model and to extend the system in accordance with this to achieve efficient execution. Examples of the concepts of time integrated within an RDBMS system are the following:

- **Instant:** is a particular chronon on the time line in the discrete time model, or a point on the time line in a continuous time model.
- **Period:** an anchored interval on the time line.
- **Interval:** is a directed, unanchored duration of time. That is, a time interval of known length with unspecified start and end instants.

Incorporating the time aspects in an RDBMS is possible by extending the data model where the object (tuple in a table) is considered as a fact, this fact is associated with a time data type attribute, represented by a timestamp type generally.

2.3.3 Moving Object Databases

The idea of moving object databases [21] is to have the ability to track objects in space and time through the extension of RDBMS by supporting geo-spatial and spatial-temporal data. For example, if you want to follow the route of a vehicle, you just have to query the database on the different points crossed by this vehicle over time. This means mapping between different locations (points or nodes) in different timestamps. The moving objects databases allow us to explore the trajectories in an efficient way through the solicitation of the states of the object in space and time. A moving point is the adequate abstraction in an RDBMS to designate a moving object in real life. Table 2.2 shows some relevant of possible moving object queries.

Moving object	Query
Cars: taxis and buses	What are the buses that crossed the road of vehicles? Taxis that exceed the authorized speed in the city center?
Ships	Are there any ships en route to shallow waters? Track down "strange" ship movements that can indicate unlawful trash disposal.
Plane	Planes that are close to collision? Will there be a collision between two approaching planes? Did aircraft fly across the airspace of state X? How quickly does this aircraft move? How fast can it go?
People	What route is taken by the protesters? Show the trajectories taken by the suspect individuals.
Military vehicles	What are the submarines that cross the area X? what are the different possible short paths to get to destination X?

Table 2.2: Moving objects and it corresponding queries examples

As we can see in the table, several possibilities of analysis that we can do through the positions of the objects with their timestamps. We can also generalize these queries for other objects such as animals, satellites, etc. In addition to moving objects, it is possible to analyze and track moving regions this time, such as the movement of storms over time which allows us to avoid a natural disaster or the movement of virus which allows us to stop the spread and contamination of the diseases, these moving regions are usually represented by real-time maps that help us making decisions. Queries for moving objects or regions are generally complex to express in relational expressions as well as for records requires an abstraction of the data model in order to simplify handling for database administrators. This is why the extension of the RDBMS by adding spatial and spatial-temporal features makes the database customizable for moving data objects. Moving object data types

embedded in an RDBMS are characterized by the application f : `instant` $\rightarrow$ `object`. This function represents an object in continuous time. In particular for the moving point object is written by the application f : `instant` $\rightarrow$ `point`. and f : `instant` $\rightarrow$ `region`. for the region object. This will produce `mpoint` and `mregion` types for moving object and moving region respectively.

2.3.4 Moving Object Features for PostgreSQL

Several commercial systems that have extended RDBMSs by integrating new functionalities and abstract new data types, in order to provide facilities to storing and analyzing moving object data as well as simplifying the declaration and expression of relational queries that interact with spatial and spatial-temporal databases. MobilityDB[9] extension for PostgreSQL has been developed to make the PostgreSQL supports the moving object data type as well as a series of pre-defined functions have been provided in order to ensure the flexibility of writing analytical queries for moving object databases. MobilityDB development team have contributed [44] over the past few years to customize PostgreSQL for handling moving object data. Moreover MobilityDB has been tested and implemented with other systems and platforms in order to cover a large scope of application. Indeed MobilityDB for PostgreSQL is considered as the basic element of our memory work, which consists in integrating it into a distributive system in the cloud. We will talk more about the features of MobilityDB in the technical background chapter.

2.4 Moving Object Database on Cloud Services

In this section, we analyze the possible solutions that can support large scale moving object databases, that are supported by the MobilityDB extension and the PostgreSQL RDBMS. Sandhu et al. [33] discuss the challenges when adopting cloud computing for treating large scale database using cloud systems. In following, we present the work carried out last year which consists in integrating the MobilityDB extension within the Kubernetes system EKS of AWS[10] which had as objective, to scaling a moving object database in order to reduce the response time of analytical queries, and guarantee an ideal performance when data changes over time. This work is carried out by us last year within the scope of the course computing project, which consists in carrying out a project during one semester. A second work already done which consists in integrating the MobilityDB extension with the Kubernetes AKS system from Azure [37] in order to scale horizontally moving object database at scale, while ensuring the integrity of the data in the event of solicitation of scale-in or scale-up operations. In addition, an auto-scaler has been developed which makes it possible to automatically scaling the resources of the cluster in the event of a heavy load or even a larger database size through the metrics measured. This work was carried out as part of an end-of-study project for a master degree. These two implementations lead MobilityDB in the cloud in order to make it scalable in a distributive environment which allows to allocates hardware resources (CPUs, memories and disks) in an automatic or manual manner in order to meet the needs of the moving objects workload. By analyzing these two solutions, this helps us to integrate MobilityDB into GCP cloud and allow us to factor the necessary features that are common to the three cloud providers in order to prepare a cloud-native MobilityDB.

[9] `https://github.com/MobilityDB`

[10] `https://github.com/MobilityDB/MobilityDB-AWS`

2.4.1 MobilityDB and AWS

The integration of MobilityDB in AWS was carried out after exploring the different services provided by AWS in order to know the services that can support the prerequisites of moving database objects. Among the services and products that ensure the scalability of MobilityDB in AWS were the Elastic Container Service and Fargate, Elastic Kubernetes Cluster EKS and the management of a group of EC2 machines from AWS.

- **Elastic Container Service and Fargate:**[11] is a service entirely managed by AWS which allows us to manage containers without going through a server in order to configure scaling these containers. In other words, the Fargate technology resembles a container orchestrator which makes it possible to deploy applications hosted in containers in an autonomous manner as well as to ensure the supply of resources in terms of CPUs, memory and storages by an automatically. This makes application deployment easy and leaves users focusing on other aspects of application. This type of service is classified platform as a service in the cloud. The advantage of ECS and Fargate is that the management of the container cluster is delegated to Fargate technology.

- **Elastic Kubernetes System:** is an AWS service that allows us to manage a Kubernetes cluster in the AWS ecosystem. The advantage of using the EKS compare ECS service is the portability, extensibility, and simplicity to declare Kubernetes cluster objects. With portability meaning if we want to migrate a Kubernetes cluster from a cloud platform to AWS, it will be easy. In addition, deploying an EKS cluster it will be paired with other AWS services for instance CloudWach for monitoring, Elastic Load Balancer for load-balancing, Identity and Access Management(IAM) for users and permissions and VPC for networking.

- **EC2 managed group:** is another way to scale our MobilityDB using an Elastic Compute cloud (EC2) service[12] which is a service from AWS that provides a list of features that allows the developer to take care only of workloads and leave the infrastructure managed by AWS. The main idea of this choice is to adapt the EC2 service so that it covers the prerequisites of a moving objects database that scales horizontally over time, is simply by creating a group of EC2 instances by associating each EC2 interface with a Citus node within the cluster. Thereafter a master node is designated and the remaining nodes are worker and a 3rd type of node which is the manager, is used to listen to new worker nodes in the same subnet in order to automatically join them with the master node. Among the limitations of the EC2 service is that it is not paired with another service, let's compare EKS, which is paired with the CloudWatch service,[13] which provides real-time resources and software monitoring.

The choice among the three possibilities was to opt for Kubernetes systems from AWS because of their advantages mentioned above. The main tasks of this work is to encapsulate all the functionality needed to distribute moving object databases in a single docker image, and then deploy that image into AKS. In addition a tutorial on how to activate the vertical auto-scaller that add more resources vertically in order to idealize the performance. This native cloud image includes the PostgreSQL, PostGIS and MobilityDB extension and Citus extension which is used for partitioning and distribution.

[11] `https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html`
[12] `https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html`
[13] `https://aws.amazon.com/cloudwatch/`

2.4.2 MobilityDB and Azure

A similar work has been developed which consists of integrating MobilityDB in Azure, this time carried out by [37]. It aims to provide a self-scaling mobile object database on Azure Cloud service. The main contributions during this work are as follows: Automation for the initialization of the Kubernetes system and Citus cluster where the user provide only a few command line parameters such as the size of his future cluster, the type and capacity of the machines desired to be used for his workloads. This is achieved by previously defined processes in the background which allow the distributive environment to be started. A second main mission of this work is the self-scaling of the distributive environment which allows to change the size of the Kubernetes cluster in a dynamic way from the observability of the metrics captured from the resources during the loading of data and the current workload's run-time. This is done by a process that periodically runs in the master node to monitor the moving object database size and performance during the workload. This process works in four phases, monitor, analyze, plan and execute, these four tasks energize the processing environment of the moving object database and allow the cluster to automatically provision Azure resources in terms of CPU, memory and storage space. Figure 2.13, illustrates a high-level architecture of the Kubernetes and Citus cluster on top the MobilityDB extension, which is implemented in a docker container image combined with Citus to manage moving objects data in a distributive and auto-scaling environment.

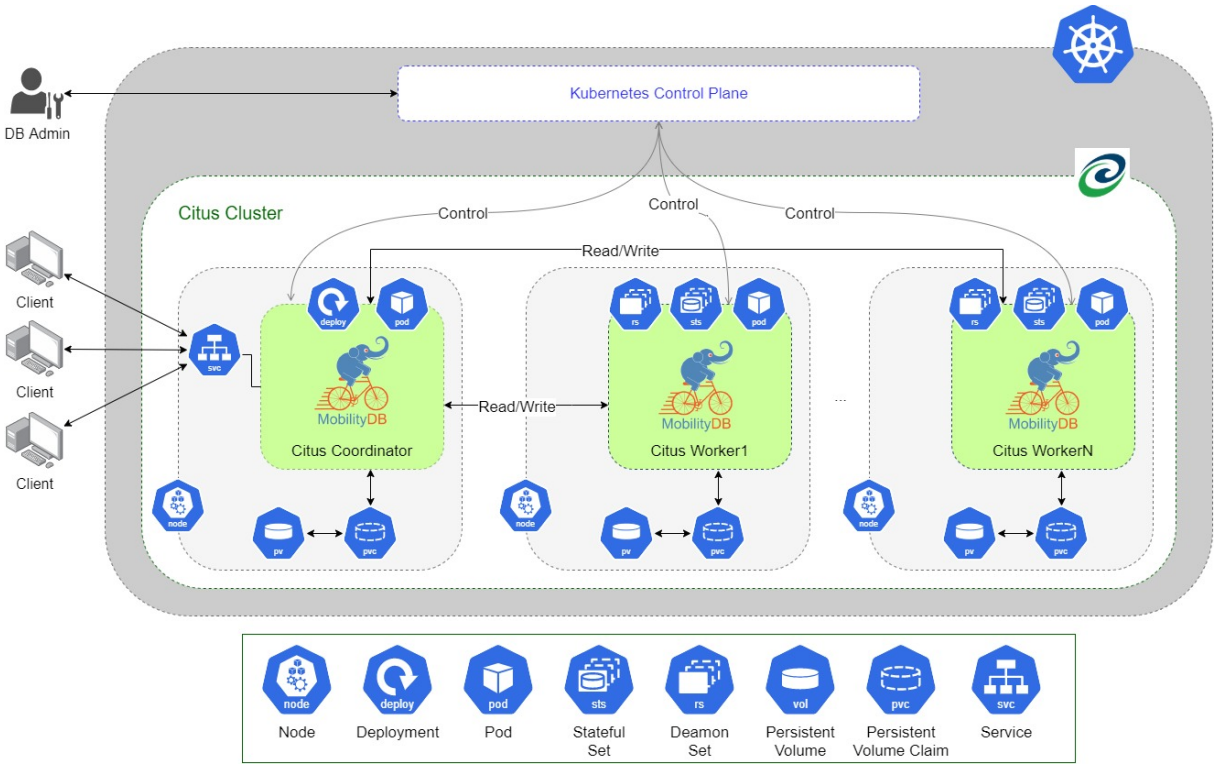

Figure 2.13: Self-scalable moving object database on Cloud: MobilityDB and Azure [37]

In order to better understand the flow that allows us to arrive at this distributed environment above is as follows: Initially, the database administrator interacted with the control plane of the Kubernetes cluster in order to initialize the Azure infrastructure in terms of the number of nodes in the cluster, the type of machine desired to be set up as well as the power of each machine in the cluster. Subsequently the administrator proceeds to the deployment of the MobilityDB and Citus extension for PostgreSQL, then he integrates

his moving object database, distributed it with Citus which allows the partitioning of the tables into a set of shards before distributing them within the cluster. The execution of the analytical queries made by the client are transferred to the Citus coordinator node which then proceeds to distribute the query to the cluster's nodes. The nodes within the cluster in turn execute the analytical query locally before sending the result to the coordinator node which will then merge the results received from the different nodes. The Kubernetes objects in blue define the Kubernetes system asset which will be explained in detail in the next chapter.

2.4.3 MobilityDB and Apache Hadoop

The Apache Hadoop framework is well known for the scalability of processing in a cluster with Hadoop Distributed File System (HDFS). It allows the evolution of the workload, as well as it is adapted for the cloud environment such as (GCP, AWS and Azure). This should be a concrete solution to support moving object databases to scale them horizontally. The writing of the implementation code in Hadoop follows the MapReduce[14] paradigm which makes it possible to formulates the syntax of SQL queries in a distributed and scalable way.

M. Bakli et al. [4] in recent years have developed a Hadoop extension that supports spatial-temporal moving object data, by integrating spatial-temporal datatype algebra and implementing MapReduce tasks that act as Hadoop operators. Further more, they introduce spatial-temporal algebra types which correspond to MobilityDB spatial-temporal types in native Hadoop types. This latter simplifies and provides the ability to write analytical queries for moving object data in the HDFS environment. This contribution makes it possible to enrich the Hadoop system so that it takes into account the types of spatial-temporal data and the operators that can be mentioned for applications that analyze geo-spatial data. According to this extension, Hadoop is now ready to parallelize dedicated analytical queries for a moving object database. Eldawy et al. [14, 15] introduces and demonstrates SpatialHadoop which is another extension of the Hadoop framework that efficiently handles spatial data. This extension covers various layers, including the MapReduce functions, enriched by the two functions, SpatialFileSplitter and SpatialRecordReader, to enhance its capabilities for processing spatial data. In the same manner, another extension was developed on top of the Hadoop framework named Hadoop-GIS [1], which is a scalable system for running large-scale spatial queries on Hadoop. Hadoop-GIS experiments show its out performance compared to traditional database management systems.

Furthermore, in [40] a new prototype named VegaCI is implemented on top of the Hadoop system using MapReduce, which enhances the performance of geospatial computations. If we take the analogy between what we saw previously, for parallel execution of SQL queries on the PostgreSQL server, this task could be carried out by the Citus extension of PostgreSQL which partitions the data and distributes the queries within the cluster. Within the Hadoop environment, SQL queries are translated into operators or jobs that can be executed in parallel through the MapReduce paradigm and concerning the data type provided by the MobilityDB extension for PostgreSQL, it can be done in HDFS environment through the integrated moving object data type. Indeed this approach could be useful in the case where the end users know how to write the code using the MapReduce paradigm which ensures parallelism.

[14]MapReduce: is a programming paradigm that involves heavy workloads that aim to distributing jobs within a cluster of servers.

The trajectory processing of spatial and temporal data is distributed within a cluster with several instances of MobilityDB, which is defined and tested by Mohamed Bakli et al. [3]. Spatial and temporal data handling in big dataset Ashfaq et al. [2] investigates and experiments the efficiency of spatial and temporal query executed in parallel processing, results that the performance is gained through three main aspect, the complexity of the query, the volume of data and the nodes availability. There are another approach that aims to outperforms the classical trajectory query processing by introducing TrajSpark [9], a distributed in-memory system that ensures efficient trajectory data management. Nidzwetzki et al. [26] implements a highly distributed and fault tolerance system by coupling an DBMS data management system with scalable key-value store. It use Apache Cassandra as storage entity and the Secondo DBMS as query processing engine. Moving object data model are distributed in a scalable manner through the Unit of Works (UOWs). Furthermore, Dingju Zhu [41] introduces a cloud parallel spatial-temporal data model with adaptable parameters to efficiently tackles the challenges of managing spatial-temporal big data. This research introduces a cloud parallel spatial-temporal data model with adaptable parameters to effectively address the challenges of managing spatial-temporal big data in applications including as e-government, digital cities, and smart cities.

2.4.4 Benchmarking Geo-Spatial Databases on Kubernetes

As mentioned above, the Kubernetes system could be a solution to provide a database as a service in the cloud. In this part we will introduce a comparative study carried out by [36] which benchmarks a geo-spatial database in a clustered environment via Kubernetes and an unclustered environment. We wanted to explore this benchmark which takes into consideration the same objectives as ours as well as the type of database is the same. Moreover through this experiment we will know the different functionalities necessary to support this kind of database, as well as to know the characteristics of the cluster This article provides a comparative study between three main environments in the AWS cloud platform including,

- Amazon Elastic Compute Cloud (AWS EC2)
- Amazon Relational Database Service (AWS RDS)
- Amazon Elastic Kubernetes Service (AWS EKS)

All these services meet the prerequisites to support a database that scales over time. In addition to these three services, two execution environments which arise on two different hardware configurations have been chosen in order to diversify the comparison of this benchmark in particular. The RDBMS used in this experiment is the PostgreSQL server extended by the PostGIS extension for PostgreSQL which supports geo-spatial data and queries. The data used for this benchmark comes from the map of the states of Colorado and Washington in the United States, in the form of a list of entities or geometric shapes characterized by a list of attributes provided by OSM. For example, a building is represented by a vector geometry of polygon which is described by attributes such as the address of the building, the owner and the year of its construction [36].

In this comparative study, eight geo-spatial queries is used where each query has its own complexity. All spatial queries are analytic, and each query represents a use case that is typically accessed the most by customers from a geo-spatial web service. As SELECT queries of geographic data are more frequent than UPDATE or DELETE operations, accounting for read queries allows for comparative analysis of geo-database performance

and deployment for real-world scenarios. Queries in [36] are described in Listing 2.1, 2.2 and 2.3:

- Get count of points, lines, and polygons

```
SELECT 'point' AS tbl, COUNT(*) AS cnt
FROM public.planet_osm_point
UNION SELECT 'line', COUNT(*)
FROM public.planet_osm_line UNION SELECT 'polygon',
COUNT(*) FROM public.planet_osm_polygon ;
```

Listing 2.1: Query 2

- Get restaurants with two or more branches.

```
SELECT name, count(name) as number FROM planet_osm_point
WHERE amenity = 'restaurant'
GROUP BY name HAVING count(name) >= 3
ORDER BY name ASC;
```

Listing 2.2: Query 6

- Get length of all roads (in km)

```
SELECT highway, name, way, st_length(way)/1000 AS length
FROM planet_osm_line
WHERE highway NOT IN ('construction', 'footway', 'path',
'steps', 'track', 'cycleway', 'pedestrian', 'abandoned',
'disused') AND (service NOT IN ('parking_aisle',
'driveway') OR service is null) AND (access NOT IN
('no', 'private') or access is null)
ORDER BY name;
```

Listing 2.3: Query 7

In the following section we will briefly present the infrastructure of the environment used, the comparison metric and finally the results of the experiments.

Hardware Configurations

Basically, in this comparative study [36], they choose two different hardware configurations, for each environment mentioned before including AWS EC2, AWS EKS and AWS RDS. The purpose of changing the hardware configuration is to know which type of environment supports the processing of geo-spatial data, and which configuration allows to keep the performance stable during the heavy loading at the level of the workload. Two main hardware configurations are used, a large one and small one in terms of resource (CPU units, memory and storage space) The large hardware configuration is composed

- Instance type–t2.Medium
- RAM–4 GB
- vCPUs–2
- Storage–8 GB general-purpose solid-state drive (SSD)

The second hardware configuration is composed by

- Instance type–t3.Large
- RAM–8 GB
- vCPUs–2
- Storage–8 GB general-purpose SSD

Experiments and Benchmark Results

In this benchmark, the performance is tested in two cases, the first consists in testing the performance during the import of data into the PostgreSQL, for the three environments mentioned above, the second test concerns the eight geo-spatial queries prepared this benchmark. The result obtained after the import experience in the three environments depends exclusively on the database structure the use of indexes. On the other hand, the speed of execution of the import between the three environments varies as follows: The ability to scale up or down based on resource utilization made it possible for databases running on AWS EKS to import data more quickly than those running on AWS EC2, which had no such scaling capability. Due to the fact that AWS RDS is not designed to interact with geo-spatial data, import times for AWS RDS vary substantially from those of the other two execution environments. According to the average execution time (AET)

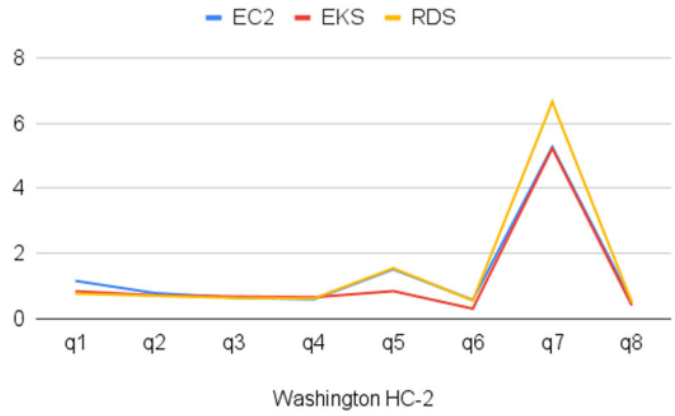

Figure 2.14: Line plot for AET for all benchmarking queries in HC-2 for Washington State [36]

captured during the execution of the eight queries for the different environments, we can say that the EKS is slightly better than the other from query one to eight. This is due to the elasticity of the EKS cluster which allows the scale-up and scale-down vertically of the cluster. Figure 2.14 show a line plot which takes in the x axis the query codes from one to eight and in the y axis the average execution time (AET), where we observe that the AET remains marginal or similar between the three environment for queries that have a low computational overhead or a low data load, on the other hand for queries that have a large data load, AWS EKS yield better AET comparing the AWS EC2 and AWS RDS seek some compatibility for the geo-spatial data. This is ensured by the elasticity of the Kubernetes cluster once again. Another metric is used during this research is the Percentage Improvement (PI) in AET, is the ration of performance gained when changing from environment to another, the formula to compute the PI is as follow:

$$PI_{AB} = \frac{AET_A - AET_B}{AET_A} * 100$$

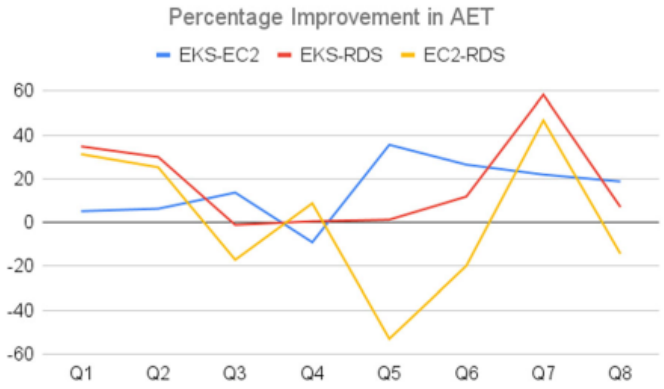

Figure 2.15: Line plot for percentage improvement PI_{AB} in AET [36]

If the PI_{AB} is positive, this mean that the AET in environment A has improved by PI_{AB} percent compared with AET in environment B for a given benchmark query [36]. As we can see from the line plot, EKS improve both environment EC2 and RDS for the majority of queries except for Query 4 due to the low computation overhead. For Query 1, Query 2, and Query 7, AWS EC2 outperformed AWS RDS due to AWS RDS's incompatibility with specific PostGIS extensions. AWS RDS outperforms AWS EC2 running regular PostgreSQL because it is designed to operate with common text queries.

Chapter 3

Technical Background

This chapter is dedicated to present the technical background used in our work to achieve our objectives. We start by introducing the Google Cloud Platform and its services which is the main object in this work that will hosts our solution. And then we will introduce PostgreSQL engine that will be used as our relational database management system to support our databases. In addition we will talk about the different extension of PostgreSQL used in this work to extends it the functionalities starting by MobilityDB extension that aims to better stores, manages and manipulating the moving object database, through the new data types and operators implemented. After that we will introduce Citus extension that aims to partitions PostgreSQL tables into a set of shards before distributing them across the cluster, and parallelize the queries across the group of nodes where each node computes the query locally before returning the result to the coordinator. Finally we defines the different deployment system used to deploy our solution from Docker container, Kubernetes system for containers orchestration and the Python client libraries used to interact with the Kubernetes cluster and Google cloud services. At the end, we will talk about the tool used to perform our deployment tests locally with Minikube system. And describing the framework used to generate our experiment's graphs and charts which is Dash Plotly for Python.

3.1 Google Cloud Platform

Google cloud platform GCP it is considered among the most known cloud providers around the world with AWS for Amazon and Azure for Microsoft. For our work, we will focus on GCP services as we need to deploy and hosts our solution on it. Using cloud services is not an easy task because of the existing of several services and products within the platform that may support the our application requirements. Choosing the adequate products to support our solution need us to explore and documenting on different GCP products. Cloud computing derives three main classes of cloud services. Software as a service (SaaS), infrastructure as a service (IaaS) and platform as a service (PaaS). Where GCP offer all those type of cloud.

Software as a Service

Software as a service (SaaS) is the uses of a fully managed software offered by the cloud vendor, in which the users do not need to handle the software by itself and focusing only on the implementation. Many benefits are ensured when using SaaS products such as high availability, scalability and security. Choosing SaaS products it is depending on

the application's needs and user's objectives. Among the disadvantages of SaaS is the flexibility and the extension of the software is not guaranteed because the SaaS products it is defined and managed by the vendor.

Infrastructure as a Service

Infrastructure as a service IaaS allows the users to define their software on it own, and use the infrastructure resources as a service in order to run the software. In another words, using IaaS is allocating resources from the cloud platform including virtual machines, CPU, memory and networking. Again, choosing such a service it depends the user needs and objectives. As an advantages when using this service is the flexibility for the software implementation that may be extended by the user.

Platform as a Service

Platform as a service PaaS is a building block of several services highly managed by the vendor, used all together in order to handle user's software in various levels such as networking, storage, integration and monitoring. It keep the user focusing only on customizing it solution. In our work we will use such a service that procures us the possibility to customize and extend our solution, and keeping the platform carrying on different layers of our application. We will talk more in details on the services that we have chosen in section Google cloud service.

3.1.1 Google Cloud Services

Google cloud provides more than 100 products within the Google cloud platform. The product could be used separately from the other or we can coupling many products together to serve user's requests. For example if we use compute engine as a service to perform some tasks, we may use Google monitoring service with it in order to views the resources utilisation in order to get visibility on the workloads performance. Table 3.1 is a map[1] of a non exhaustive list of Google cloud products features and services. Each GCP service contains a list of products that are fully managed by GCP teams. If we take an example of compute service, it have as products, Google Kubernetes Engine GKE used for deploying docker containers within GCP and compute engine is dedicated to use a virtual machine using GCP resources (CPU, memory). The storage service provides cloud storage, filestore storage, local SSD and persistent disk products that allow users provisions volumes to consumes or stores their data. In the scope of our project, we have used the compute and storage services to achieve our application requirements. Figure 3.1 illustrates the building block of the GCP products used in our solution's implementation. When creating GKE product, it will invoke automatically compute engine product to instantiate the virtual machines, in order to start and run the Kubernetes objects. In the same way GKE use cloud storage product that allows the volumes object of Kubernetes provision storages. The Cloud identity and access management (IAM) product is used to manages the user's identities and authentication processes before using GCP services. Finally the cloud monitoring product is used in our work to analyze the workload's performance and it gives us an overviews on the GCP resources utilization such CPU and memory. In the following section, we will introduce the GKE product in more details.

[1]Google cloud products, services and features `https://googlecloudcheatsheet.withgoogle.com/`

Service	Product	Description
Compute scalable VMs and containers	GKE	Managed Kubernetes/containers
	Bare Metal Solution	Hardware for specialized workloads
	Blockchain Node Engine	Fully managed blockchain node
	Shielded VMs	Hardned VMs
	Cloud Run	Serverless for containerized applications
	App Engine	Managed application platform
	Batch	Managed batch services
	Cloud function	Event-driven serverless function
	Sole-tenant nodes	Dedicated physical servers
	Compute Engine	VMs, TPUs, GPUs, Disks
	Preemptible VMs	Short-lived compute instances
	VMware Engine	VMware as a service
Storage short and long term	Storage Class	Multi-class multi-region, object storage
	Backup and DR service	Backup and disaster recovery SaaS
	Persistent Disk	Block storage for VMs
	Cloud Filestore	Managed NFS server
	Local SSD	VMs locally attached SSDs
Database relational and non-relational	Cloud Spanner	Horizontal scalable relational database
	Cloud SQL insights	SQL Inspector
	Database migration service	Migrate to Cloud SQL
	Cloud Firestore	Serverless NoSQL document DB
	AlowDB	Scalable performance PostgreSQL compatible DB
	Cloud SQL	Managed MySQL, PostgreSQL and SQL server.
	Cloud Bigtable	Peta-byte scale, low latency and non-relational
Identity and Security Policy and compliance tools	Cloud IAM	Resource access control
	Cloud Audit Logs	Audit trails
	Access context manager	Fine-grained and attribute based access-control
	Assured Workloads	Workload compliance controls
	Identity Platform	Drop-in Authentication and access management
Operations And Monitoring	CLoud Profiler	CPU and heap profiling
	Error Reporting	App error reporting
	CLoud Monitoring	Infrastructure and application monitoring
	Cloud Trace	Apps latency insights

Table 3.1: Few of GCP products by services

3.1.2 Google Kubernetes Engine

After performing our state of the art by analyzing the past researches on how to distributes PostgreSQL databases on the cloud. We have opted to use Kubernetes system that delivers more flexibility to extend and support our solution in the cloud. Kubernetes system is integrated as a product within GCP compute service under the name Google Kubernetes engine(GKE). As a fully managed product, GKE allows the users focusing

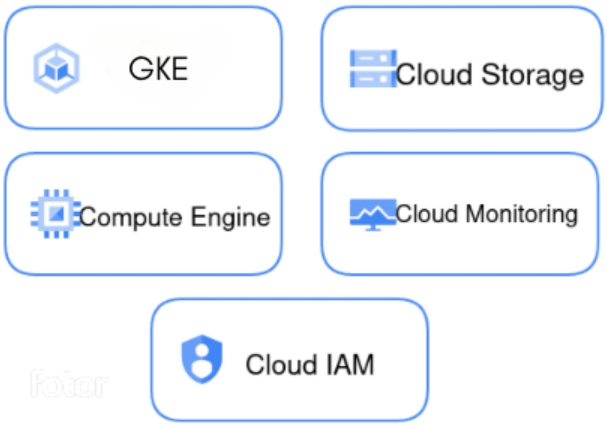

Figure 3.1: Building blocks of GCP products used for our solution

only on deploying and managing their application through the enabled features such as high availability, scalability, monitoring and logging. There are two kinds of GKE creation, auto-pilot and standard GKE. When creating the auto-pilot, many features will be activated automatically without configuring them such as auto-scaling that takes as initial number of nodes equal to one node by letting the auto-scaler adding the nodes automatically when a high of resources utilization on the workload occurs. The standard GKE allow us configures all the cluster initialization steps including the number of nodes of the cluster, the capacity of the machine in term of CPU cores, memory and the type of the disk, the network configuration and the auto-scaler capabilities. In our work we have used the standard GKE to have the possibility to design our own cluster with the desired initial number of nodes without enabling the auto-scaler.

In following, are the benefits of using GKE product.

- **Security and identity** GKE is integrated with the IAM service allowing to control the accesses into the cluster and provide a private cluster and several network policies.
- **Monitoring and logging** GKE is integrated with the GCP's monitoring and logging module that provides a comprehensible monitoring to gives an insight on the resource utilization and alerting the user with the metrics rules definitions.
- **Scalability and Elasticity** GKE makes the horizontal scaling and vertical scaling easily according to the workload and resources consumption.
- **Seamless Integration with GCP services** When using GKE is it possible to integrates other GCP products such as BigQuery, Cloud Spanner and other fully managed products.

3.2 PostgreSQL Engine

PostgreSQL [18] is an object-relational database management system (ORDBMS) developed at the University of California at Berkeley Computer Science Department. PostgreSQL is widely used by the organizations that allows them storing, managing and interrogating their data in a structured manner. Is designed to support SQL standardization that make it easy for users to querying data through SQL syntax. And it is known as a robust ACID (Atomicity, Consistency, Isolation, Durability) compliant where the data integrity is guaranteed for transactional and analytical databases.

PostgreSQL is highly extendable where the users can add their own data type, function, operators and procedures to makes the server specialized for a specific domain, for instance supporting moving object data. Because it is an open source solution, PostgreSQL have an active community from companies and individuals that contributes to its development and making it run in divers environments and operations systems. In the following sections, we will present the PostgreSQL extensions used in our work starting by PostGIS that extends PostgreSQL operators and functions in order to support geo-spatial data and then MobilityDB extension that add functionalities and new data types in order to manages moving object databases. Finally we introduce the Citus data extension that distributes both PostgreSQL tables and queries across a cluster of machines. This features makes PostgreSQL scalable at any scale, especially when we have analytical applications that interrogate the big tables.

3.2.1 MobilityDB Extension for PostgreSQL

MobilityDB is a database management system for moving object trajectories, such as GPS traces. It adds support for temporal and spatial-temporal objects to the PostgreSQL database and its spatial extension PostGIS.[2] MobilityDB extends PostgreSQL by defining new data types on top of PostgreSQL and PostGIS types, where spatial-temporal types for moving geometry and geography points is added, as well as for temporal integer, real, Boolean, and string. Zimanyi et al. [44] add also temporal data types including tgeompoint, tgeogpoint, tfloat, tint, ttext, and tbool. Furthermore, the types are supported with spatial-temporal index access methods by extending GiST (Generalized Search Tree) and SP-GiST (Space Partitioning GiST). Zimanyi et al. [42] explain in details the MobilityDB data types implementation, as a fully integrated abstract type for representing moving object database. Several researches have been elaborated in behind MobilityDB extension, [47, 32, 38]. In addition, the development team participate in relevant conferences and provides workshops to introduces and motivates the new features developed on top of PostgreSQL in MobilityDB, [34, 43, 8]. The abstract data types provided, are standardized as PostgreSQL data types that are useful to manipulates moving object queries, aggregations and indexing easily. The suite of MobilityDB development and contributions is updated in the Github repository.[3]

MobilityDB Data Types

MobilityDB extends the capabilities of PostgreSQL engine by adding new data type to handle spatial and spatial-temporal databases. Such data types represent the evolution over time of certain data type such as integer, boolean or string, those type will be associated with temporal type. For instance, temporal integer may be used to represent the evolution over time of the number of employees for such a department. In this case, the data type is temporal integer and the base type is integer. Similarly, a temporal float may be used to represent the evolution over time of the temperature within a room. As another example, a temporal point may be used to represent the evolution over time of the location of a car, as reported by GPS devices. Zimanyi provides definitions and explanations for these new data types in [45]. Table 3.2 highlight MobilityDB temporal data type.

[2] `http://postgis.net`
[3] `https://github.com/MobilityDB`

Data type	Description
tbool	Based on PostgreSQL bool data type
tint	Based on PostgreSQL int data type
tfloat	Based on PostgreSQL float data type
ttext	Based on PostgreSQL text data type
tgeompoint	Based on PostgreSQL geometry data type
tgeompoint	Based on PostgreSQL geography data type
period	represents a range of time between two instants. It consists of a start timestamp and an end timestamp and is often used to represent durations.
timestampset	It can be used to store a collection of instants or timestamps.
periodset	It can be used to store a collection of period.
timestamptzset	It represents a set of timestamp ranges. It can be used to store a collection of non-overlapping periods.

Table 3.2: MobilityDB temporal data types

Realistic MobilityDB Use Case

In this part we will motivates MobilityDB uses by introducing a concrete use case with a realistic example, that is already done on the MobilityDB workshop with BerlinMOD benchmark [46]. In this short MobilityDB scenario, we assume that MobilityDB and PostGIS are installed with the PostgreSQL server. In order to use MobilityDB data types and features, we need to create the MobilityDB and PostGIS extensions with the following SQL code:
`CREATE EXTENSION MobilityDB CASCADE;` And then create the BerlinMOD schema with the new data type supported by MobilityDB in order to querying the database. Listing 3.1 highlight the schema of BerlinMOD benchmark model.

```
1  CREATE TABLE Vehicles(vehId int, licence text, type text,
2      model text);
3  CREATE TABLE Points(pointId int, PosX float, PosY float,
4      geom geometry(Point));
5  CREATE TABLE Regions(regionId int, geom geometry(Polygon));
6  CREATE TABLE Instants(instantId int, instant timestamptz);
7  CREATE TABLE Periods(periodId int, BeginP TimestampTz,
8      EndP TimestampTz, period tstzspan);
9  CREATE TABLE Trips(tripId SERIAL, vehId int, day date,
10     seqNo int, sourceNode bigint, targetNode bigint,
11     trip tgeompoint, trajectory geometry);
12 CREATE TABLE Licences(licenceId int, licence text, vehId int);
```

Listing 3.1: BerlinMOD schema creation

Assuming the tables are fulfilled with some data following the MobilityDB workshop [46]. The data is generated for Brussels city, in order to explore the analysis of their trajecto-

ries, tracking vehicles and computing the duration of the trips and performing additional analysis. Listing 3.2 query computes some statistics on trips table including the total duration of the trip and the total length in Km for all the trips.

```
SELECT MIN(timespan(Trip)), MAX(timespan(Trip)),
AVG(timespan(Trip)) FROM Trips;
-- "00:00:29.091033" "01:13:21.225514" "00:22:02.365486"
```

Listing 3.2: Statistics query

Listing 3.3 show another query that allow us to extract the longest trip within the city, by filtering all trajectory above 50 Kilometers.

```
SELECT vehicle, seq, source, target, round(length(Trip)::numeric
    / 1e3, 3), startTimestamp(Trip), timespan(Trip)
FROM Trips
WHERE length(Trip) > 50000 LIMIT 1;
-- 90 1 23078 11985 53.766 "2020-06-01 08:46:55.487+02"
--  "01:10:10.549413"
```

Listing 3.3: The longest trip query

The visualization of the query results as highlighted in the workshop [46], gives more comprehension on the benefits when using MobilityDB on top of PostgreSQL by customizing it to handle spatial and spatial-temporal databases, and allow the trajectories analysis efficiently. Figure 3.2 shows the longest trip within the city of Brussels with the highlight of the starting point (home node) and ending point (work node) of this trips using QGIS.[4]tool. MobilityDB was combined with PostgreSQL Citus extension in order

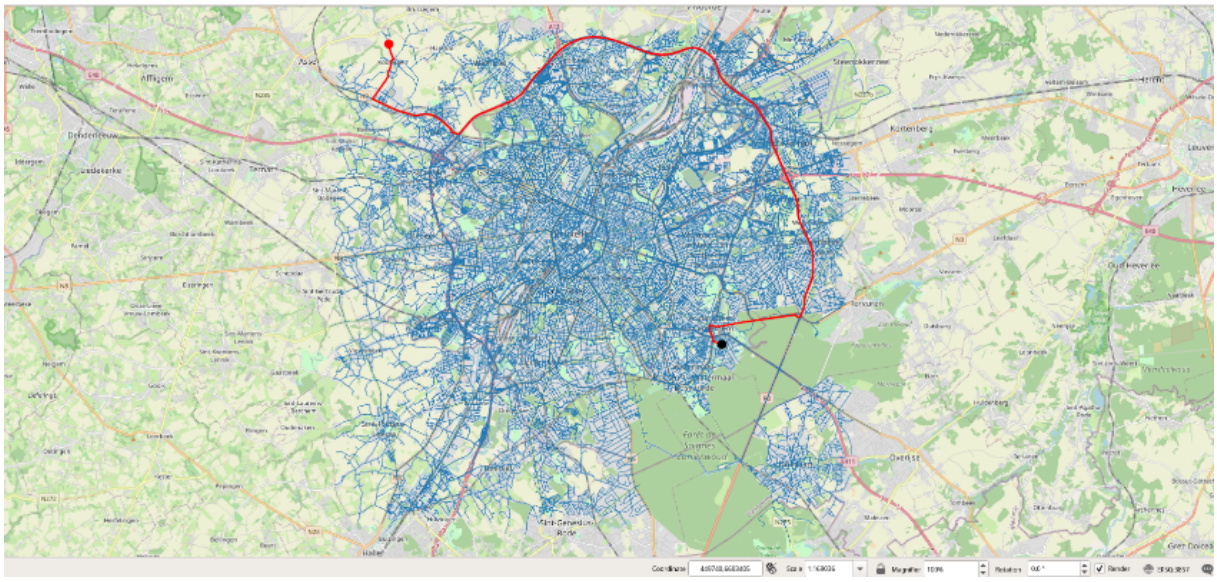

Figure 3.2: Visualization of the longest trips within Brussels marked by the home and work node

to distributed the complex queries that produces bottleneck on the resources when the data grows. Bakli et al. [44] benchmarks the Citus extension on top of MobilityDB in order to know until which levels, Citus can helps on the execution of complex MobilityDB queries and then test the capabilities of Citus extension on a cluster of machines through the BerlinMOD [46] benchmarks.

[4]QGIS is used to handle Geographic Information System: `https://qgis.org/en/site/index.html`

3.2.2 Citus Data Extension for PostgreSQL

Citus[5] is an open source extension for PostgreSQL that enables horizontal scaling of PostgreSQL databases across multiple machines, considered as worker nodes. It designed to conducts companies gaining high performance for analytical and real-time workloads. Citus allows scaling out PostgreSQL database beyond a single server by partitioning the data into a set of shards across multiple nodes. Where the partitioning method it may be configured on the coordinator node that stores the metadata of the cluster. Each node in the Citus cluster contains a group of shards that represents the distributed data. Once the data is distributed, the queries are automatically parallelized across the nodes to take advantage of the available computing resources. Umur Cubukcu et al. [12] demonstrate the performance and the scalability of Citus cluster on different workload patterns where each pattern was tested using Citus extension at scale. Useful information is used from the documentation of the Citus extension for PostgreSQL [25]. Citus data recently has been acquired by Microsoft as revealed in [23].

Query Processing Using Citus Extension

A Citus cluster is composed of both a coordinator instance and multiple worker instances. Within this setup, data is distributed across the worker instances through sharding, while the coordinator instance maintains essential metadata associated with these shards. Every query directed to the cluster is executed through the coordinator. The coordinator then divides the query into smaller fragments, enabling each fragment to be independently executed on a shard within the worker nodes. This process ensures local execution on each worker node. Subsequently, the coordinator allocates the query fragments to the respective worker nodes, supervises their execution, merges their results, and returning back the final result to the user. Figure 3.3 illustrates the query processing architecture.

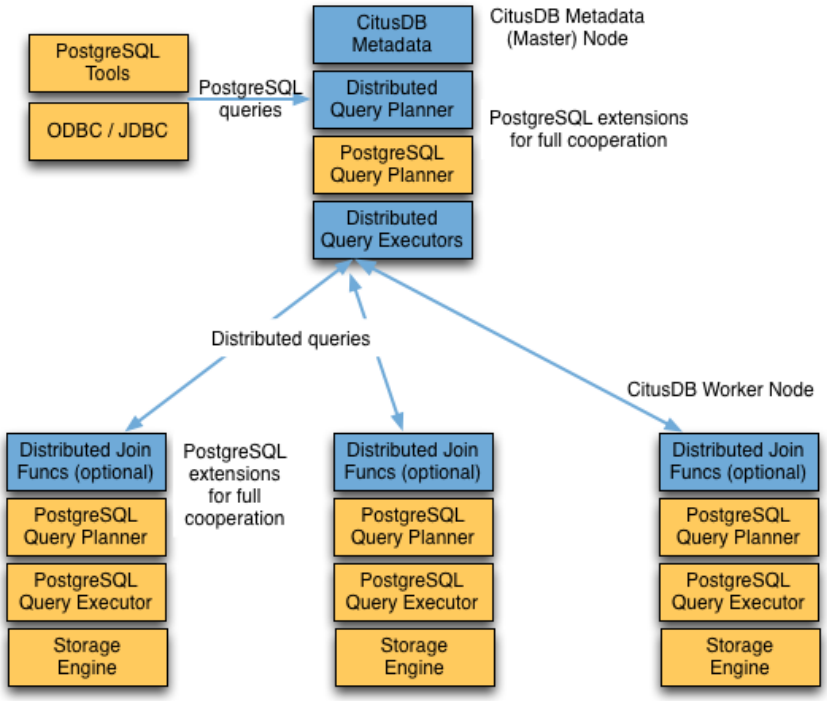

Figure 3.3: Query processing architecture with Citus operator [24]

[5] https://docs.citusdata.com/en/v11.3/get_started/what_is_citus.html

Table and Query Distribution Using Citus Extension

Citus operator tackle the performance issues through the partitioning column and the hashing algorithm that create a distributed shards equally as much as possible over all the cluster nodes. This technique is similar to create an index for such a table, not only on single server but on a whole group of servers this time. As a best rule to insert new records efficiently over the distributed database, is to choose the adequate distributed column in order to route the new record into the right node and then into the right shard within this node based on it hash value.

3.3 Docker Engine

Docker technology is gaining a lot of popularity in recent years through a series of advantages that help overcome the problems of developers and system engineers. Docker has contributed considerably to the development of applications, platforms and other technological tools, but also to several phases of the implementation of an IT project, in particular the development, deployment, testing and maintainability phases of system life cycle. Docker containers are the lightweight, portable software packages that contain all of the code, libraries, and system tools required to run an application. Among the advantages of the Docker system described in [20], few of them are as following:

- It ensures the separability between hardware and software which allows the portability of the software in order to install it in different environments.
- It ensures the extensibility of technologies in an easy way by giving developers the flexibility to add functionality layer by layer according to the requirements.
- It allows rapid and efficient deployment through a formal language that allows the infrastructure to be written in lines of code.
- Docker makes it easy to maintain and update code and application functionality, this requires developers to change only the lines of the correct infrastructure code and do a new build, this avoids removing and referencing a new deployment again. Additionally, application and deployment testing, becomes easier by using a container that allows the developer to do more test iterations than before.

These are the steps that generally takes place, when using Docker containers as mentioned in [5]:

- Establish a Dockerfile: a text file called a Dockerfile contains instructions for creating a Docker image. It details the setup, dependencies, base image, and application code.
- Create the Docker image: create the Docker image from the Dockerfile using the Docker CLI. A snapshot of the application and its surroundings is made throughout this procedure. This procedure runs the program in a separate setting.
- Interact with the Container: once the container is up and running, you may use a variety of Docker commands to communicate with it, view its logs, and control its life cycle.

Concerning our application, we used Docker container in order to containerize all the necessary functionalities in order to support a moving object database as well as the prerequisites which consist in partitioning the data and distributing them within a group

of nodes. These features include the PostgreSQL database server and their extensions, PostGIS, MobilityDB and Citus.

3.4 Kubernetes Ecosystem

The Kubernetes system or also called K8s as described in the official documentation [22] plays a fundamental role in containerized applications deployed as micro-services, it allows the orchestration of several containers in order to automate deployment tasks and the evolution of the application over time. The Kubernetes system is an open source system that is flexible for any kind of application, regardless of the complexity and the needs of these applications. Designer to be a lever for the development of evolving applications in terms of technologies, business extensions, operation, monitoring and maintenance. Kubernetes offers the ability to work in on-premises, hybrid or in public cloud service. Among the capabilities and responsibilities of the Kubernetes system are the following:

- **Service discovery and load-balancing:** Kubernetes exposes the application as a service through the IP address automatically generated by K8s objects and the DNS name configured before. In addition, K8s provides the load balancing service which distributes customer requests in the event of heavy traffic arriving in a group of instances of the current application. Application instances are instantiated by the Pod object, which we will discuss in detail in the Kubernetes components section.
- **Storage orchestration:** Kubernetes allows us to automatically mount a storage system to acquire application data, this is supported by local storage, storage from public cloud providers like AWS or GCP, or a network storage system such as NFS.
- **Automated rollouts and rollbacks:** we can describe the desired state for our containers deployed using Kubernetes in the application extension case, and it can change the current state to the desired state at a controlled pace. For example, we can automate Kubernetes jobs to add new containers for our deployment, remove existing containers, and adopt all their resources into the new container. Restoration to previous states is guaranteed in case of disaster through K8s self-health. This feature allows us to develop our application without any risk of having a downtime or crashing.
- **Self healing:** the monitoring and maintenance mechanisms of K8s allow us to perform self-healing in particular, the recreation of dead containers or the restarting of containers that crash, the migration of containers to a healthy node in the event of the node's death.
- **Horizontal scaling:** K8s provides horizontal auto scalability which allows resources to be added automatically through the addition of new instances or Pods, in order to reduce the load on the workload. The auto-scaling is operated by a controller which runs periodically in the control plane node in order to monitor the use of resources, CPU, memory or other if they exceed certain thresholds (defined beforehand by the user) in order to increase or scale-down in case of idle workloads.
- **Designed for extensibility:** Kubernetes is very extensible and configurable. Therefore, there is rarely a need to fork or submit patches to Kubernetes project code. Configuration, which simply involves modifying command-line arguments, local configuration files, or API resources, and Extensions, which involves running additional programs, additional network services, or adding more containers by coupling them

with the existing containers. These possibilities allow a personalizing the application to make it expand in an easy way without compromise.

3.4.1 Kubernetes Components

A deployment within a Kubernetes system is ensured by a group of components in order to guarantee the capabilities and responsibilities mentioned in the previous part, A deployment is represented by a cluster of worker nodes which are managed and manipulated by the control node called control plane which includes the K8s API and different agents that control each instance within the cluster as highlighted in [5]. Figure 3.4 gives a high level overview the architecture of the K8s system with its different components. The

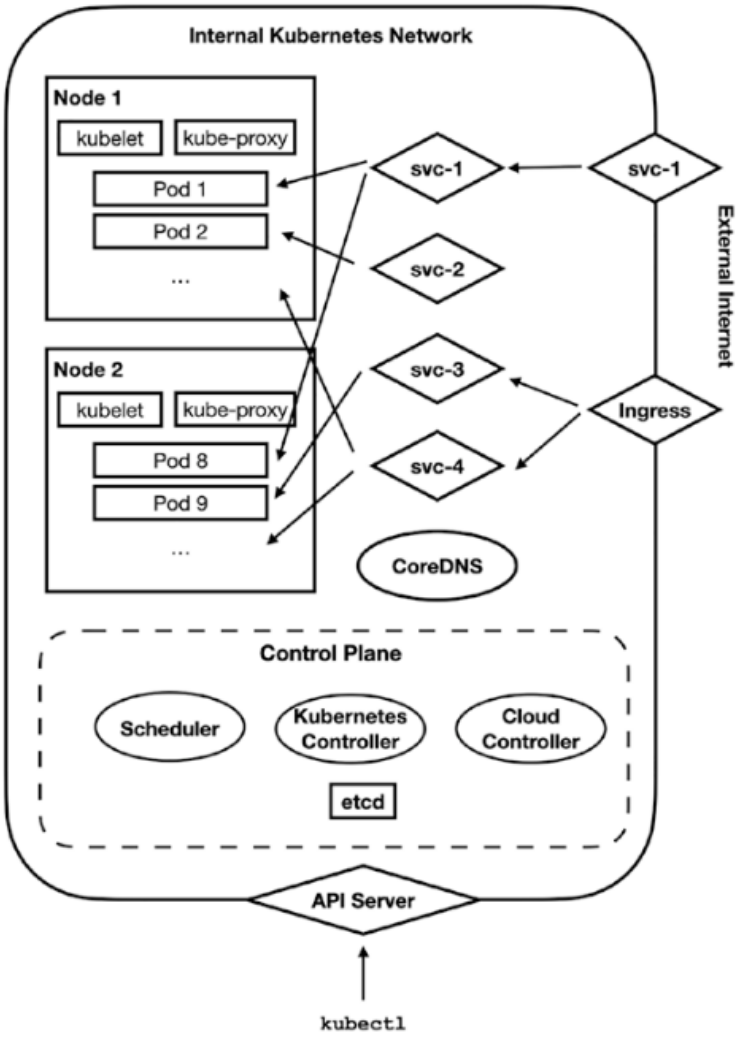

Figure 3.4: Kubernetes architecture [5]

worker nodes have the role of running the Pods on top of containerized applications by providing the infrastructure layer of the cloud platform in terms of CPUs, memories and local disk space. In following we will briefly define the role of each K8s component.

Control Plane Components

A control plane plays the role as decision makers within the Kubernetes cluster. Here, all choices regarding the cluster are made. For instance what is the current state of the cluster now? Should we add more worker nodes? Does a deployment require the launch of

additional Pods? The control plane is in charge of making those kinds of choices. Among the main components of the control plane are the following:

- **kube-apiserver:** the API server is the implementation of the Kubernetes system that allows us to create a K8s cluster through command line interactions by the K8s Kubectl client. The K8s API server is designed to scale the cluster horizontally, where you can create multiple API servers and then send traffic to them from the external Kubernetes environment.
- **etcd:** it is a crucial element in the control plane that stores the history of the state of the cluster in order to makes possible the backup of the parameters and the configurations which were in the control plane. The duplication of the etcp makes it possible to ensure high availability of the cluster.
- **Cloud controller manager:** the cloud manager controller is used to interact with the cloud platform (such as AWS, Azure or GCP) via the API of these platforms in order to use the functionalities of K8s within these platforms. The advantage of the cloud controller manager is to support the cloud environment existing in the market, in order to provide the K8s with portability and adapting it to several cloud platforms. The cloud controller makes it possible to control the resources of the cloud platform such as the nodes, in order to determine whether the node is still running or not, if the node does not respond, then the cloud controller sends a restart signal in order to ensure the availability. Similar for the services offered by the cloud platform, the cloud controller controls with a loop which rotates the state of the services in order to increase the availability to the load-balancer which routes the traffic to the available services.
- **kube-scheduler:** the scheduler determines what should be done. The scheduler determines when and which Nodes are appropriate for Pod deployment. Component of the control plane that searches for newest Pods that have not yet been given a node and chooses one for them to run on. Individual and group resource needs, hardware/software/policy restrictions, affinity and anti-affinity standards, data locality, inter-workload interference, and deadlines are all taken into consideration while making scheduling decisions.

Node Components

- **Kubelet:** is an agent that is installed on each cluster node. It confirms that containers are operating within a healthy Pod. The kubelet checks that the containers described in a set of PodSpecs, delivered through various mechanisms are active and in good condition. Containers that weren't built by Kubernetes are not managed by the kubelet.
- **kube-proxy:** every node in your cluster runs kube-proxy, a network proxy that executes a portion of the Kubernetes Service concept. On nodes, kube-proxy keeps track of network policies. These network rules, permit communication to the Pods from sessions, both inside and outside of the K8s cluster.
- **Container runtime:** the program in charge of operating containers is known as the container runtime. Container runtimes like containerd or container daemon, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface) are defined in Kubernetes.

3.4.2 Kubernetes Objects

The start of an application within the Kubernetes ecosystem [22] is accompanied by several instances or objects of K8s, which become part of the new application deployed, all together the objects will be harmonized and linked together to build a K8s cluster. Each K8s object has a life-cycle and has its own role in the cluster. These objects can be classified into several roles such as, controller objects which control other objects within the cluster to ensure availability and zero downtime, objects which represent physical resources (node object) in order to facilitate configuration and increasing the capacity of these resources if needed. There are deployment objects `Pod`, `Deployment`, `StatefulSet` which make it possible to instantiate the application containers as well as the objects which represent the storage to supplying the application. In the rest of this part, a brief definition of the main objects used in our work is given.

Node and Pod

A node is a virtual or physical machine that makes the docker container abstraction of the application called Pod work. Each node is managed by the plane control. Regarding the Pod object, it is the smallest entity in a K8s cluster, it includes one or more containers that share the same storage layer, network and resources. Containers within a Pod are called co-located containers. In an operational node, it can be either one or more Pods paired together to provide the appropriate workload for the desired application.

Service and Ingress

In the Kubernetes system, the service object is a method that allows us to expose the application that works in the Pods, the service is a key element that allows you to keep the same access parameters to the application whether it is IP address or port, even if there is a change at the Pod level or an update of the application code within a container. This remains unchanged on the client side who accesses the application, always ensures a fixed service. In other words, If for some reason the Pod die, a new one is created and a new IP address is assigned to it and the old IP address is gone. The exposure IP address remains the same for the service on front of the client. The Ingress is an API object that manages the external connections received from client through internet, especially HTTP request. This object is considered to play the same role as the service object but between the outside of the application's private network and the inside. An important role of the Ingress object is load-balancing, which allows traffic from client requests to be routed to the appropriate service.

ConfigMap and Secret

`ConfigMap` is a K8s object that allows storing application meta-data in peer-key-value form, ConfigMap data is generally not confidential. Application Pods use this configuration information as global or so-called environment variables, or as command-line arguments as well as used to specify links to volume files attached to the application. `Secret` is a K8s object which has the same role as ConfigMap but to specify confidential data which can be for identification such as logins, passwords, keys and certificates defined in an encrypted manner. The purpose of ConfigMap and Secret is the easy portability of the configuration parameters if we want to migrate our application to another cloud platform, this is possible by separating the environment variables from the container images in ConfigMap and Secret.

Volume

The Volume object of K8s is the fundamental element for applications that persist their data in the disk, because the volume within the container is ephemeral which poses loss of data and states if the Pod is restarted. In addition, for containers that share the same volume support, it is difficult to manage access for all containers. Like other K8s objects, the Volume object allows data separability and portability when migrating from one platform to another. The volume in K8s can support multiple storage types including AWS, Azure and GCP storage classes.

Deployment and StatefulSet

We always have to replicate the Pod to avoid user downtime and provides high availability. Generally Pods with multiple replicates are exposed in a single service. If the service receives a request from the client, the service sees which Pod is available for, or less busy in order to forward the request to it (load-balancing). Replication must therefore be specified in the `deployment` component. In reality the application is not instantiated by a Pod, but by the deployment object that enables us to precise how many replicas of the Pod we must creates. If we want to preserve the states and the data of the application after modifications or new insertions or even after a breakdown or crash, we need to create our application in `StatefulSet` mode accompanied with a volume object in order to persist the data in the disk. StatefulSet is a mechanism that can save the state of data. Because if we duplicate the Pod which concerns a database application for example, we risk losing the consistency of the data in the case of a scale-out or scale-in. StatefulSet mechanisms guarantee us horizontal scaling without losing data as well as readings and writings on the database are guaranteed without data inconsistencies. We say that the Pods are load balanced, this allows us to have a robust application if the number of replicas is high. Figure 3.5 show the relationship between the K8s objects for a given application. Deploying PostgreSQL server as Kubernetes deployment with the required

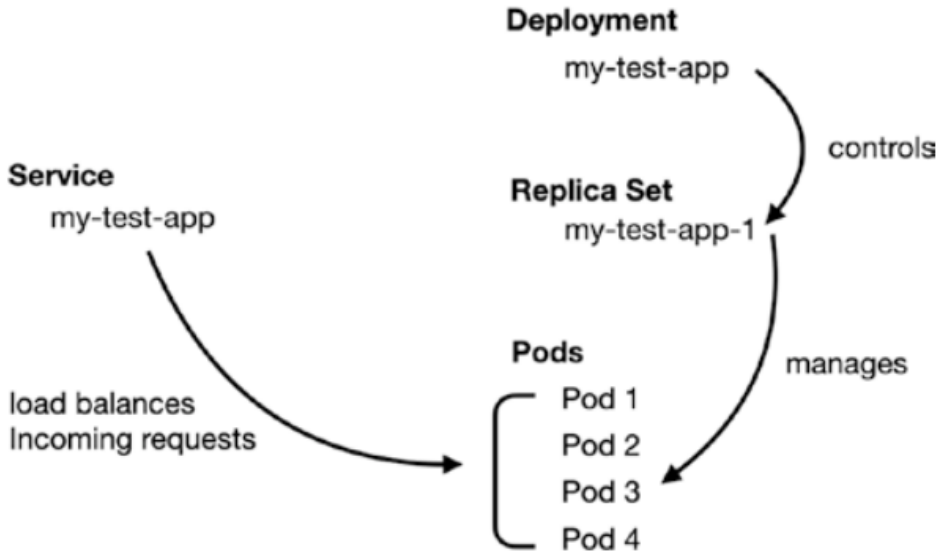

Figure 3.5: Kubernetes cluster object [5]

Kubernetes objects are described in [39] in the part creating PostgreSQL cluster.

3.4.3 Minikube

As the Cloud services is very costly. The billing is calculated depending on resources used per second passed. So to avoid using cloud resources during our analysis part, we have used Minikube[6] allows us to create a cluster in our local machine, that helps to analyse and prepare our infrastructure needed to achieve the objectives of our work. Finally when all the architecture is tested and validated locally in our machine, we deployed our work in Google Cloud services, in order to perform our experiments. Minikube is helpful for students or researchers that want to do experiments on Cloud services.

3.5 Python Client

Python language has become the most used language in the world, which keeps it first place until now in the top 10 of the best programming languages according to the index provided by TIOBE[7] company, specialist in the measurement of the quality of the programming language across the analysis of more than one billion lines of code for their customers' software worldwide, in real time. Python language is growing in popularity compared to other languages for several reasons such as simplicity, readability and the large number of libraries and frameworks that makes it attractive to developers and companies. As well as it has a large community which contributes continuously to make it more suitable in different fields of application including web development, data science, machine learning, automation, cloud computing and more. Python is weakly typed through the dynamic typing, it offers the garbage collector. It supports several programming paradigms including structured, semi-structured or procedural, as it supports object oriented and functional programming.

When it comes to our application, we used Python code in several phases of our project, in particular for the interaction with the GKE product of Google cloud platform in order to initialize the Kubernetes cluster, automate the deployment of the PostgreSQL server, change the size of the Kubernetes cluster and the manipulation of cluster objects and resources in order to respond to user requests. In addition, the tests of our solution as well as the visualization of the graphs are generated by a Python script.

In the following of part, we introduce the Python tools and framework used to complete our implementation part where we start with the GCP client library which allows interaction with GCP services and to create APIs which manipulate the products. (like our case the GKE) remotely in order to automate certain tasks. Then we introduce the Python client of Kubernetes which allows interaction with the functionalities and objects of Kubernetes instance, in order to automate the tasks of scale-out and scale-in of the Citus cluster. At the end we briefly introduce the visualization tool used in our test and evaluation part.

3.5.1 Python Client for Google API

Python client for GCP products[8] is an API written entirely in Python that allows us to build, run, manage and manipulate instances created in the GCP platform. This API includes a suite of features organized by GCP service which have the same role as if we manipulates GCP through the interface or from the command line. So the Python client

[6] `https://minikube.sigs.k8s.io/docs/start/`

[7] `https://www.tiobe.com/tiobe-index/`

[8] `https://github.com/googleapis/google-cloud-python`

for GCP is another way to interact with GCP resources that we can design as a client-side application to commands and controls the GCP infrastructure. In our case, we used the Python client to manage and manipulate the GKE cluster in order to automate the steps of initialization, installation of prerequisites and creation of the cluster for the user efficiently. The access methods to the GCP platform provided by the Python client as described in [28] have always the same structure and hierarchy. The piece of code shown in Listing 3.4, show an example of classes necessary to launch a GCP command, more precisely to retrieve information from a GKE cluster that is already initialized in the platform.

```
from google.cloud import container_v1
def sample_get_cluster():
    # Create a client
    client = container_v1.ClusterManagerClient()
    # Initialize request argument(s)
    request = container_v1.GetClusterRequest(project_id=None,
    zone=None, cluster_id=None, name=None)
    # Make the request
    response = client.get_cluster(request=request)
    # Handle the response
    print(response)
```

Listing 3.4: Get cluster information method

As we can see get_cluster method in Listing 3.4 return the information of the cluster, it derived from the `container_v1` which is a stack of features needed to interact with GKE cluster. This method takes as input mandatory fields including, `project_id` that represent the project number, the `zone` that represent the locality of the cluster and the `cluster_id` for the identifier of the cluster and the `name` as an optional field that represent the name of the cluster.

3.5.2 Python Client for Kubernetes API

Regarding Python client for Kubernetes,[9], is a package that includes several features that allow you to manage and control the resources of a Kubernetes cluster whether in the cloud or locally. The goal of using this type of library in our code, is to generalize our API and allow it to work in the three cloud including GCP, Azure and AWS and other platforms. By using this Kubernetes Python client API, we can customize our own API to automate the repetitive tasks that are necessary performed after each change in the size of the Citus cluster, whether for the scale-in or scale-out operation. As in the Python client of GCP, the Python client of Kubernetes is structured in hierarchy, so you must follow the same format proposed in the documentation [10] which is used to manipulate the application and deployment objects as well as for the documentation [11] which is used to extend Kubernetes systems. Listing 3.5 consist of getting all the existing Pods in the Kubernetes cluster, where the `kubernetes.client` package is composed of a all the features needed to interact with the K8s objects. The `configuration` class in line three from Listing 3.5 consist of capturing the meta-information of the K8s cluster from the local machine file `kube-config`. With those configuration information, the `APIClient` is instantiated in order to run the call of the existing Pods through the method `list_pod_for_all_namespaces`.

[9] `https://github.com/kubernetes-client/python`

```
import kubernetes.client
from kubernetes.client.rest import ApiException
configuration = kubernetes.client.Configuration()

# Defining host is optional and default to http://localhost
configuration.host = "http://localhost"
# Enter a context with an instance of the API kubernetes.client
with kubernetes.client.ApiClient(configuration) as api_client:
    # Create an instance of the API class
    api_instance = kubernetes.client.CoreV1Api(api_client)
    try:
        api_response = api_instance.list_pod_for\
        _all_namespaces(watch=watch)
    except ApiException as e:
        print("Exception when calling CoreV1Api->
        list_pod_for_all_namespaces: %s\n" % e)
```

Listing 3.5: Get the list of existing Pods

3.5.3 Dash Plotly for Visualization

Dash Plotly is an open source framework that provides a suite of features that aims to visualize data in an interactive way with high quality. Plotly's graphics library is written in several programming languages including Python, R Julia and MATLAB. This makes it more attractive to developers. With Graphic Plotly for Python, we can represent our datasets on a variety of graphs and charts including line plots, scatter plots, area charts, bar charts, error bars, box plots, histograms, heatmaps, etc. The interactions provided by Python's Plotly framework allows data to be viewed from multiple facets and different angles. In our work, we used this tool to generate our graphs for the possible tests carried out in the Experiments and Performance Evaluation part. Among the keys capabilities of Dash Plotly tool:

- **Dynamic Visualizations:** Plotly is known for its interactive and visually appealing charts, we note line charts, scatter plots, bar charts, pie charts, 3D plots, maps, and more. These visualizations can be easily embedded into web applications built with Dash.
- **Dash Layouts:** Dash provides a set of features for arranging and composing visualizations into a layout. It allows the combination of HTML components and Dash core components to create a structured web application.
- **Callbacks and Interactivity:** is one of the powerful features of Dash,it gives the ability to define callbacks. Callbacks allows us to update the content of web page based on user interactions, like clicking on a plot or selecting options from a dropdown.
- **Data Table Support:** Dash supports interactive data tables that can be filtered, sorted, and searched, making it easier for users to explore data directly in the application.
- **Dash Components:** Dash provides a wide range of components, basic and advanced, to build interactive UI elements such as sliders, dropdowns, date pickers, graphs, maps, etc.

- **Large datasets support:** Plotly supports large datasets and can handle streaming data updates in the Dash application through the connected API.
- **Community and Documentation:** Dash Plotly tool have an active community, and there are plenty of examples, tutorials, and documentation available to help users get started and solve common issues.

In this chapter, we have defined the different technologies used in the deployment and implementation of our solution, notably the Docker engine, which enables the isolation of a program from the hardware layer. This also facilitates application portability and deployment. We have observed that through the Kubernetes system, we can orchestrate a set of containers, and ensuring high availability, load-balancing, self-health checks, and scalable deployment within a machine cluster. Towards the end, we have introduced the Python client for GCP and Kubernetes, which will subsequently allow us to remotely manage our GCP platform, and more precisely, the Kubernetes cluster created through the GKE service. The visualization tool Dash Plotly is briefly introduced, this will enable us to generate graphs and charts to evaluate our experiments later on.

Chapter 4

Distributed Database Management on Google Kubernetes Engine

In this chapter we will present the methodological approaches adopted in order to deploy a distributed geo-spatial and spatial-temporal database on Google Cloud Platform. We will explain how did we use the technical background to achieve the target objectives. As we have seen from the comparison part in the state of the art, Kubernetes system gives more advantages comparing the managed services proposed by the cloud providers. The most important pros, is the flexibility to install a customized features with Docker images and the portability of the deployed workloads from Cloud platform to another. The existing Kubernetes system in Google cloud platform called Google Kubernetes Engine (GKE) as a service. In the following sections, we will present Kubernetes cluster management on GKE service and it requirements and then we describe the distributed workload management using Citus extension for PostgreSQL.

4.1 GKE Cluster Management

In this section, we describe the necessary elements to construct our cluster using the Google Kubernetes Engine service offered in GCP. Initializing this latter will allocate the hardware resources to establish our infrastructure that supports our distributed system within a designated region. This can be achieved through Google commands using the `gcloud` client, allowing the customization and specification of the cluster including the number of nodes in the cluster, the desired machine type, as well as the amount of CPU cores and memory used for each machine.

Native Cloud PostgreSQL for Distributed Database

Kubernetes system is the orchestrator of several containers where each container contain an isolated program from the hardware. As a starting point for our cloud native distributed database solution, we have prepared a single container image that contain all necessary requirements as shown in Figure 4.1 that shows the wrapping of all features needed to support our distributed workload in Google Kubernetes Engine (GKE) From PostgreSQL and PostGIS images, we added MobilityDB's features and it required geo-spatial libraries. On top of that, Citus extension and its libraries requirements is added to ensure the data partitioning and query distribution over the cluster. The advantage of the prepared image is open source and is it possible to deploy it on any cloud platform.

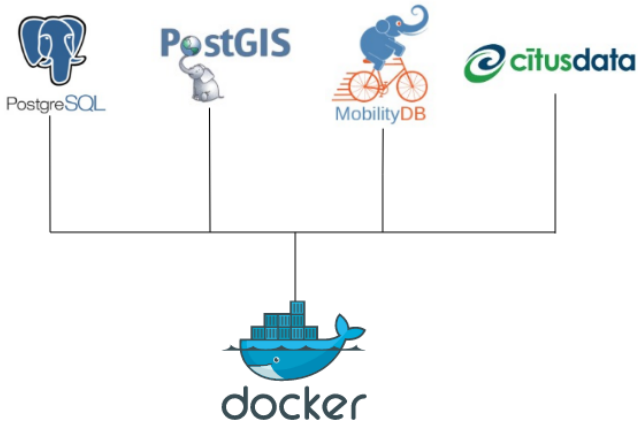

Figure 4.1: `mobilitydb-cloud` container image for distributing database on cloud

The image is available on the Docker hub.[1]

GKE Service Requirement

Before starting using Google cloud services, we need to enable them from the GCP console. Concerning our needs, enabling Google Kubernetes Engine service is necessary to host our distributed environment on it.[2]

In addition, we need to open a project and associating it to a billing account that have have credits on it, otherwise we cannot use GKE service. In our case, in the scope of this work, we have used Google cloud educational credits.

Among the recommendation that need to be installed, is the `gcloud` client that allows us creating and managing Google cloud services and resources from distance using our local machine through commands line. The client `gcloud` is used to manipulates GCP services and products and specifying its using the adequate options for each product. Another useful client tool that allows us to communicates with our infrastructure and Kubernetes system is the `kubectl` client. Kubectl enable managing Kubernetes cluster from our local machine.

GKE Cluster Initialization

Before deploying our cloud native image, we need to initialize our GKE cluster using `gcloud` command line and specifying all it arguments according to the cluster specifications. To interact with the GKE cluster, we use `gcloud container` command that permit to create, edit and delete cluster. Listing 4.1 describes the minimum of parameters needed to create GKE cluster.

```
1
2 gcloud container clusters create mobilitydb-cluster --zone
3 europe-west1-c --node-pool mobilitydb-pool-config --machine-type
4 e2-standard-4 --disk-type pd-balanced --num-nodes 4
```

Listing 4.1: GKE cluster initialization

This command will create us a GKE cluster in the zone `europe-west1-c` with size of `4 nodes`, where each node have as specification `e2-standard-4` of machine type and

[1] `https://hub.docker.com/repository/docker/bouzouidja/mobilitydb-cloud/general`
[2] `https://console.cloud.google.com/apis/enableflow?apiid=container.googleapis.com`

`pd-balanced` as disk type.

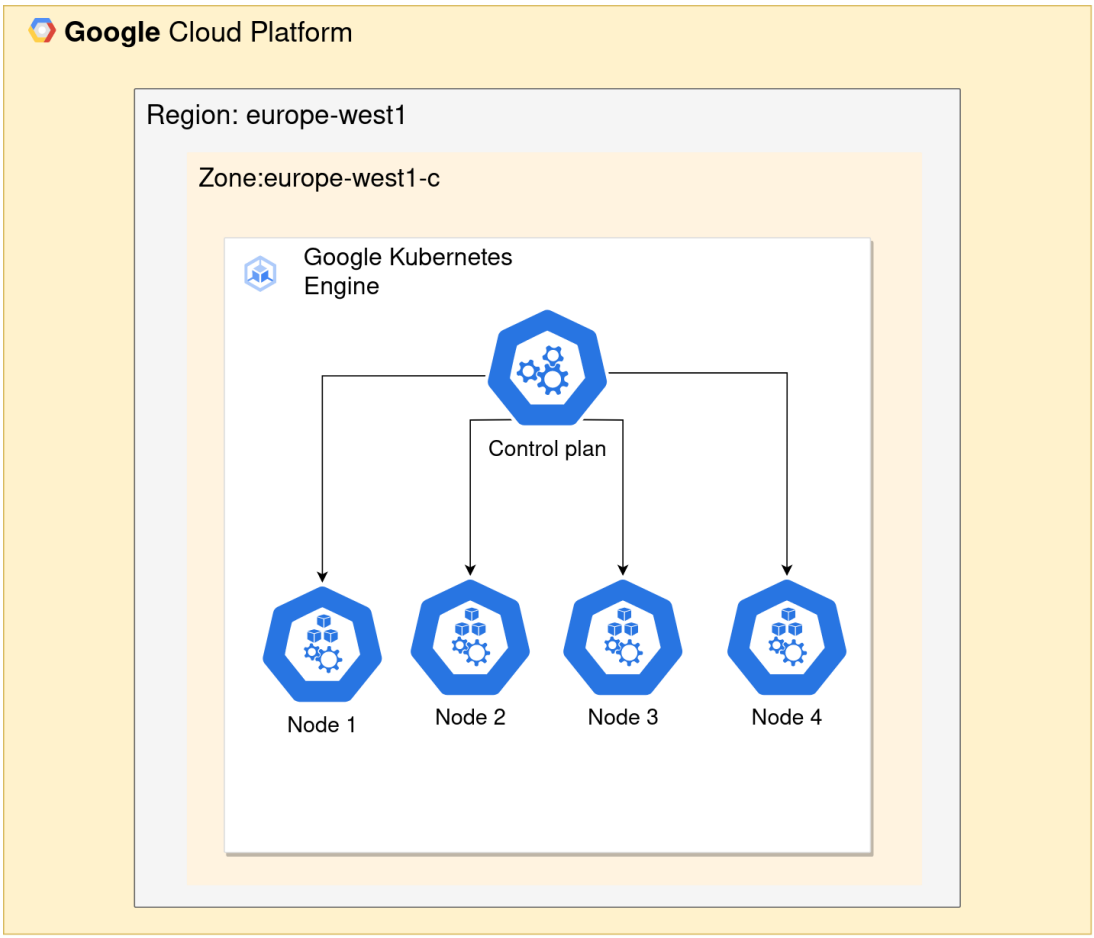

Figure 4.2: GKE cluster with four nodes

4.2 Citus Cluster Management

Once our Google Kubernetes Engine (GKE) cluster is running, we can deploy our Citus cluster by deploying the cloud native docker image in each cluster's node. Citus cluster is constructed by a coordinator node that have as purpose distributing tables and queries across the worker nodes and then get back and merges the results in order to responses the user request. The worker node that have as mission persisting data on disk and executing the distributed query from the coordinator. In this section we will present how to initialize the Citus cluster using `Kubectl` client and `YAML` language that offer deploying application and manipulating cluster resources as a code.

4.2.1 Citus Cluster Initialization

The final step that makes our distributed workload ready to use is to deploying the Citus cluster by initializing it using the deployment files that was declared beforehand in YAML. Kubernetes objects are declared as a key-value pair that makes it easy to manipulate it behavior in a declarative manner on YAML files. In the following sub-sections, we will describe the deployment objects that are needed to deploy or distributed environment.

Citus Coordinator StatefulSet

Kubernetes Deployment is defined by the docker image specification that will instantiates a `Pod` instance. In our case the image will be our pre-defined image mobilitydb-cloud.[3] The type of the Pod that need to be instantiated may takes two kind of deployment, either `StatefulSet` or `Deployment` depending on the application's needs. For our use scenario, Citus coordinator node, need to store the metadata of the cluster such as the IPs addresses of the active worker nodes, the indexes of the partitioned tables. Those data need to be persisted and highly available to avoid the inconsistency for the user's queries results. We have used the StatefulSet kubernetes object as a kind of our Citus coordinator Pod to ensure the persistence. Moreover we need a `storage` object to persist those data on the disk. To do that, we have declared the storage specification with certain amount of storage request as a persistent volume object that will automatically provision volume from Google cloud storage class.

In addition, we need to expose our Citus coordinator StatefulSet as a service to receive the external user's request. To achieve that, we declared a `service` object that play the role as an intermediate between the users and the coordinator. This latter receive the query request from the user before sending it to the coordinator and then return back the result of the query when it's processed. To make the connection possible from the outside, we assume that the firewall is open in the node that host the coordinator StatefulSet. Allowing accesses for a given port in our case, we have made the coordinator listen on the port 30001. it can be done through the following command:
`gcloud compute firewall-rules create mobilitydb-port --allow tcp:30001`
Listing 4.2 describe the YAML declaration pieces of code that show the essential Kubernetes objects declarations needed to deploy the Citus coordinator StatefulSet.

- Citus coordinator StatefulSet

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: citus-coordinator
  labels:
    app: citus-coordinator
spec:
  serviceName: citus-coordinator
  selector:
    matchLabels:
      app: citus-coordinator
    spec:
        containers:
      - name: postgresdb
        image: bouzouidja/mobilitydb-cloud:latest
        ports:
        - containerPort: 5432
        env:
        - name: PGDATA
          value: /var/lib/postgresql/data/pgdata
        volumeMounts:
            - mountPath: /var/lib/postgresql/data
```

[3] `https://hub.docker.com/repository/docker/bouzouidja/mobilitydb-cloud/general`

```
23              name: postgredb
24    volumes:
25        - name: postgredb
26          persistentVolumeClaim:
27            claimName: postgres-pv-claim-coordinator
```

Listing 4.2: Coordinator StatefulSet YAML declaration

Where in the metadata section, we specify citus-coordinator as a name of the StatefulSet and the label citus-coordinator used to organize our cluster's object. The specification section is dedicated to the Pod and the docker image, where the service name is referring to which service will be exposed. The selector is used to associate the created container and the StatefulSet by the match labels rule. The last specification it contain all the information of the docker container, where postgresdb is the name of the container instantiated with the image mobilitydb-cloud:latest, listen on the port 5432. The environment of the container is specified in `env` section where we can specify the environment variables such as the PGDATA is the volume of PostgreSQL server. This volume will be mounted in the host machine outside the container with the specified path in `mountpath` argument. Listing 4.3 list the code needed to specify the name of the mounted path within the container specification in order to provision disk space from Google storage class. The full specifications and arguments for the coordinator StatefulSet object will be available in Appendices B.2.

- Citus coordinator volume

```
1  apiVersion: v1
2  kind: PersistentVolumeClaim
3  metadata:
4    name: postgres-pv-claim-coordinator
5    labels:
6      app: citus-coordinator
7  spec:
8    accessModes:
9      - ReadWriteOnce
10   resources:
11     requests:
12       storage: 5Gi
13   storageClassName: standard-rwo
```

Listing 4.3: Coordinator volume YAML declaration

Where the `PersistentVolumeClaim` object is used to demand a volume that will be provisioning by the Citus coordinator StatefulSet. This volume itself it request the physical volume from Google storage class with size 5GB disk space. Listing 4.4 show the YAML declaration code that allow to configure the Kubernetes object service used for the coordinator in order to receive client requests.

- Citus coordinator service

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: citus-coordinator
```

```
5     labels:
6       app: citus-coordinator
7   spec:
8     selector:
9       app: citus-coordinator
10    type: NodePort
11    ports:
12      - port: 5432
13        nodePort: 30001
```

Listing 4.4: Coordinator service YAML declaration

Where `citus-coordinator` is the name of the service that have as goal, select the available Pod with the label apps that are labeled citus-coordinator and listen to outside connection on the node port 30001.

Figure 4.3 summarize the deployment of Citus coordinator within the node one, with all its necessary Kubernetes objects.

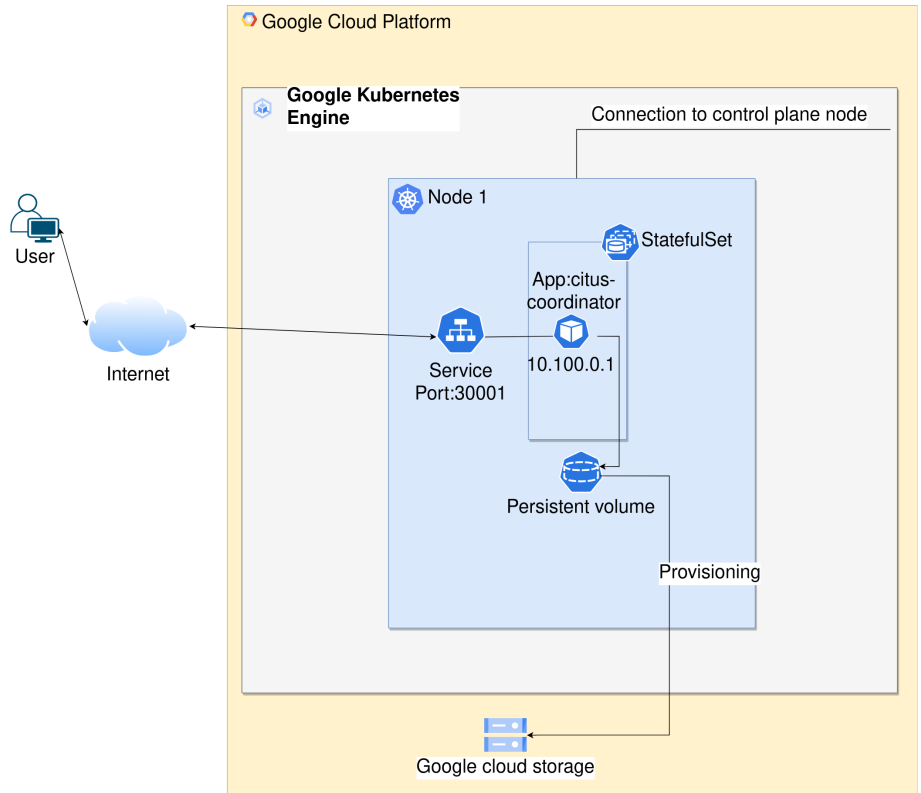

Figure 4.3: Citus coordinator StatefulSet deployment in node one

Citus Workers StatefulSet

Same as the Citus coordinator we need to persist data located in the worker nodes to ensure the high availability. However the specifications will differ in the Kubernetes objects declaration for the worker StatefulSet object. We will need more than one replicate where each replicate represent Pod per each node. For example if we have a worker StatefulSet with three replicates, this mean we will have three instance of our application hosted in three nodes. The docker image will be the same as the coordinator node which is mobilitydb-cloud:latest. The service specification will be differ on the port level, where

the port will be listen only internally inside the private network on port 5432. For the volume specification we will have a template definition that allow the creation of one volume per replicate. Listing 4.5 list the specifications of the Citus workers deployment.

- Citus workers StatefulSet

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: citus-workers
spec:
  serviceName: "citus-workers"
  selector:
    matchLabels:
  replicas: 3
      app: citus-workers
  template:
    metadata:
      labels:
        app: citus-workers
spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: "app"
                    operator: In
                    values:
                    - citus-workers
                    - citus-coordinator
              topologyKey: "kubernetes.io/hostname"
      containers:
      - name: mobilitydb-cloud
        image: bouzouidja/mobilitydb-cloud:latest
        ports:
        - containerPort: 5432
        env:
        - name: PGDATA
          value: /var/lib/postgresql/data/pgdata
        volumeMounts:
            - mountPath: /var/lib/postgresql/data
              name: postgres-pv-claim-worker

  volumeClaimTemplates:
  - metadata:
      name: postgres-pv-claim-worker
    spec:
      accessModes: ["ReadWriteOnce"]
      storageClassName: "standard-rwo"
      resources:
```

```
46          requests:
47            storage: 12Gi
```

Listing 4.5: Worker StatefulSet YAML declaration

Where the difference in the worker StatefulSet comparing the coordinator StatefulSet is the number of replicates that should be more than one as the first state of our distributed database. Concerning the selection of the node in the specification of the Pod section, we have additional parameters. We have add new rules with matching expression that assign each new created Pod replicate per a node if the matching expression is satisfy. The matching expression, app label value should be present in node labeled app= citus-coordinator and citus-workers. This rule allow to separate the worker nodes from the coordinator node in the topology of the Kubernetes cluster. topologyKey:"kubernetes.io/hostname" tells the scheduler to assign one Pod per node host name, in order to avoid having two Pods in the same node which is the default scenario in the K8s ecosystem. Listing 4.6 describe the Citus worker declarations needed to configure the service in order to communicate with the coordinator and other worker nodes.

- Citus workers service

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: citus-workers
5    labels:
6      app: citus-workers
7  spec:
8    ports:
9    - port: 5432
10     name: postgres
11   clusterIP: None
12   selector:
13     app: citus-workers
```

Listing 4.6: Worker service YAML declaration

The workers as a service listen only on internal connection, between workers and the coordinator node.

Figure 4.4 summarize the deployment of Citus workers with three nodes coordinated with one coordinator. GKE cluster should be aligned with the Citus cluster, this mean if we create a GKE of four nodes, Citus cluster will assign one node for the coordinator and three nodes as workers.

4.2.2 Citus Cluster Scale-Out

Scaling out a pre-deployed Citus cluster is an operation that resize the actual size of the cluster by adding new nodes horizontally in order to increase the workload's performance. When the database grows up, especially if the data arrives as a stream, the query's execution time increase exponentially and a resources bottlenecks will be produced. Scaling Citus cluster is conducted by scaling GKE cluster by adding more nodes within the cluster. This operation is resumed in three phases, resize the GKE cluster, resize the Citus

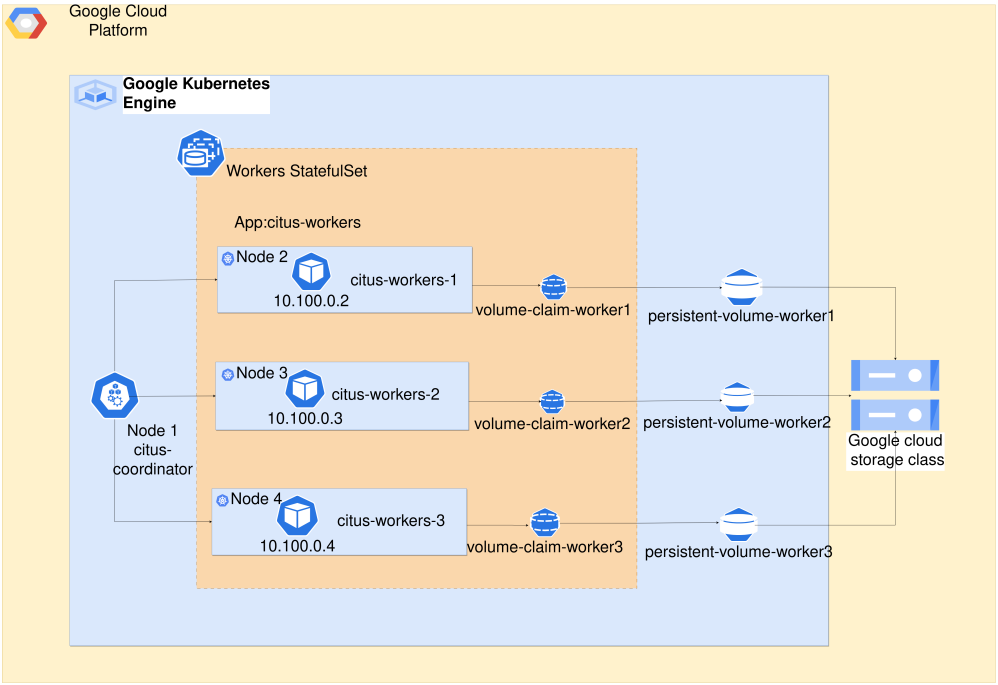

Figure 4.4: Citus workers StatefulSet

cluster and rebalance the distributed tables within the workers in order to redistribute the shards equally on all the worker nodes including the new added nodes. Figure 4.5 show an overview of the cluster after applying the scale-out operation.

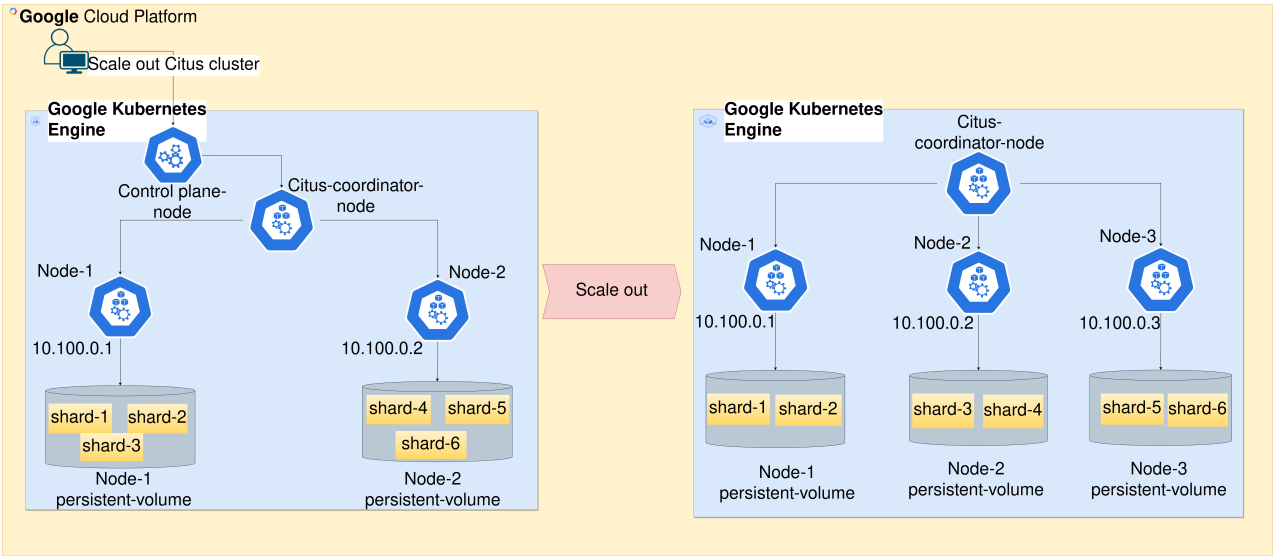

Figure 4.5: Citus cluster after Scaling out operation

4.2.3 Citus Cluster Scale-In

Scaling in Citus cluster operation is the contrast of the scale out, it allow the user to resize the cluster by removing worker nodes from the cluster. If there are distributed tables within the Citus cluster, it is mandatory to drain the desired worker nodes before delete them in order to ensure the consistency of the data. Draining the nodes it will done by first marking them as nodes that should not receive shards anymore and then moving all its shards into the other worker nodes by running Citus rebalancing command,

before removing them from the Citus cluster. Listing 4.7 describe the Citus commands that drains the nodes of the cluster.

```
SELECT * FROM citus_set_node_property('10.100.0.3', 5432,
    'shouldhaveshards', false);
---Drain the node with citus_rebalance_start:
SELECT * FROM citus_rebalance_start(drain_only := true);
---Wait until the draining rebalance finishes
---Remove the node
SELECT citus_remove_node('10.100.0.3', 5432);")
```

Listing 4.7: Drain Citus node script

Figure 4.6 show the state of the Citus cluster after applying scale-in operation on it. Scale-

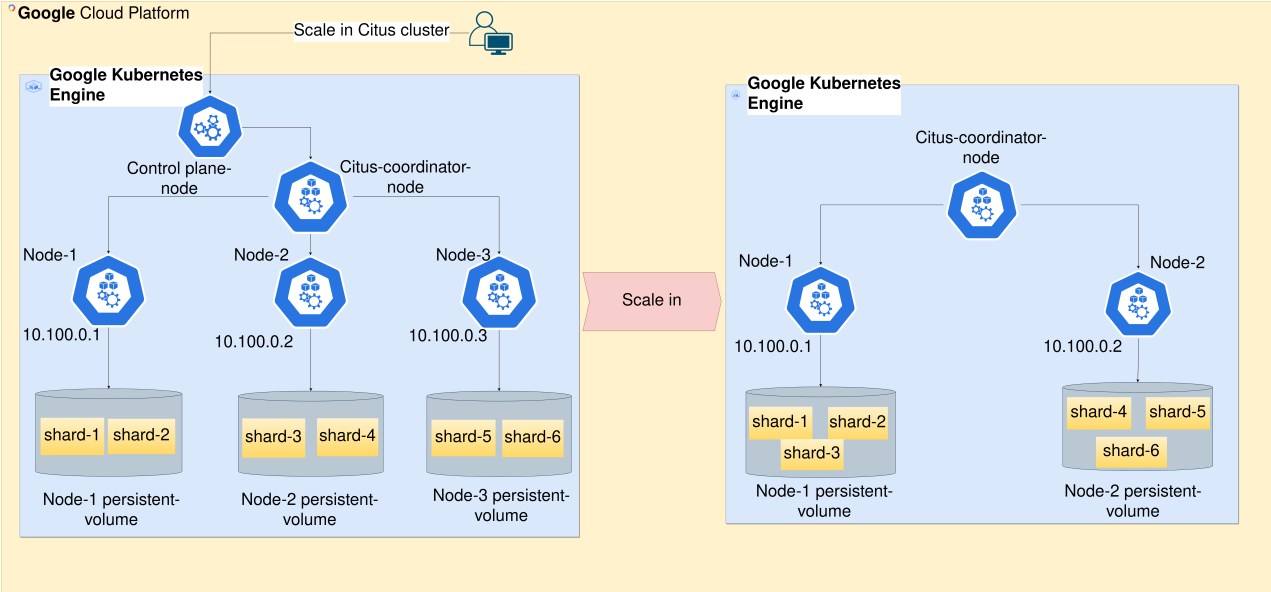

Figure 4.6: Scaling in Citus cluster operation

in and scale-out operations are developed as a stack of functions that need to be executed whether before (case of scale in) or after (case of scale out) to ensure the consistency of the workload data and allow gaining the performance correctly. This mean, in the case of scale-out if we do not re-balance the shards after recruiting new nodes, the cluster keep the same state without re-partitioning the tables for those new nodes, especially if we have a huge database. The same idea for scale-in, if we do not re-balance shards within the cluster before removing the desired nodes, we will lose data located in those nodes. We have implemented this two mandatory operations in order to preserve the database integrity. The database administrator can re-balance the Citus cluster by simply lunching the `resize` command in the command line detailed in Listing 4.8.

```
python citus_cluster_management.py resize --cluster-name
CLUSTER_NAME --cluster-zone CLUSTER_ZONE --cluster-project
CLUSTER_PROJECT --num-nodes NUM_NODES
```

Listing 4.8: Resize Citus cluster command

Where the `--cluster-name` argument is the name of the cluster desired to resize and the `--cluster-zone` is to specify which zone is located this cluster and the `--cluster-project` is the name of the project where the GKE cluster is associated and the `--num-nodes` is

used to specify the desired new size of the cluster including one coordinator node + worker nodes. In addition, initializing the Citus cluster on GKE ether is accompanied with a series of operations before being running. Among those operations, adding the worker nodes automatically to the coordinator and updating the metadata with the newest worker nodes information. Sometimes the Pods crash or lose connection with the coordinator due to stop command or when GKE cluster goes down, the Kubernetes scheduler generates new Pods, when the nodes is established and get ready to continue supporting the existing workload, the metadata of those new Pods such as the internal IP address, the name of the node is not updated in the metadata of the Citus coordinator that cause disconnection with the workers. To tackle this issue, we have implemented an SQL script that update the coordinator metadata by itself when new Pods is created.

Listing 4.9 describe the commands for further facilitation in cluster management, we have add `stop` and `start` command for the user to automating some redundant tasks.

```
python citus_cluster_management.py start --cluster-name
CLUSTER_NAME --cluster-zone CLUSTER_ZONE --cluster-project
CLUSTER_PROJECT --num-nodes NUM_NODES

python citus_cluster_management.py stop --cluster-name
CLUSTER_NAME --cluster-zone CLUSTER_ZONE --cluster-project
CLUSTER_PROJECT
```

Listing 4.9: Resize Citus cluster command

Where its takes the same arguments as resize command except for the stop command do not need the desired number of nodes. Wrapping it all together, Figure 4.7 shows the stack added in the database administrator side that allow managing the Citus cluster from it local machine.

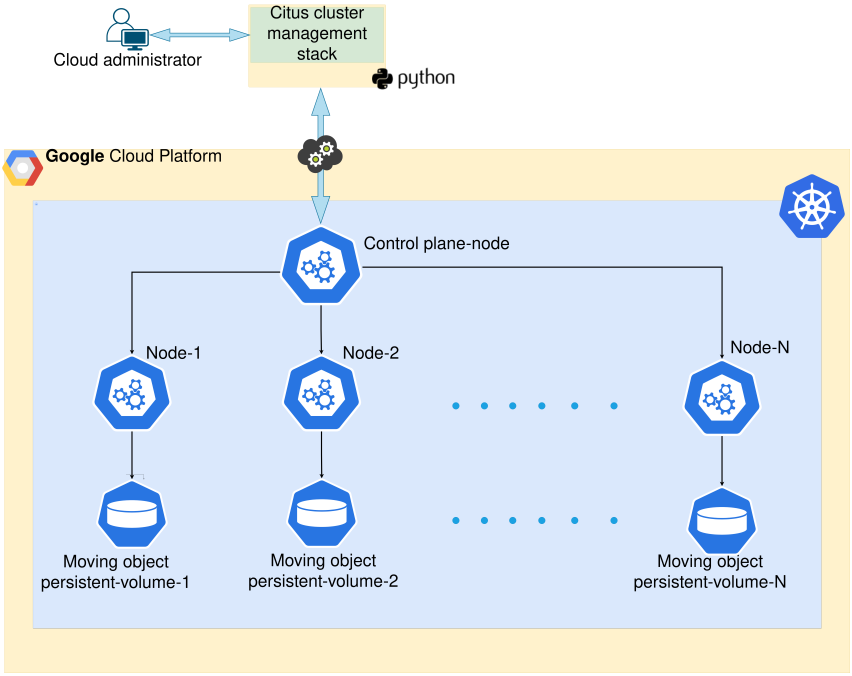

Figure 4.7: Citus cluster management with operation stack

In this chapter, we have examined the various deployment phases of our distributed environment within the GCP platform. We began with the initialization of the GKE cluster,

enabling the allocation of infrastructure resources. Subsequently, the Citus cluster was initialized to partition and distribute the shards across all worker nodes. The roles of the scale-in and scale-out methods, along with their implementations, have been detailed to ensure the integrity of the database. In the following chapter, we will conduct experiments on our implemented distributed system to evaluate performance across different cluster sizes. These experiments will be conducted using both theoretical and real-world data.

Chapter 5

Experimental Evaluation

In this chapter we will perform our experiments for our solution on a Google Kubernetes Engine (GKE) cluster for a distribute geo-spatial database with PostgreSQL server doted by MobilityDB and Citus extension.

In order to give an insight on the performances gained when using our distributed workload, we will evaluate MobilityDB queries on a different cluster configurations in term of the size of the cluster, the number of resources (number of CPU core and memory size) used, then we capture those evaluation results using Python script and GCP monitoring service metrics. Each MobilityDB query have it own logic and behavior according to it target objectives. For that reason we will classify those queries to classes in order to see if we distribute efficiently all kind of query complexity. In the following sections, we will explain the experiment environment in which we will define four different configurations to variate the resource capacity of our cluster. Afterwards, we initialize first our GKE cluster and then we will discuss the benchmark tool BerlinMOD and it data generator used to perform our experiments, this tools was used in several geo-spatial databases benchmarks. And then we will talk about the partitioning tables phase using Citus extension which is a crucial step that helps Citus operators distributes correctly the tables within the cluster and the sharing of the query across the cluster. And then, we present our experiment's results through some graphs in order to analyse them and take some conclusions. Finally, we will experiment with our distributed environment using real-world data sourced from the Danish Maritime Authority. This data comprises AIS information and is intended for storing and analyzing the tracking data of their fleet of ships.

5.1 Experiment Environments

In order to evaluate the performance of our cloud-native and distributed database of moving objects on GCP, we have prepared different execution environments to meet different test scenarios. Our experiments will be tested in four clusters where each cluster has its own characteristics in terms of number of CPU cores and memory capacity, disk type and number of nodes connected in the cluster. In the rest of this part, we will generate synthetic data with different database sizes in order to query them via predefined SQL queries. The executions will take place on these four environments. In following we will discuss the characteristics of each cluster environment. We have made four cluster environment in two different regions. All the virtual machine is as type general purpose.

Table 5.1, we summarize the different configuration environments chosen to perform our

experiments. All the existing machine type offered by GCP it can be found in [27] with

	VM type	CPU	RAM	Disk	Cluster size	Region/Zone
cluster-config1	Intel(R) Core(TM)	5 cores	6 GB	SSD	Single	Local machine
cluster-config2	e2-standard-4	4 cores	8 GB	Balanced	4 nodes	europe-west1-c
cluster-config3	e2-standard-4	4 cores	8 GB	Balanced	8 nodes	europe-west1-d
cluster-config4	e2-standard-4	4 cores	8 GB	Balanced	16 nodes	europe-west1-c

Table 5.1: Cluster environment description

the full description of each machine. Even for the different regions and zone available in GCP are detailed in [29]

5.2 GKE Cluster Initialization

After choosing the four cluster configuration environments, we proceed to the creating and initializing the Google Kubernetes Engine cluster by allocating the appropriate resources from the GCP platform. In the code provided in Listing 5.1, an example of the commands needed to create cluster of eight node from the terminal using the Google cloud client `gcloud`. To simplify the lecture, we will create a cluster with few parameters using the configuration `cluster_config2` cited in Table 5.1.

```
1 gcloud container clusters resize mobilitydb-cluster-2 --zone
2 europe-west1-c --node-pool mobilitydb-pool-config2
3 --machine-type e2-standard-4 --disk-type pd-balanced
4 --num-nodes 8
```

Listing 5.1: Cluster creation with gcloud commands

Where the name of the cluster is mobilitydb-cluster-2 created in europe-west1-c zone with eight nodes as number of virtual machines. Node pool parameter means the template of the nodes, each node have the same configuration specified in the node pool. After few minutes, normally we will see the cluster created with eight nodes as showing in Figure 5.1. We have used the command `kubectl` to show the information of the nodes within the

```
bouzouidja@linux:~/Desktop/Thesis_Work/MobilityDB-GCP$ kubectl get node -o wide
NAME                                                  STATUS   ROLES    AGE     VERSION           INTERNAL-IP    EXTERNAL-IP      OS-IMAGE
       KERNEL-VERSION    CONTAINER-RUNTIME
gke-mobilitydb-clust-mobilitydb-pool--6eb648cb-07fw   Ready    <none>   5h21m   v1.25.8-gke.500   10.132.0.10    34.78.42.213     Ubuntu 22.04.
2 LTS   5.15.0-1028-gke   containerd://1.6.18
gke-mobilitydb-clust-mobilitydb-pool--6eb648cb-2dtp   Ready    <none>   16h     v1.25.8-gke.500   10.132.0.9     35.241.141.97    Ubuntu 22.04.
2 LTS   5.15.0-1028-gke   containerd://1.6.18
gke-mobilitydb-clust-mobilitydb-pool--6eb648cb-5kkg   Ready    <none>   156m    v1.25.8-gke.500   10.132.0.15    104.155.115.209  Ubuntu 22.04.
2 LTS   5.15.0-1028-gke   containerd://1.6.18
gke-mobilitydb-clust-mobilitydb-pool--6eb648cb-5krw   Ready    <none>   156m    v1.25.8-gke.500   10.132.0.20    34.22.200.100    Ubuntu 22.04.
2 LTS   5.15.0-1028-gke   containerd://1.6.18
gke-mobilitydb-clust-mobilitydb-pool--6eb648cb-9v75   Ready    <none>   156m    v1.25.8-gke.500   10.132.0.28    35.195.185.14    Ubuntu 22.04.
2 LTS   5.15.0-1028-gke   containerd://1.6.18
gke-mobilitydb-clust-mobilitydb-pool--6eb648cb-c33q   Ready    <none>   156m    v1.25.8-gke.500   10.132.0.22    34.22.181.138    Ubuntu 22.04.
2 LTS   5.15.0-1028-gke   containerd://1.6.18
gke-mobilitydb-clust-mobilitydb-pool--6eb648cb-gnc6   Ready    <none>   156m    v1.25.8-gke.500   10.132.0.23    35.187.54.1      Ubuntu 22.04.
2 LTS   5.15.0-1028-gke   containerd://1.6.18
gke-mobilitydb-clust-mobilitydb-pool--6eb648cb-hmlz   Ready    <none>   5h20m   v1.25.8-gke.500   10.132.0.12    35.195.100.192   Ubuntu 22.04.
```

Figure 5.1: GKE cluster of eight nodes

Kubernetes cluster.

5.3 BerlinMOD Benchmarks

BerlinMOD is such a benchmark tool that aim to demonstrate and evaluate moving object databases through a generated data. As explained in [13] the BerlinMOD benchmark is based on several simulation scenarios where the positions of vehicle is tracked during a period of time within Berlin city. BerlinMOD use the SECONDO [19] tool which is a data generator that simulate trajectories of moving object within a given city. For our experiments, we have generated data for Brussels city using a customized BerlinMOD data generator to benchmark MobilityDB called MobilityDB-BerlinMOD where the full manual it can be found in [46]. Such a benchmark tool,it provides to us a well defined data model and queries dedicated to analyse moving object data, and also simplify to us the experimentation by allowing us repeating the simulation many time and it point out to us the weaknesses of our distributed environment. The schema of the generated data model from the BerlinMOD generator is represented in Listing 5.2.

```
Vehicles(VehId integer,Licence varchar,Type varchar,
    Model varchar).
Instants(InstantId integer, Instant timestamptz).
Periods(PeriodId integer, beginp timestamptz, endp timestamptz).
Points(PointId integer,posx double precision,
    posy double precision, Geom Geometry).
Regions(RegionId integer, Geom Geometry).
Licences(LicenceId integer, VehId integer,Licence varchar.
Trips(TripId integer,vehId integer, day date, seqno integer,
    sourcenode integer, targetnode integer, trip tgeopoint,
    trajectory geometry).
```

Listing 5.2: BerlinMOD schema

Where the trips table represents the trajectories of vehicles with timestamps for each record. A trip start from a given point to a destination point such as from home node to work node. The licences table represent information about vehicle licence number. Point and region tables represent geometry data type such as point and polygon that records map information in binary format. And the periods table represent the duration of the trip. Concerning the vehicle table, it records the vehicles information including the licence of the vehicle, the type and the model of the vehicle. This scheme will be the basis of our experiments.

Data Generation

As mentioned before, BerlinMOD allows us to generate a simulated moving object data that include trajectories and trips of vehicles from the work location to the home location in the week days or other destination for the whole week. BerlinMOD provides a set of `SQL-like` procedures and functions supported by PostgreSQL server. It's a customizable tools that we can add our own procedures, tables and new schema on top of it such as in the case of MobilityDB-BerlinMOD [46]. Using MobilityDB-BerlinMOD data generator, we can generate data of any scale that we want by specifying the scale factor parameter as a PostgreSQL query such as SELECT $berlinmod\_generate(scaleFactor := 0.5)$ If we don't specify any scale, by default, the scale factor is 0.005. The full manual on how to use and generates data with scale factor parameter, it can be found in [6]. Table 5.2 show us the information about the different scale generated to pursuit our experiments. At the end, we have generated four databases where each database represent one scale of

BerlinMOD generated data.

Scale factor	Vehicles	number of days	Trips	Size	Duration
0.05	447	8	9025	846 MB	8 minutes
0.2	894	13	62510	2296 MB	40 minutes
0.5	1414	22	77044	4858 MB	2 hours
1	2000	28	292940	11GB	5 hours

Table 5.2: Databases generation description

Figure 5.2 show the generated trips within Brussels city for the scale factor 0.005 using the QGIS[1] tool.

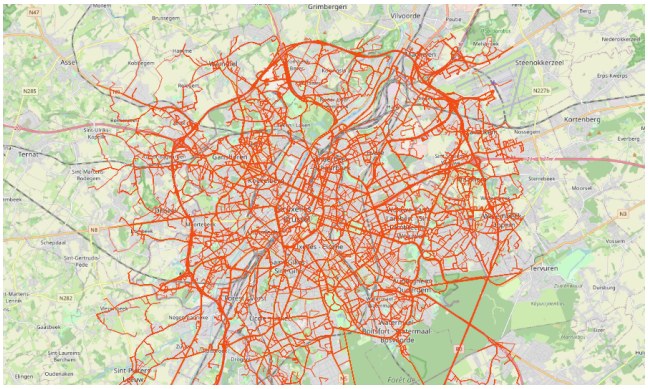

Figure 5.2: The generated trips within Brussels city with scale factor 0.005

5.4 Citus Cluster Initialization

After deciding the amount of data generation and which environments will be support our experiments. We start deploying our native cloud docker image composed by MobilityDB and Citus extensions for PostgreSQL server on the Google Kubernetes Engine clusters. Before running the BerlinMOD data generator for each scale factor and querying those data through the dedicated BerlinMOD queries, we need to initialize the Citus cluster which is a required phase to partition the tables and distribute the queries across the cluster. After creating the StatefulSet for the coordinator and the worker nodes, we will get two different type of Pods with a certain number of replicas where each replica is considered as one Citus node. For example a coordinator StatefulSet with one replica produce us one coordinator Pod and the worker StatefulSet with three replicas produce us three worker Pods. Figure 5.3 show us Kubernetes cluster with PostgreSQL server with eight nodes, one node as coordinator and seven nodes as workers.

Once the PostgreSQL cluster is up and all the Pods is running, we proceed to the connection of the Citus cluster by adding worker nodes to the coordinator. The commands listed in Listing 5.3 need to be run from the coordinator node side in order to attach their worker nodes and fulfill the metadata tables.

```
1
2 SELECT citus_set_coordinator_host('10.100.0.12', 5432);
```

[1]QGIS used to handle Geographic Information System. `https://qgis.org/en/site/`

```
bouzouidja@linux:~/Desktop/Thesis_Work/MobilityDB-GCP$ kubectl get all -o wide
NAME                    READY   STATUS    RESTARTS   AGE   IP            NODE                                                NOMINATED NODE
   READINESS GATES
pod/citus-coordinator-0   1/1     Running   0          30h   10.100.0.12   gke-mobilitydb-clust-mobilitydb-pool--8fb8882a-hgpc   <none>
   <none>
pod/citus-workers-0       1/1     Running   0          11h   10.100.2.4    gke-mobilitydb-clust-mobilitydb-pool--8fb8882a-5pvr   <none>
   <none>
pod/citus-workers-1       1/1     Running   0          11h   10.100.3.3    gke-mobilitydb-clust-mobilitydb-pool--8fb8882a-svjb   <none>
   <none>
pod/citus-workers-2       1/1     Running   0          11h   10.100.1.4    gke-mobilitydb-clust-mobilitydb-pool--8fb8882a-q38j   <none>
   <none>
pod/citus-workers-3       1/1     Running   0          68m   10.100.4.2    gke-mobilitydb-clust-mobilitydb-pool--8fb8882a-85px   <none>
   <none>
pod/citus-workers-4       1/1     Running   0          67m   10.100.5.2    gke-mobilitydb-clust-mobilitydb-pool--8fb8882a-j26c   <none>
   <none>
pod/citus-workers-5       1/1     Running   0          66m   10.100.7.4    gke-mobilitydb-clust-mobilitydb-pool--8fb8882a-d5xn   <none>
   <none>
pod/citus-workers-6       1/1     Running   0          65m   10.100.6.3    gke-mobilitydb-clust-mobilitydb-pool--8fb8882a-1w5m   <none>
   <none>
```

Figure 5.3: Citus cluster for PostgreSQL with eight nodes

```
SELECT * from citus_add_node('10.100.2.4', 5432);
SELECT * from citus_add_node('10.100.3.3', 5432);
SELECT * from citus_add_node('10.100.1.4', 5432);
SELECT * from citus_add_node('10.100.4.2', 5432);
SELECT * from citus_add_node('10.100.5.2', 5432);
SELECT * from citus_add_node('10.100.7.4', 5432);
SELECT * from citus_add_node('10.100.6.3', 5432);
```

Listing 5.3: 8 nodes Citus cluster initialization commands

Depending on the IP addresses generated by the Kubernetes scheduler that are given to the Pods, the first command means the coordinator assign itself the IP of the coordinator Pod and then add the other IP addresses as workers. Figure 5.4 show the connected worker nodes from the coordinator server.

```
brussels-s1=# SELECT * from citus_get_active_worker_nodes();
 node_name  | node_port
------------+-----------
 10.100.7.4 |      5432
 10.100.5.2 |      5432
 10.100.3.3 |      5432
 10.100.1.4 |      5432
 10.100.4.2 |      5432
 10.100.2.4 |      5432
 10.100.6.3 |      5432
(7 rows)
```

Figure 5.4: Citus cluster with one coordinator and seven worker nodes

5.4.1 Table Partitioning Using Citus Extension

Partitioning tables is a crucial phase in a distributed database system if we want to have a high performance when querying the data. `A good partitioning tables is ensured by a good choice of a distributed column`. From the coordinator node, Citus data operator distributes table using the selected distributed column by hashing it in order to creates a global index on to of the cluster. Based on the active worker nodes, Citus operator creates shards from the partitioning table phase that contain a set of rows from each distributed table. Those shards serve as the final location of records within the global index and each row within a shard is unique based on the hash value assigned to it. As best practice, choosing the column that have the highest cardinality than other in order to have an efficient cluster index for the distributed table. High cardinality means high number of shards within the cluster's nodes. This will gives us a strong cluster index with a well spreads hash values across the cluster.

If we want to distribute the `Trips` table as described in Listing 5.2 for example, we need to choose as distributed column the `TripId` because it have high cardinality than other attributes. The remaining tables will be distributed as shown in Listing 5.4 from the coordinator node. Where the smallest tables (dimensions) will be referenced in each worker node to accelerate the query operations such us join or aggregation with `Trips` table.

```
SELECT create_distributed_table('trips', 'tripId');
SELECT create_reference_table('vehicles');
SELECT create_reference_table('instants');
SELECT create_reference_table('licences');
SELECT create_reference_table('periods');
SELECT create_reference_table('points');
SELECT create_reference_table('regions');
```

Listing 5.4: Distributing Trips table and referencing dimension tables on the cluster

If for a given query need to join the tables `Trips` and `Vehicles` the Cartesian product will be easy because each node operates separately since it have partitions from the table `Trips` and as lookup table `Vehicles` locally.

5.4.2 Query Distribution Using Citus Extension

Moving object queries often encounter CPU and memory bottlenecks, especially in geometry functions like `ST Intersects`, `ST Distance`, and `Trajectory` when applied to large tables. To address this issue, it becomes essential to distribute Moving object queries evenly across the nodes of the cluster. In the realm of relational database systems, operations such as joins, aggregation functions, and CTE queries tend to consume a significant CPU and memory compared to selection or projection operations.

To optimize performance, the approach involves distributing the extensive tables and dispatching all reference tables to worker nodes. The Citus planner then takes the query plan and transforms it into an associative and commutative form that can be executed in parallel by the workers. Each worker handles a fragment from the global query plan and executes it on its local shards. Once the fragment query plan results are obtained from the workers, the coordinator consolidates these results. In Listing 5.6 we illustrate the distribution of tasks across all workers while executing the "EXPLAIN" and "ANALYZE" commands in PostgreSQL for BerlinMOD Query 13, as outlined in Listing 5.5.

```
SELECT DISTINCT R.RegionId, P.PeriodId, P.Period, C.Licence
FROM Trips T, Vehicles C, Regions1 R, Periods1 P
WHERE T.VehId = C.VehId AND T.trip && stbox(R.Geom, P.Period)
AND ST_Intersects(trajectory(atTime(T.Trip, P.Period)), R.Geom)
ORDER BY R.RegionId, P.PeriodId, C.Licence;
```

Listing 5.5: Description of BerlinMOD Query 13

Assuming `Trips` table are distributed and `Vehicles`, `Regions` and `Periods` tables are referenced in the three worker nodes.

```
Sort  (cost=1009.64..1010.14 rows=200 width=64) (actual time=12100.005..
      12100.056 rows=631 loops=1)
   Sort Key: remote_scan.regionid, remote_scan.periodid, remote_scan.licence
   Sort Method: quicksort Memory: 75kB
```

```
5    ->  HashAggregate (cost=1000.00..1002.00 rows=200 width=64)
6                       (actual time=12099.322..12099.423 rows=631 loops=1)
7          Group Key: remote_scan.regionid, remote_scan.periodid,
8                     remote_scan.licence, remote_scan.period
9          Batches: 1 Memory Usage: 121kB
10         ->  Custom Scan (Citus Adaptive) (cost=0.00..0.00 rows=100000
11                         width=64) (actual time=12098.712..12098.819
12                         rows=1192 loops=1)
13               ->  Distributed Subplan 8_1
14                     Subplan Duration: 27.39 ms
15                     Intermediate Data Size: 10 kB
16                     Result destination: Send to 3 nodes
17                     ->  Custom Scan (Citus Adaptive) (cost=0.00..0.00
18                                    rows=0 width=0)
19                                    (actual time=1.550..1.552 rows=10 loops=1)
20                           Task Count: 1
21                           Tuple data received from nodes: 10 kB
22                           Tasks Shown: All
23                           ->  Task
24                                 Tuple data received from node: 10 kB
25                                 Node: host=10.48.0.6 port=5432
26                                  dbname=mobilitydb
27                                 ->  Limit  (cost=0.00..1.30 rows=10
28                                 width=897) (actual time=0.009..0.018
29                                 rows=10 loops=1)
30                                       ->  Seq Scan on regions_102013
31                                       regions (cost=0.00..13.00 rows=100
32                                       width=897) (actual time=0.007..0.015
33                                       rows=10 loops=1)
34                                    Planning Time: 0.056 ms
35                                    Execution Time: 0.070 ms
36                     Planning Time: 0.000 ms
37                     Execution Time: 1.591 ms
38               ->  Distributed Subplan 8_2
39                     Subplan Duration: 11.59 ms
40                     Intermediate Data Size: 580 bytes
41                     Result destination: Send to 3 nodes
42                     ->  Custom Scan (Citus Adaptive) (cost=0.00..0.00
43                     rows=0 width=0) (actual time=0.624..0.626 rows=10 loops=1)
44                           Task Count: 1
```

Listing 5.6: Explain and Analyze of Query 13

In Listing 5.6, the query execution plan's distribution is facilitated by the Citus operator using a customized scan known as "Citus adaptive.". In line 3 we notice that remote scan is needed on all workers, the remote scan concern the table regions, periods and licences that are already dispatched by the coordinator. As indicated in line 13 and 38, there are two sub-plan destined to the worker nodes to perform parallel tasks. Each Citus operator within the worker nodes conducts a sequential scan exclusively on the existing shards of the *Trips* table and the associated reference tables.

Additionally, in line 30 reveals one sequential scan on shard regions_102013 to execute

the join operation efficiently using the *Regions* reference table. At the line 41, an intermediate result need to be sent to the workers (three nodes). To conclude this section, a good distributed query is ensured by a good table partitioning phase with the adequate distributed column that may be selected if it has a high cardinality in term of unique row.

5.5 BerlinMOD Query Results and Analysis

To benchmarks our native cloud application for the distributed MobilityDB workload, we had run the customized MobilityDB queries with BerlinMOD benchmark as detailed in [46]. The goal is to inspect the execution time of the queries and analyzing the performance gained when we change the database scale. To achieve our experiments, we have prepared a benchmark script that run some of MobilityDB for BerlinMOD queries and highlighting the execution time of those queries grouped databases scale using an interactive visualization tool Dash Plotly for Python.[2] In Table 5.3 we have summarized the description of the queries that are selected to perform our experiments. Each BerlinMOD

Query id	**Description**
Query 4	Which vehicles have passed the point A from point B.
Query 7	What are the licence plate numbers of the passenger cars that have reached the points from Points first of all passenger cars during the complete observation period.
Query 9	What is the longest distance that was travelled by a vehicle during each of the periods from Periods.
Query 13	Which vehicles travelled within one of the regions from Regions1 during the periods from Periods1.

Table 5.3: Description for MobilityDB queries used in the experiments

query have it own behavior and complexity level in term of the number of sequential scan needed to perform the join operation and the amount of the intermediate data needed to be stored in memory to compute the result of the sub-queries. To have an accurate approximation of the query's execution time, we have run the queries five iterations and perform an average execution time between them, this allow us comparing between the different cluster size for different database scale. Because the data will not yet be loaded in the cash memory, we pass the first iteration within the loop. The complete benchmark script will be shown in Appendices part C. Figure 5.5 illustration is an overview of the execution time for Query 4 in different database sizes that was already generated with different scale factors. The execution is tested on four different cluster size for different database scale.

As we can see, the execution time is drastically improved from single node cluster size to 16 nodes for all the scale factors. If we take the scale factor 0.5, we notice that the Query 4 takes 800 seconds (approximated to 13.30 minutes) in single node and takes 72 seconds (approximated to 1.20 minute) in cluster size of 16 nodes, this is due to the distributed work assigned to the workers. Table 5.4 summarizes all the execution time by scale factors and cluster sizes. As explained in Figure 5.5, the join operation within the Citus cluster is parallelized on the worker nodes. To viewing those line charts in another angle behind the Google Kubernetes cluster resources used, we have take a look on the CPU and memory utilization used during the run-time of the Query 4 in the cluster of four nodes, one for

[2] `https://dash.plotly.com/dash-core-components/graph`

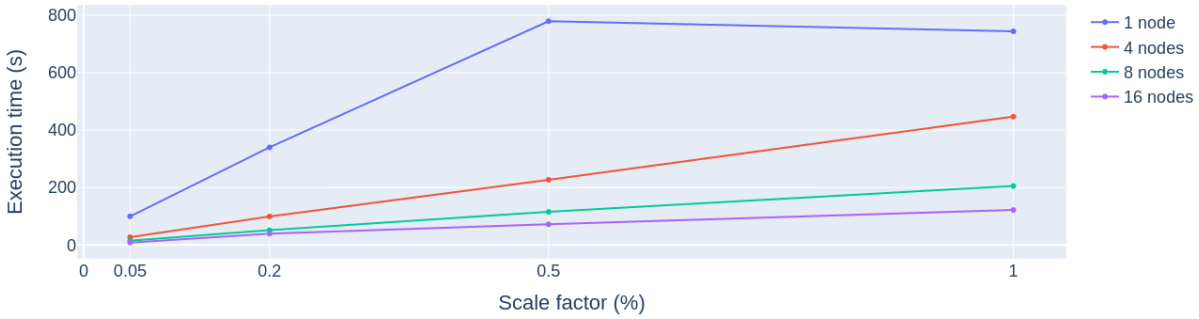

Figure 5.5: Query 4 execution time grouped by cluster size and scale factor

	Scale 0.05	Scale 0.2	Scale 0.5	Scale 1
1 node	99.84	339.87	778.88	743.5
4 nodes	27.19	99.27	226.98	446.65
8 nodes	15.29	51.47	115.45	205.25
16 nodes	8.36	39.44	72.01	121.8

Table 5.4: Averages execution time of Query 4 per cluster size and scale factors

Citus coordinator and three as worker nodes using Google Cloud monitoring tool. We will focusing only on the resources consumed in the workers. Figures 5.6 and 5.7 illustrated from the Google monitoring tool show the CPU and memory usage when executing Query 4, where we can see two peaks, one for the memory utilization and one for the CPU utilization graph due to consecutive execution of BerlinMOD Query 4. Google Cloud

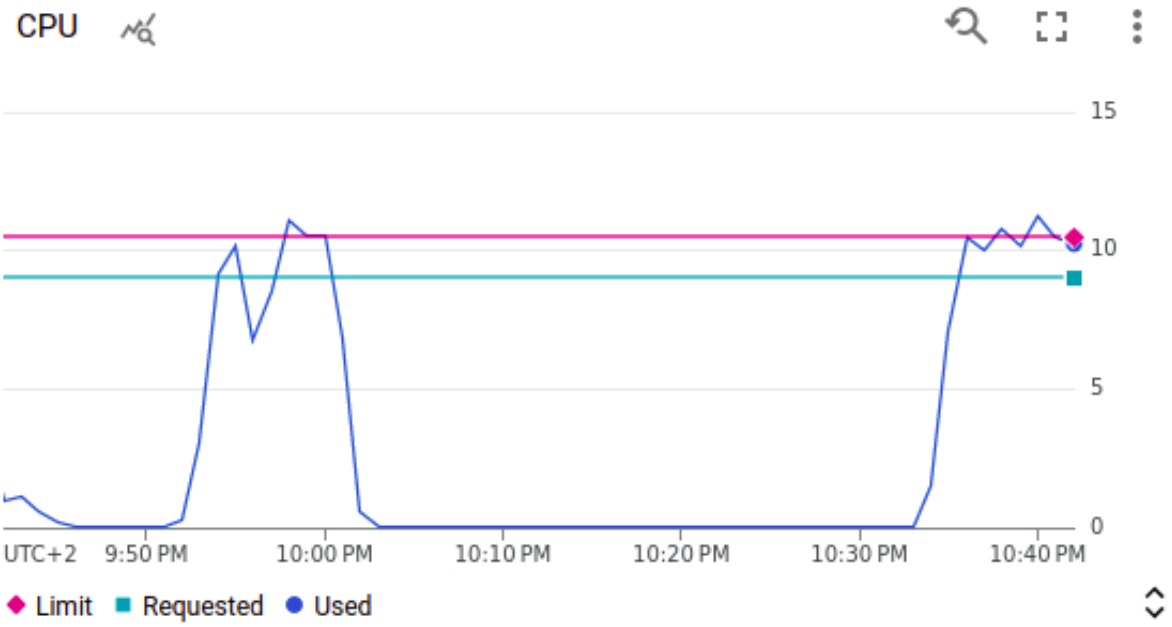

Figure 5.6: CPU utilization with three worker nodes when executing Query 4

monitoring work permanently to save metrics utilization from Kubernetes workloads and summarize them in monitoring dashboards. Google Monitoring tool gives the possibility to navigate in the old monitoring data by changing the range of time in the X axis that

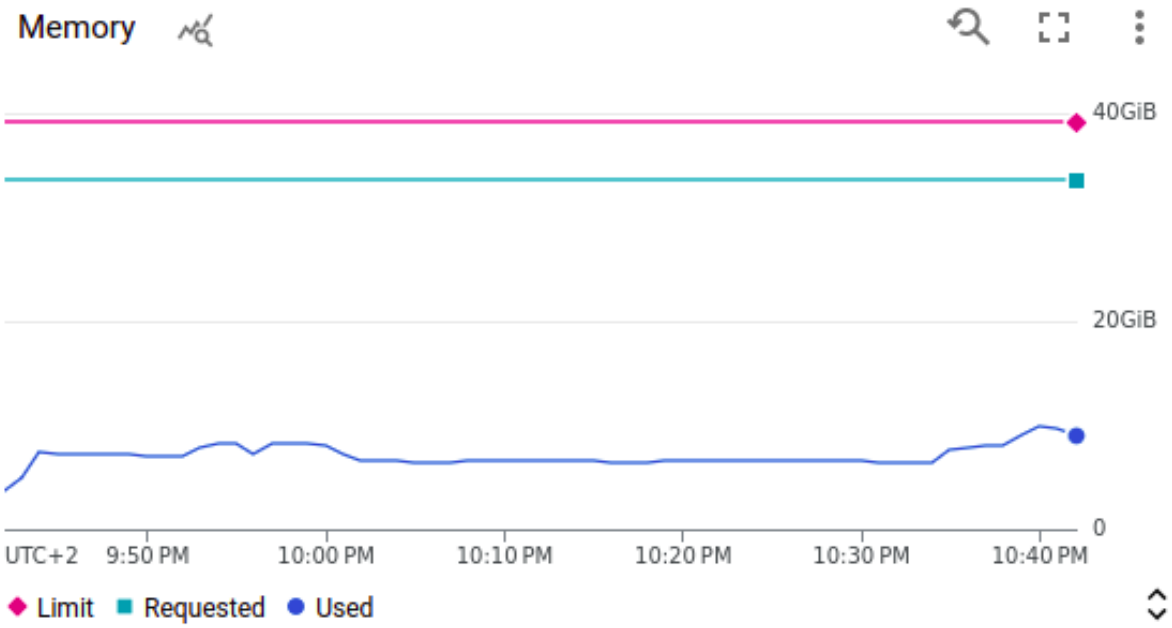

Figure 5.7: Memory utilization with three worker nodes when executing Query 4

help us to visualize the past query's execution in order to compare it with different cluster sizes and configurations. The red line in both graphs is the threshold limit that we have specified during the deployment phase in the configuration of the Pods in order to limit the utilization for the node's resources, where the limit here is the sum of resources for three workers. The green line is the threshold request that allow Pods request this amount of resources from the node. The blue line is dedicated to the resource utilization of the workers when they received the fragment of the global execution plan from the coordinator node. As we can see in the CPU utilization graph, all the workers request their CPU until the limit, this due to the complex operations within the Query 4 such as `stbox(P.Geom)`, `trajectory(T.Trip)` and `ST_Intersects(trajectory(T.Trip))`. The threshold limit value of CPU utilization is equal to 12 CPU cores which is the sum of 3 * 4 worker CPU cores without counting the coordinator CPU cores. The memory threshold value in the left graph is equal to 38 GB which is the sum of 3 * 14 GB worker memory. The total memory used during the Query 4 is about 3 * 2.5 GB per each worker, this used amount is not much comparable because the Query 4 don't have sub-queries that usually store temporary result in memory.

Figure 5.8 is dedicated to the average execution time for the Query 9 in different cluster sizes with different scale factors. Comparing from Figure 5.8 on scale factor one, we can see that single node takes a huge amount of time which is approximately 1060 seconds (about 17.66 minutes), where in 16 nodes cluster the time is improved to 172 seconds (about 2.86 minutes). Again this improvement is due to the parallel join operations computed by the workers. For further understanding one could inspect the Query 9 SQL description in Listing 5.7 and see what is happened in the workers side.

```
1 WITH Distances AS (SELECT P.PeriodId, P.Period, T.VehId,
2 SUM(length(atTime(T.Trip, P.Period))) AS Dist FROM Trips T,
3  Periods P WHERE T.Trip && P.Period GROUP BY P.PeriodId,
4  P.Period, T.VehId)
5 SELECT PeriodId, Period, MAX(Dist) AS MaxDist
6 FROM Distances
```

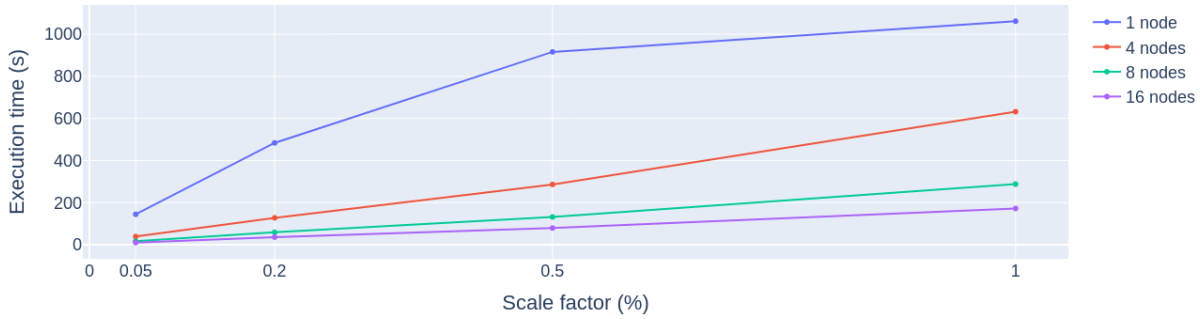

Figure 5.8: Query 9 execution time grouped by cluster size and database scale

```
7 GROUP BY PeriodId, Period
8 ORDER BY PeriodId;
```

Listing 5.7: BerlinMOD Query 9 descreption

From section Citus cluster initialization, in Listing 5.4, we have partitioned the tables *Trips* over the workers and make as reference table the tables, *Instances* and *Periods*. Once the Query 9 is received on the coordinator node, Citus adaptive operator prepare a distributed query plan where each worker will be asked to complete a fragment of it. From the SQL description of Query 9 in Listing 5.7 we see that a join operation is needed between *Trips* and *Periods* table in the `WITH` clause. Each worker have certain amount of shards of the distributed table `Trips`, so it can perform the join between the shards of trips and the reference table period. Table 5.5 summarize the detail of all the execution time of Query 9 by different cluster size and different database scale.

	Scale 0.05	Scale 0.2	Scale 0.5	Scale 1
1 node	144.44	483.62	915.93	1061.65
4 nodes	38.84	127.85	286.31	631.86
8 nodes	16.31	59.01	131.76	287.9
16 nodes	10.15	35.92	79.31	172.35

Table 5.5: Details of execution time for Query 9

Figure 5.9 show a global overview of all queries grouped by the Citus cluster size with the scale factor equal to 0.05. Where all the queries are improved from single node cluster size to 16 nodes cluster size. This result prove that our distributed workload gain the performance by scaling out the cluster horizontally. Table 5.6 show the numerical average execution time for all queries per cluster size with scale factor=0.05.

	Q4	Q7	Q9	Q13
1 node	99.84	15.89	144.44	115.88
4 nodes	27.19	2.79	38.84	20.75
8 nodes	15.29	1.48	16.31	11.37
16 nodes	8.36	0.91	10.15	7.18

Table 5.6: Details of execution time grouped by query and cluster size for scale factor=0.05

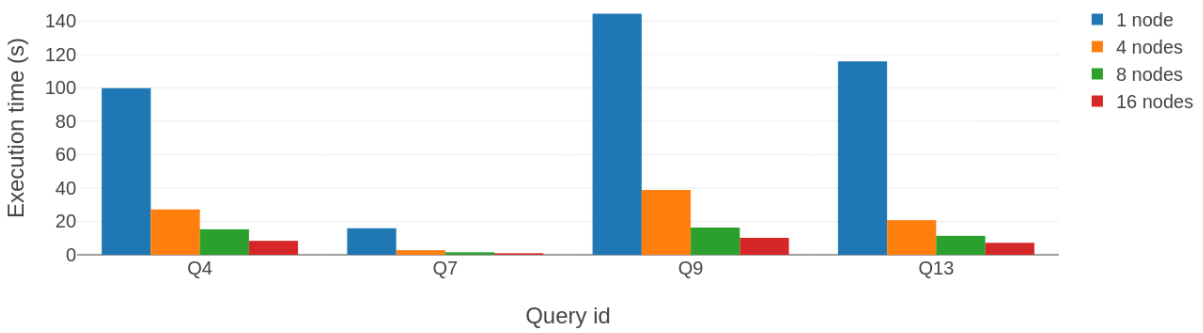

Figure 5.9: Performance insight of BerlinMOD queries by cluster size with scale factor=0.05

In order to overview the performance when we change the scale of the generated database, the scale factor=1 generates more then 292000 trips and allows to track more than 2000 vehicles which is a big dataset that help us analyzing the gained performance proportionally with the size of the tables. In our case, for those experiments, the *Trips* table is the most significant in terms of size and the primary one in terms of user objectives. All the join operations on trips table decreases the performance when the number of records grows during the time. Figure 5.10 summarize the average execution time per cluster size with scale factor 1. The results from Figure 5.10 shows that all the queries are improved with

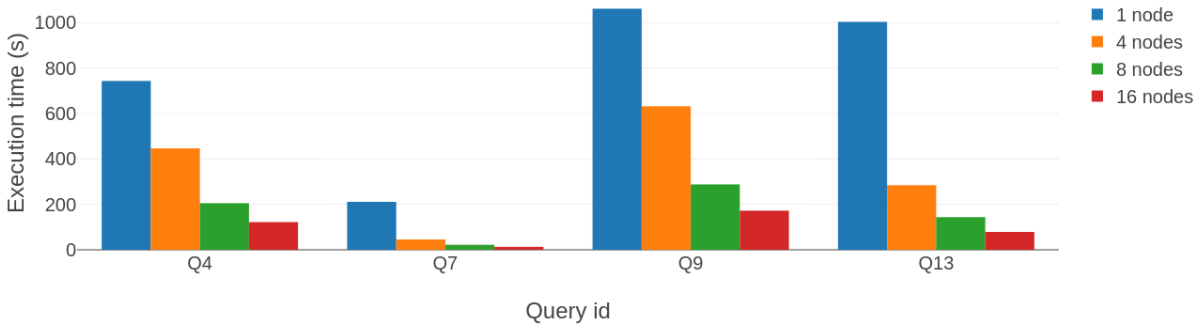

Figure 5.10: Performance insight the four queries by cluster size with scale factor=1

scale factor 1. As we can see, the Query 13 takes approximately 1000 seconds (about 16 minutes) in a single node. this later takes about 78 seconds (1.3 minute) in cluster size of 16 nodes. Table 5.7 shows the remaining results of the other queries on scale factor 1.

	Q4	Q7	Q9	Q13
1 node	743.5	211.13	1061.65	1003.48
4 nodes	446.65	45.47	631.86	284.33
8 nodes	205.25	21.93	287.9	143.75
16 nodes	121.8	13.02	172.35	78.57

Table 5.7: Summary of execution time for by query and cluster size for scale factor 1

To give more insight on the performance gained from single node to 16 nodes of the cluster size, we have used another method to show the performance gained with the distributed

workload. We have computed the Percentage Improvement (PI) that tells us how much a new environment can improves the average execution time of the queries executed in the current environment. The formula to compute the PI is as follow:

$$PI_{EN} = \frac{AET_E - AET_N}{AET_E} * 100$$

Where the AET_E is the average execution time on the existing environment and AET_N is the average execution time on the new environment. The PI_{EN} is the percentage improvement of the existing environment E in respect of the new environment N. Figure 5.11 show the PI of the single node environment by the environment with four nodes, eight nodes and 16 nodes with scale factor equal to one. If we inspect the PI of the Query 4, the average execution time is improved by 40% with the four nodes environment, 72% with the eight nodes environment and 83% with 16 nodes. As we can see the Query 7 is the most improved one with PI equal to 78% with the environment of nodes, 89% with eight nodes environment and 93% with 16 nodes environment. Table 5.8 show the details

Figure 5.11: Percentage improvement PI per size of the cluster with scale factor=1

of the other PI's queries.

	Q4	Q7	Q9	Q13
4 nodes	39.92	78.46	40.48	71.66
8 nodes	72.39	89.61	72.88	85.67
16 nodes	83.61	93.83	83.76	92.17

Table 5.8: Percentage improvement from single node to 16 nodes

5.6 Benchmark on Real-World Data

The experiments conducted on synthetic data generated using the BerlinMOD and SECONDO benchmarking tools, enabled us to assess the performance of a distributed environment for a moving object database across various data sizes. Furthermore, the queries offered by the BerlinMOD tool enable us to interact with the moving object database by using queries that simulate diverse scenarios, with different degrees of complexity. Although the benchmark tool offers certain benefits, it still diverges from reality. Consequently, utilizing real-world data for querying becomes essential to comprehensively evaluate our native solution within the context of tangible datasets. In this section, we

will introduce the AIS data employed in our experiments, providing details about the data source and the entity responsible for its provision. We have explored and drawn inspiration from the use cases developed in [30, 16], which utilize AIS data as the basis for their benchmarking activities.

5.6.1 AIS Dataset

AIS stand for Automatic Identification System which is a system used for navigation and collision detection of ships. Ships equipped with this type of system allow us to follow their trajectories in real time by providing information such as the registration of the vessel, the position defined by the longitude and latitude and then the detail on the trip such as the source and the destination location and the activity of the trip. To carry out our experiments, we will use the historical data of the trajectories of vessels provided by the Danish Maritime Authorities which use the AIS system to control and analyze the data of their maritime fleet as well as to make statistics or traffic patterns analysis. The historical data captured by AIS are organized by days, where each day has a set of row, and each row has a series of values. The description and the format of these values are defined in Table 5.9 that are downloaded from the official web site of the danish maritime authority.[3] Prior to proceeding with the tests, it was imperative to perform a

Column name	Format
Timestamp	Timestamp from the AIS base station, format: 31/12/2015 23:59:59
Type of mobile	Describes what type of target this message is received from (class A AIS Vessel, Class B AIS vessel, etc)
MMSI	MMSI number of vessel
Latitude	Latitude of message report (e.g. 57,8794)
Longitude	Longitude of message report (e.g. 17,9125)
ROT	Rot of turn from AIS message if available
SOG	Speed over ground from AIS message if available
COG	Course over ground from AIS message if available
Heading	Heading from AIS message if available
IMO	IMO number of the vessel
Callsign	Call sign of the vessel
Name	vessel name
Ship type	Describes the AIS ship type of this vessel
Cargo type	Type of cargo from the AIS message
Width	Width of the vessel
Length	Length of the vessel
Destination	Destination from AIS message
Navigational status	Navigational status from AIS message if available (e.g. engaged in fishing or under way using engine)

Table 5.9: AIS columns description

preprocessing step on the raw data sourced from the AIS system. This involved cleansing the data and organizing the records into distinct tables, a process that greatly facilitated the execution of SQL queries. The schema that will be used in our experiments which corresponds to the AIS vessel tracking data, is derived from the input data shown in

[3] `https://web.ais.dk/aisdata/`

Table 5.9. Once the data is captured in the PostgreSQL server and after cleaning phase, we organized our data model as described in Listing 5.8.

```
Ships(mmsi integer, trip tgeompoint, SOG tfloat, COG tfloat).
Periods(periodId integer, period tstzspan).
Ports(pid integer,latitude double precision, longitude double
    precision, Geom Geometry).
```

Listing 5.8: AIS schema

Where the `Periods` table contain a series of period depending on the number of days existing in the dataset. If we integrates two days of ASI data, this mean periods table should contain all the timestamp ranges existing in the input data. The `Ships` contains the list of ship with their daily computed trip. In another word, each raw from the ships table have a unique trip with it SOG and COG. The `Ports` table contains all the ports of Denmark organized by id and recorded with a unique latitude and longitude, the `geom` type is dedicated to represent a geographic point in the map that helps us tracking vessel and analysis. Geom field is computed from the longitude and the latitude with the PostGIS method. The `Ports` table is filled from the European Market Observatory for Fisheries and Aquaculture Products (EUMOFA)[4] that serve us to spot the danish ports.

5.6.2 AIS Data Integration

The Danish maritime authorities records daily ship tracking data which is open to the public where anyone can download it from their official website[5] for free. The integration step needs to be carried out efficiently and carefully because of the large volume of tracking data, where each file contains the tracking of the entire maritime fleet spread over 24 hours. Each line from these files corresponds to one tuple of information that are presented in Table 5.9. In our thesis, we are placing our emphasis on historical data or what is commonly referred to as data at rest. However, our exploration also extends to the realm of processing continuously data. This introduces another facet of our research, involving the continuous processing of moving object data using the capabilities of the PostgreSQL server in conjunction with its MobilityDB and Citus extensions. To accommodate this inherent characteristic, the automation of the integration stage could be a potential solution for that. Listing 5.9 describe the SQL script used to integrate the AIS data into the PostgreSQL server, more precisely in the coordinator node.

```
-- AISInput table creation
CREATE TABLE AISInput(T timestamp, TypeOfMobile varchar(50),
    MMSI integer, Latitude float, Longitude float, ROT float,
    SOG float, COG float, Heading integer, IMO varchar(50),
    Callsign varchar(50), Name varchar(100), ShipType varchar(50),
    CargoType varchar(100), Width float, Length float,
    Destination varchar(50), navigationalStatus varchar(100),
    Geom geometry(Point, 4326));
--Copyging the data into the table AISInput
\COPY AISInput(T, TypeOfMobile, MMSI, Latitude, Longitude,ROT,
    SOG, COG, Heading, Name, Width, Length, NavigationalStatus,
    Destination) FROM './aisdk-2023-03-07.csv' DELIMITER ','
```

[4] https://www.eumofa.eu/documents/20178/24415/annex+8_ports+seaboard.pdf/1690f53f-b5e3-4d57-974d-e2662492cac1?version=1.3

[5] https://web.ais.dk/aisdata/

```
    CSV HEADER;
-- Compute the geom field from latitude and longitude
UPDATE AISInput SET Geom = ST_SetSRID(ST_MakePoint(Longitude,
Latitude), 4326);
```

Listing 5.9: AIS data integration script

The `geom` field is added in the `AISInput` table in order to convert longitude and latitude coordinates to a geo-graphic point object that allows us to determines the positions of the ships in the map.

5.6.3 AIS Experiments

In order to keep consistency between the experiments on the theoretical data and the real data, we keep the same configurations of our GKE cluster infrastructure, the detail of the different configurations is written in Table 5.1. The volume of data integrated in these experiments relates to one day of historical data of the AIS system provided by the Danish maritime authorities. We chose the day 2023-03-07 for our experiments downloaded from the official website.[6] After having integrated the data of this day under the relational schema in Listing 5.8, the distribution of the tables through the Citus extension fundamental in order to idealize the performance of the analytical SQL queries. Listing 5.10 is the SQL script that allows to send the table `Ports` and `Periods` as reference table for all the worker nodes within the cluster. The queries used in these experiments are inspired from the MobilityDB workshop carried out by Mahmoud Sakr and Esteban Zimányi [31].

```
SELECT create_reference_table('Ports');
SELECT create_reference_table('periods_temp');
SELECT create_distributed_table('trips', 'mmsi');
```

Listing 5.10: AIS data distribution script

In the following, we explain the three SQL queries used to evaluate the performance for the four different GKE cluster configurations in terms of number of nodes in the cluster, CPU units and accumulated memory used. For each query, we associate a graph that shows the gain in performance between the three cluster sizes in order to show the impact of the horizontal scalability of a moving object database. Finally, we give an overview of the performance of the three queries for the different environments.

The first AIS query shown in Listing 5.11 has as objective, spotting ships that located in a given port or near a port with a perimeter radius equal to 500 meters. This are translated by a join between the Ships and Ports table in order to find the intersection of ships positions with a bounding box of 500 meters in length. Given that the Ports table is referenced and accessible across all nodes within the cluster, and considering the partitioning of the trips table into multiple shards that are distributed among various workers, the process of performing join operations for a substantial Trips table is greatly efficient.

```
SELECT T.mmsi, P.code, P.description
FROM ports P, trips T
WHERE ST_Intersects(trajectory(T.trip),
    ST_Transform(ST_MakeEnvelope(P.longitude,P.latitude,
```

[6] `https://web.ais.dk/aisdata/`

```
    P.longitude+0.001409, P.latitude+0.001409, 4326), 4326));
```

Listing 5.11: AIS Query 1

The second query shown in Listing 5.12 consist of knowing the distance traveled during the period, this requires an aggregation on the trip attribute of each ship with the period attribute of the periods_temp table. Since the "periods_temp" table is accessible and referenced across all worker nodes, it effectively aids in enabling localized joins of the shards belonging to the trips table within each individual node. As well as the aggregation operation (the sum in this example) is also calculated locally with trip values as input from the local partitions of the trips table and the period values from the periods_temp table.

```
SELECT T.mmsi, P.pid, P.Period, SUM(length(atTime(T.Trip,
    P.Period))) AS Dist
FROM Trips T, periods_temp P
WHERE T.Trip && P.Period
GROUP BY T.mmsi, P.pid, P.Period
ORDER BY T.mmsi, P.pid;
```

Listing 5.12: AIS Query 2

The Query 3 is used from the MobilityDB workshop carried out in [31], this query is a version adapted for AIS tracking vessel data which consists of knowing the port which is visited the most by a maximum of different ship. This query is very expensive in terms of memory space which needs to keep an intermediate result in memory in order to proceed to intersect the trajectories of the ships with the peripheral of the port.

```
WITH PortCount AS (SELECT P.code, COUNT(DISTINCT T.mmsi)
    AS Hits FROM Trips T, ports P
    WHERE ST_Intersects(trajectory(T.Trip)
        ST_Transform(ST_MakeEnvelope(P.longitude,P.latitude,
        P.longitude+0.01, P.latitude+0.01, 4326), 4326))
    GROUP BY P.code)
SELECT p.code, Hits
FROM PortCount AS P
WHERE P.Hits = (SELECT MAX(Hits) FROM PortCount);
```

Listing 5.13: AIS Query 3

Table 5.10 highlight the results of AIS queries for different cluster sizes. After having parallelized the three queries through the Citus extension of PostgreSQL, we can clearly see that the performance was improved from one configuration to another. This improvement is the result of the global index implemented on top of the distributed database as well as the partitioning of the fact tables which makes joining, aggregation and other operations faster.

Figure 5.12 show the performance gained by the four different cluster sizes (one,four,eight,16 nodes) implemented. As can be seen, the query which takes at least 780 seconds in a single node, this is improved in 448 seconds in a distributed environment with four nodes and 180 seconds in a cluster of eight nodes, 80 seconds in 16 nodes. To conclude, the execution time is reduced to almost 50% if we double the size of the cluster.

In the next chapter we will talk about the desired future perspectives developed in relation

	Q1	Q2	Q3
1 node	782.56	122.56	720.81
4 nodes	448.32	43.83	456.34
8 nodes	179.03	32.08	192.2
16 nodes	80.09	10.95	116.72

Table 5.10: AIS queries result for one day of AIS data

Figure 5.12: Performance of AIS queries by cluster size

to our work and thus give an overview of other approaches and technological tools that may be useful to explore in order to meet our objectives. We end with the conclusion of this thesis.

Chapter 6

Conclusion

The completion of our thesis work provides a database as a service hosted in the Google Cloud platform in order to support geo-spatial and spatial-temporal data or even moving objects data that are usually generated rapidly over time, these data are manipulated and managed by the MobilityDB extension of PostgreSQL server. Our cloud-native solution makes the hosted database supported by MobilityDB and fully distributive horizontally through the Kubernetes system which ensures the elasticity of the cluster, and the PostgreSQL Citus extension which partitions and distributes the spatial-temporal tables and queries within the cluster. Analytical queries that interrogates those type of data are much more complex due to geometric data types such as polygon, line, graph and regions. As a result, the query is parallelized by the whole cluster's machine in order to idealize the performance by reducing the execution time. Our approach and implementation are versatile and extendable and it could be adapted to other types of database.

In this chapter, we conclude our thesis work where we summarize the different parts of this document and giving an overview of its problems encountered and the contributions added to meet the objectives. These will be presented in Section 6.1. Thereafter, we discuss the future perspectives discovered during the development of each part of our thesis, in particular the exploratory research part of the methodology and technological tools which can satisfy our objectives or which can improve our native cloud solution, this latter will open new rooms of research in this domain that aim to enhance or extend our solution. This part will be developed in Section 6.2 future perspectives.

6.1 Thesis Summary

In this part we will clearly describe the result obtained from each chapter and highlight the key contributions that allow us to overcome the challenges of our subject. To begin with, the **introductory** part was fundamental in order to identify the problem of our subject as well as setting the objectives that we wish to achieve, this helps us to clarify the field of research that we must explore and to determine the scope of the technologies that need to be investigated. Among the main objectives of this work is to provide as a service a scalable database in a distributive environment in GCP, in order to allow the user querying and analyzing geo-spatial and spatial-temporal data through the PostgreSQL MobilityDB extension efficiently. In addition, the automation of the infrastructure initialization stage and the deployment of the necessary applications as well as ensuring high availability, security and data integrity. Furthermore, the feature factorization and spotting the common prerequisites between our solution and the previous ones (Mobili-

tyDB for Azure and AWS) it needed in order to found a cloud-native implementation of MobilityDB that can seamlessly function across the three major cloud platforms with a minimum of configuration and specific development.

Chapter two dedicated to the **state of the art** allows us to understand in depth our field of study, in particular the database as a service with their environment, their functionalities and the logical architecture and materials implemented behind, in order to to support customer requests. In order to know the different best-known suppliers who offer this type of service in order to be able to make a comparison between their systems in place which provide a database as a service. Through the exploratory research carried out to understand the approaches and methodology that allow distributed databases, we have gained knowledge of the main mechanisms of partitioning, and distribution of data and SQL queries within a group of machines. Thus to know the factors that contribute to the scalability of the database over time. In addition, the understanding of geo-spatial and spatial-temporal data has allowed us to understand their characteristics and help us to estimate the degree of complexity of SQL queries interrogating those data. While the exploration of platform systems that manipulate the same type of database that we deal with in our work, in particular moving object database in the cloud and the Kubernetes system, has allowed us to know the adequate functionalities to deploy our solution in the cloud efficiency. After the comparison part between the possible system that fulfill our objective, Kubernetes cluster is the most appropriate one, because of it portability, extensibility and rendering it adaptable across various cloud provider as a product coupled with other products that solidifies our implementation. After choosing the appropriate system that supports our application and defining the architecture of the necessary infrastructure that ensures our prerequisites (scalability, integrity and extensibility) of our future solution. In the **technical background** chapter, we describe all the technologies and systems used, in particular the Docker system which is used to create our native-cloud container image which includes all the necessary functionalities, including the PostgreSQL server accompanied by the MobilityDB extension and Citus as well as other libraries. In order to deploy our container image, the Kubernetes orchestrator is adopted in our solution which ensures the deployment of our application, this system ensures the high availability and horizontal scalability of our database. This is achieved by an elastic cluster that can change its size according to the demand of the workload. The K8s system API greatly facilitates the deployment that provides the ability to manage the infrastructure through YAML code, regarding our application, a series of YAML declarations have been provided in order to automates the deployment phase. The documentation of the Docker system and the Kubernetes ecosystem made it easier for us to use the appropriate services and functionality for our deployment. The GCP platform offers the possibility of integrating a Kubernetes deployment through its Google Kubernetes Engine product, which has the advantage of coupling with other GCP services such as the identity and authenticity management service for IAM users, the monitoring, networking and storage and more. The documentation on some GCP services was necessary in order to understand the functioning and manipulation of our infrastructure within this giant platform. At the end of this chapter, the Python client was presented in order to communicate with the API of the GKE cluster and the API of the Kubernetes system hosted in GKE. This communication is necessary in order to initialize, manage and change the size of the GKE cluster through the commands implemented on the user side of our solution. This automates certain repetitive steps, but also ensures data integrity in the event of a scale-out or scale-in of the cluster.

the chapter **distributed database management on GKE engine** presents the work-

flow of the implementation of our native cloud solution. After creating the GKE cluster by allocating GKE resources including virtual machines and storage classes. The outcome of this chapter is to provide a list of YAML files that allow the deployment of our previously designed container image. This allows the initialization of the Citus cluster inside the GKE cluster where each node of the GKE cluster corresponds to a PostgreSQL server which is intended to be either a worker node or a coordinator node. In addition, the scale-out and scale-in operation are implemented on the client side which allows to change the size of the cluster by the desired number of nodes. The purpose of these two methods is to preserve the integrity of the data, more precisely during the scale-in operation, one of the nodes must be deleted, this induces a drainage of this node which consists in redistributing the data of this node for the other active nodes. To showcase our native cloud solution, **the experiment and performance evaluation** chapter is dedicated to testing our distributive database first with theoretical data through the BerlinMOD generator and on real data from the AIS system. The results of both experiments show the significant performance gain for a larger cluster size. As well as the percentage of improvement PI it increases according to the number of node added to the cluster.

6.2 Future Perspectives

We chose to utilize the Kubernetes system because it offers the advantage of supporting portability across different cloud platforms and the ability to scale, allowing us to incorporate new features in the future. Among the key enhancements we plan to implement for our solution are as follows:

- **Multi-Cloud Integration:** Our foremost goal is to ensure seamless operation across various cloud platforms, with a particular focus on AWS and Azure. This involves creating a centralized infrastructure that empowers us to leverage diverse services offered by distinct cloud providers. Consequently, we can utilize GCP's storage class for data retention while harnessing AWS's Elastic Cloud Compute (EC2) for analytical queries. Furthermore, we aim to replicate our deployment across different regions and zones offered by various cloud providers. For instance, we can duplicate the deployment in both AWS-managed eu-west-1 region in Ireland and the europe-west-1 region located in St. Ghislain, Belgium. The integration of multi-cloud capabilities will substantially elevate MobilityDB's potential beyond its current scope.
- **Large-Scale Experimentation:** To comprehensively assess the performance of our distributed environment, we intend to conduct extensive experiments using substantial volumes of moving object data. A practical example would be analyzing AIS data from the last five months, which can easily accumulate over 350 gigabytes on average.
- **Robust Scaling Mechanism:** As historical data continuously expands, the ensuing performance degradation becomes a concern. Our solution involves the development of a robust scaling mechanism that automatically adjusts the cluster's resources. This alignment ensures that storage and computational capacities of each node are optimized, thereby preventing undue strain during local node computations. The envisaged tool will base its decisions on factors such as the volume of existing rows in the table and the influx of rows from the data source. This approach will determine the number of nodes to be added to regulate performance effectively.
- **Enhanced Python API:** Addressing the time-intensive process of integrating his-

torical data, we plan to enhance our Python API on the client side. This will facilitate effortless connections to data sources, automating the population of tables. Moreover, we aspire to empower our API to execute ETL (Extract, Transform, Load) jobs, either by embedding these jobs within the implementation or by routing the data flow to an external ETL tool for data cleansing.

Appendices

Appendix A

Tutorial

In this tutorial, we will create a workflow that outlines the necessary steps to deploy a GKE cluster, aiming to establish a distributed environment for a PostgreSQL database. The updated code used in this tutorial it can be found in `https://github.com/bouzouidja/MobilityDB-GCP` or `https://github.com/MobilityDB/MobilityDB-GCP/`.

A.1 GKE Cluster Initialization

First and foremost, you need to possess a Google account to link it within the GCP console. If you already have an existing Google account, you can associate it with GCP by signing in here: "https://console.cloud.google.com/". Once you have a GCP account, it's necessary to acquire credits to utilize GCP services through the billing account. For further information about GCP credits and billing accounts, please refer to this link: "https://console.cloud.google.com/billing/".

Next, the subsequent steps detail the process of setting up a GKE cluster.

- **Create GCP project:** Once you have created the billing account with active credits on it, create GCP project and associate the billing account with it. The project can be created through the GCP console or by commands line from your local machine. You can find the full documentation on how to create GCP project in `https://cloud.google.com/resource-manager/docs/creating-managing-projects?hl=en#gcloud`. The following command creates a GCP project with the desired project_id.

```
gcloud projects create PROJECT_ID
```

- **Enable GKE product:** Enabling GKE product through the activation process here `https://cloud.google.com/kubernetes-engine/docs/how-to/workload-identity`
- **Create GKE cluster:** Creating GKE cluster with the full understanding of it parameters, you can visit the documentation `https://cloud.google.com/kubernetes-engine/docs/how-to/workload-identity#enable_on_cluster`. Or use the following command with the necessary parameters for our cluster.

```
gcloud container clusters create mobilitydb-cluster --zone europe-west1-c
--node-pool mobilitydb-node-pool --machine-type e2-standard-4
--disk-type pd-balanced --num-nodes 4
```

- **Connect to GKE cluster:** We assume that the **kubectl** Kubernetes client is already installed.

```
# Get the credentials of the cluster
gcloud container clusters get-credentials mobilitydb-cluster
--zone europe-west1-c --project PROJECT_ID
# View your cluster information
gcloud container clusters describe mobilitydb-cluster-1
--zone europe-west1-c
# You should see your cluster created after few
# minutes in your GCP console
# View your cluster nodes information via the kubectl command
kubectl get node -o wide
```

A.2 Citus Cluster Initialization

Once the GKE cluster is established, you can deploy our cloud-native solution to initialize the Citus cluster. To accomplish this, you must first perform the following operations.

- **Clone MobilityDB-GCP solution:**

```
git clone https://github.com/MobilityDB/MobilityDB-GCP
```

- **Metadata configuration:** Adapt the file postgres-secret.yaml to update your confidential information including POSTGRES_USER, POSTGRES_PASSWORD and POSTGRES_DB

- **Export your environment variables:**

```
export POSTGRES_USER=your-user
export POSTGRES_PASSWORD=your-password
export POSTGRES_DB=your-db-name
export POSTGRES_PORT=30001
source ~/.bashrc
```

- **Clone the cloud native solution in the docker registry:**

```
# Create the docker registry in GCP artifact registry
gcloud artifacts repositories create repo-test --repository-format=docker
--location=europe-west1 --description==testing --immutable-tags --async
# Pull the cloud native docker image from the docker hub
docker pull bouzouidja/mobilitydb-cloud:latest
# Tag the pulled docker image mobilitydb-cloud into the docker
# registry repo-test
docker tag bouzouidja/mobilitydb-cloud:latest europe-west1-docker.pkg.dev/
PROJECT_ID/repo-test/mobilitydb-cloud:latest
# Push the taged image to the repo-test docker registry
docker push bouzouidja/mobilitydb-cloud:latest europe-west1-docker.pkg.dev/
PROJECT_ID/repo-test/mobilitydb-cloud:latest
```

- **Open the port 30001 to accept requests from the outside of the GKE cluster:**

```
gcloud compute firewall-rules create mobilitydb-node-port
--project PROJECT_ID --allow tcp:30001
```

- **Deploy the application YAML files:**

```
# Execute those commands simultaneously, and wait until the Pods will
# be in the running state
kubectl create -f postgres-secret.yaml
kubectl create -f coordinator-deployment.yaml
kubectl create -f workers-deployment.yaml
# View your Pods
kubectl get pods -o wide
# You should see the coordinator Pod and the workers Pods running
```

- **Test your PostgreSQL:**

```
# Access to your PostgreSQL database from the coordinator using
# the psql client
psql -h 34.34.151.202 -U docker -p 30001 -d mobilitydb
mobilitydb=#
# 34.34.151.202 is the external IP address of the coordinator
# node in my created cluster, it can be found using the
# command kubectl get node -o wide
NAME STATUS ROLES AGE VERSION INTERNAL-IP EXTERNAL-IP OS-IMAGE
KERNEL-VERSION CONTAINER-RUNTIME
gke-mobilitydb-clust-mobilitydb-node--77a55075-cssr Ready <none> 121m
v1.27.3-gke.100 10.132.15.217 34.34.151.202 Container-Optimized OS
from Google 5.15.109+ containerd://1.7.0
gke-mobilitydb-clust-mobilitydb-node--77a55075-l2wd Ready <none> 121m
v1.27.3-gke.100 10.132.15.220 35.187.85.28 Container-Optimized OS
from Google 5.15.109+ containerd://1.7.0
gke-mobilitydb-clust-mobilitydb-node--77a55075-ldgc Ready <none> 121m
v1.27.3-gke.100 10.132.15.219 34.77.253.167 Container-Optimized OS
from Google 5.15.109+ containerd://1.7.0
gke-mobilitydb-clust-mobilitydb-node--77a55075-wcl8 Ready <none> 121m
v1.27.3-gke.100 10.132.15.218 35.205.205.195 Container-Optimized OS
from Google 5.15.109+ containerd://1.7.0
```

- **View the worker nodes:**

```
mobilitydb=# SELECT * from citus_get_active_worker_nodes();
 node_name | node_port
-----------+-----------
 10.48.1.15 |     5432
 10.48.2.8 |     5432
 10.48.3.7 |     5432
(3 rows)
```

A.3 Horizontal Scaling

It is recommended to scale out the cluster when the database becomes large in order to optimize performance. Scaling out involves adding more nodes to the existing cluster. Since PostgreSQL tables are distributed across GKE worker nodes through the Citus extension, adding new nodes triggers the redistribution of tables to evenly distribute

partitions across the cluster that incorporates the new nodes. In other words, adding more nodes translates to an increase in computational capacity, including CPU cores and memory. In the event of scale-out or scale-in operations, it is essential to execute the following command to ensure data consistency. This command triggers a Python script designed to automate the redistribution of the cluster when new nodes are added and to manage data evacuation in the event of node deletion.

```
# Make sure to export your environment variables, including POSTGRES_USER,
# POSTGRES_PASSWORD, POSTGRES_PORT and POSTGRES_DB.
python citus_cluster_management.py resize --cluster-name mobilitydb-cluster-1
--cluster-zone europe-west1-b --cluster-project PROJECT_ID --num-nodes 8
# Waiting the completion of all operations.
```

Appendix B

Citus Cluster Deployment:YAML Declaration

B.1 Secret Parameters

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: postgres-secret
type: Opaque
data:
  POSTGRES_USER: ZG9ja2Vy #docker
  POSTGRES_PASSWORD: ZG9ja2Vy #docker
  POSTGRES_DB: bW9iaWxpdHlkYg== #moving-object-db
# Always encode the secret parameters with:
# echo -n your-parameter-value |base64
```

B.2 Coordinator StatefulSet

```yaml
# Statefulset declaration
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: citus-coordinator
  labels:
    app: citus-coordinator
spec:
  serviceName: citus-coordinator
  replicas: 1
  minReadySeconds: 10
  selector:
    matchLabels:
      app: citus-coordinator
  template:
    metadata:
      labels:
        app: citus-coordinator
```

```
19      spec:
20        terminationGracePeriodSeconds: 10
21        affinity:
22          podAntiAffinity:
23            requiredDuringSchedulingIgnoredDuringExecution:
24              - labelSelector:
25                  matchExpressions:
26                    - key: "app"
27                      operator: In
28                      values:
29                      #- citus-workers
30                      - citus-coordinator
31                topologyKey: "kubernetes.io/hostname"
32        containers:
33        - name: postgresdb
34          #image: bouzouidja/mobilitydb-cloud:latest
35          image: >
36              europe-west1-docker.pkg.dev/distributed-postgresql-
37              82971/sidahmed-gcp-registry/mobilitydb-cloud
38          imagePullPolicy: "IfNotPresent"
39          args:
40              - -c
41              - max_locks_per_transaction=128
42              - -c
43              - shared_preload_libraries=citus, postgis-3.so,
44               pg_stat_statements
45              - -c
46              - wal_level=logical
47              - -c
48              - listen_addresses=*
49
50          ports:
51          - containerPort: 5432
52          #resources:
53          #  limits:
54          #    cpu: 3500m
55          #    memory: 10G
56          #  requests:
57          #    cpu: 3000m
58          #    memory: 8G
59          lifecycle:
60              postStart:
61                exec:
62                  command:
63                  - /bin/sh
64                  - -c
65                  - |
66                    sleep 5
67                    psql -U $POSTGRES_USER -d $POSTGRES_DB --command=
68                    "CREATE EXTENSION IF NOT EXISTS citus;
```

```
                CREATE EXTENSION IF NOT EXISTS hstore;
                CREATE EXTENSION IF NOT EXISTS mobilitydb cascade;
                CREATE EXTENSION IF NOT EXISTS pgrouting;"
              psql -U $POSTGRES_USER -d $POSTGRES_DB --command=
              "CREATE TABLE IF NOT EXISTS
               pod_information (pod_ip VARCHAR(20),
               nodeid INTEGER, nodeport INTEGER,
               pod_hostname VARCHAR(40));"
              psql -U $POSTGRES_USER -d $POSTGRES_DB --command=
              "SELECT citus_set_coordinator_host('$POD_IP', 5432);"
              psql -U $POSTGRES_USER -d $POSTGRES_DB --command="
               INSERT INTO pod_information(pod_ip, nodeid, nodeport,
               pod_hostname)
               SELECT nodename, nodeid, nodeport, '${HOSTNAME}'
               FROM pg_dist_node WHERE nodename = '$POD_IP' and
               NOT EXISTS (SELECT pod_ip FROM pod_information
               WHERE pod_ip = '$POD_IP') and NOT EXISTS
               (SELECT pod_hostname FROM pod_information
               WHERE pod_hostname = '${HOSTNAME}');"

      env:
      - name: POD_IP
        valueFrom:
          fieldRef:
            fieldPath: status.podIP
      - name: MY_NODE_NAME
        valueFrom:
          fieldRef:
            fieldPath: spec.nodeName
      - name: MY_POD_NAME
        valueFrom:
          fieldRef:
            fieldPath: metadata.name

      - name: POSTGRES_USER
        valueFrom:
          secretKeyRef:
            name: postgres-secret
            key: POSTGRES_USER
      - name: POSTGRES_PASSWORD
        valueFrom:
          secretKeyRef:
            name: postgres-secret
            key: POSTGRES_PASSWORD
      - name: PGPASSWORD
        valueFrom:
          secretKeyRef:
            name: postgres-secret
            key: POSTGRES_PASSWORD
      - name: POSTGRES_DB
```

```
          valueFrom:
            secretKeyRef:
              name: postgres-secret
              key: POSTGRES_DB

        - name: PGDATA
          value: /var/lib/postgresql/data/pgdata
        volumeMounts:
            - mountPath: /var/lib/postgresql/data
              name: postgredb

      volumes:
        - name: postgredb
          persistentVolumeClaim:
            claimName: postgres-pv-claim-coordinator

# Volume declaration
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pv-claim-coordinator
  labels:
    app: citus-coordinator
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
  storageClassName: standard-rwo

# Service declaration
apiVersion: v1
kind: Service
metadata:
  name: citus-coordinator
  labels:
    app: citus-coordinator
spec:
  selector:
    app: citus-coordinator
  type: NodePort
  ports:
    - port: 5432
      nodePort: 30001
```

B.3 Citus Worker StatefulSet

```

# Citus worker StatefulSet declaration
```

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: citus-workers
spec:
  serviceName: "citus-workers"
  replicas: 15
  selector:
    matchLabels:
      app: citus-workers
  template:
    metadata:
      labels:
        app: citus-workers

    spec:
      #terminationGracePeriodSeconds: 160
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: "app"
                    operator: In
                    values:
                    - citus-workers
                    - citus-coordinator
              topologyKey: "kubernetes.io/hostname"

      containers:
      - name: mobilitydb-cloud
        #image: bouzouidja/mobilitydb-cloud:latest
        image: >
            europe-west1-docker.pkg.dev/distributed-postgresql-
            82971/sidahmed-gcp-registry/mobilitydb-cloud
        imagePullPolicy: "IfNotPresent"
        args:
            - -c
            - max_locks_per_transaction=128
            - -c
            - shared_preload_libraries=citus, postgis-3.so
            - -c
            - wal_level=logical
        ports:
        - containerPort: 5432
        #resources:
        # limits:
        #   cpu: 3500m
        #    memory: 14G
        #  requests:
```

```
        #    cpu: 3000m
        #    memory: 12G

        lifecycle:
            postStart:
              exec:
                command:
                - /bin/sh
                - -c
                - |
                  sleep 9
                  psql -U $POSTGRES_USER -d $POSTGRES_DB --command=
                  "CREATE EXTENSION IF NOT EXISTS citus;
                  CREATE EXTENSION IF NOT EXISTS hstore;
                  CREATE EXTENSION IF NOT EXISTS mobilitydb cascade;
                  CREATE EXTENSION IF NOT EXISTS pgrouting;"
                  psql -h citus-coordinator -U $POSTGRES_USER
                  -d $POSTGRES_DB --command=
                  "SELECT * from citus_add_node('$POD_IP', 5432)
                   WHERE NOT EXISTS (SELECT pod_ip FROM pod_information
                   WHERE pod_ip = '$POD_IP');"
                  psql -h citus-coordinator -U $POSTGRES_USER
                  -d $POSTGRES_DB --command=
                  "INSERT INTO pod_information(pod_ip, nodeid, nodeport,
                   pod_hostname)
                  SELECT nodename, nodeid, nodeport, '${HOSTNAME}'
                  FROM pg_dist_node WHERE nodename = '$POD_IP' and
                  NOT EXISTS (SELECT pod_ip FROM pod_information
                  WHERE pod_ip = '$POD_IP') and NOT EXISTS
                   (SELECT pod_hostname FROM pod_information
                  WHERE pod_hostname = '${HOSTNAME}');"
                  psql -h citus-coordinator -U $POSTGRES_USER
                  -d $POSTGRES_DB --command=
                  "SELECT citus_update_node((select nodeid
                  FROM pod_information
                  WHERE pod_hostname='${HOSTNAME}'),'$POD_IP',
                   5432) FROM pg_dist_node WHERE nodename =
                  (SELECT pod_ip FROM pod_information
                  WHERE pod_hostname='${HOSTNAME}');
                  UPDATE pod_information SET pod_ip = '$POD_IP'
                  WHERE nodeid = (select nodeid
                  FROM pod_information
                  WHERE pod_hostname='${HOSTNAME}');"

        env:
        - name: POSTGRES_USER
          valueFrom:
            secretKeyRef:
              name: postgres-secret
              key: POSTGRES_USER
```

```
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: postgres-secret
              key: POSTGRES_PASSWORD
        - name: PGPASSWORD
          valueFrom:
            secretKeyRef:
              name: postgres-secret
              key: POSTGRES_PASSWORD
        - name: POSTGRES_DB
          valueFrom:
            secretKeyRef:
              name: postgres-secret
              key: POSTGRES_DB

        - name: PGDATA
          value: /var/lib/postgresql/data/pgdata
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        volumeMounts:
            - mountPath: /var/lib/postgresql/data
              name: postgres-pv-claim-worker
  volumeClaimTemplates:
  - metadata:
      name: postgres-pv-claim-worker
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "standard-rwo"
      resources:
        requests:
          storage: 8Gi

# Citus worker service declaration
apiVersion: v1
kind: Service
metadata:
  name: citus-workers
  labels:
    app: citus-workers
spec:
  ports:
  - port: 5432
    name: postgres
  clusterIP: None
  selector:
    app: citus-workers
```

Appendix C

Experiments Script

```
import time
import psycopg2
import psycopg2.extensions
from psycopg2.extras import LoggingConnection, LoggingCursor
import logging
import statistics
import csv

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)
list_queries=[{'Q2':"\
        SELECT COUNT (Licence)\
        FROM Vehicles C\
        WHERE Type = 'passenger';\
        "},
{'Q3':"\
        SELECT DISTINCT L.Licence, I.InstantId, I.Instant AS Instant,\
        valueAtTimestamp(T.Trip, I.Instant)\
        AS Pos\
        FROM Trips T, Licences1 L, Instants1 I\
        WHERE T.VehId = L.VehId AND\
        valueAtTimestamp(T.Trip, I.Instant)\
        IS NOT NULL\
        ORDER BY L.Licence, I.InstantId; \
        "},
{'Q4':"\
        SELECT DISTINCT P.PointId, P.Geom, C.Licence\
        FROM Trips T, Vehicles C, Points P\
        WHERE T.VehId = C.VehId AND T.Trip && stbox(P.Geom) AND\
        ST_Intersects(trajectory(T.Trip), P.Geom)\
        ORDER BY P.PointId, C.Licence;\
        "},
{'Q7':"\
        WITH Timestamps AS (\
        SELECT DISTINCT C.Licence, P.PointId, P.Geom,\
        MIN(startTimestamp(atValues(T.Trip,P.Geom))) AS Instant\
```

```
        FROM Trips T, Vehicles C, Points1 P\
        WHERE T.VehId = C.VehId AND C.Type = 'passenger' AND\
        T.Trip && stbox(P.Geom) AND ST_Intersects(trajectory(T.Trip),\
        P.Geom)\
        GROUP BY C.Licence, P.PointId, P.Geom )\
        SELECT T1.Licence, T1.PointId, T1.Geom, T1.Instant\
        FROM Timestamps T1\
        WHERE T1.Instant <= ALL (\
        SELECT T2.Instant\
        FROM Timestamps T2\
        WHERE T1.PointId = T2.PointId )\
        ORDER BY T1.PointId, T1.Licence;\
        "},
{'Q9':"\
        WITH Distances AS (\
        SELECT P.PeriodId, P.Period, T.VehId,\
        SUM(length(atTime(T.Trip, P.Period)))\
         AS Dist FROM Trips T, Periods P\
        WHERE T.Trip && P.Period\
        GROUP BY P.PeriodId, P.Period, T.VehId )\
        SELECT PeriodId, Period, MAX(Dist) AS MaxDist\
        FROM Distances\
        GROUP BY PeriodId, Period\
        ORDER BY PeriodId;\
        "},
{'Q13':"\
        SELECT DISTINCT R.RegionId, P.PeriodId, P.Period, C.Licence\
        FROM Trips T, Vehicles C, Regions1 R, Periods1 P\
        WHERE T.VehId = C.VehId AND T.trip && stbox(R.Geom, P.Period)\
        AND ST_Intersects(trajectory(atTime(T.Trip, P.Period)),\
        R.Geom) ORDER BY R.RegionId, P.PeriodId, C.Licence;\
        "},
        ]
POSTGRES_USER="docker"
POSTGRES_PASSWORD="docker"
POSTGRES_HOST="35.241.141.97"
POSTGRES_PORT="30001"

list_db=[('POSTGRES_DB_S005', "brussels005"), ('POSTGRES_DB_S02',
"brussels02"), ('POSTGRES_DB_S05',"brussels05"),
('POSTGRES_DB_S1',"brussels-s1") ]

def  average_execution_time(cursor,query):
    iterations=[]
    for iter in range(1,6):
        start_time= time.time()
        cursor.execute(query)
        end_time= time.time()
        res =(end_time - start_time)
        iterations.append(res)
```

```
87          print("\nQuery is in iteration "+str(iter)+": "+str(res)+" s")
88      avg =statistics.mean(iterations[1:])
89      print(str(iter)+" iterations are finished", iterations,
90      "\n The average=",avg)
91      return round(avg,2)
92
93  def fill_configuration_results():
94      config_res=[]
95      for db in list_db:
96          print("New database connection..into",db[1])
97          conn = psycopg2.connect(database=db[1], user=POSTGRES_USER,
98           password=POSTGRES_PASSWORD, host=POSTGRES_HOST,
99           port=POSTGRES_PORT)
100         cur = conn.cursor()
101         print("\n Start executing BerlinMOD queries:")
102         for query in list_queries:
103             average_execution = average_execution_time(cur,
104             query[list(query.keys())[0]])
105             if query[list(query.keys())[0]] not in \
106             [li[list(li.keys())[0]] for li in config_res]:
107                 new_res= query
108                 new_res['sf_'+db[1]]=average_execution
109                 config_res.append(new_res)
110                 print("\nNew query in config list.",new_res)
111             else:
112                 query_from_config=next(qr for qr in config_res\
113                 if qr[list(qr.keys())[0]] == query[
114                 list(query.keys())[0]])
115                 query_from_config['sf_'+db[1]]=average_execution
116                 print("\nExisting query was updated.",
117                 query_from_config)
118
119     print("\n Final list of configuration result: ",config_res)
120     return config_res
121
122 experiments=fill_configuration_results()
123 print("Final experiments result:\n",experiments)
124 result_file = open("experiment_result3.txt", "w")
125 for ex in experiments:
126     result_file.write(str(ex.items()))
127 result_file.close()
128
129 with open('cluster-config3-result.csv', 'w', newline='') as csvfile:
130     spamwriter = csv.writer(csvfile, delimiter=',',
131      quoting=csv.QUOTE_MINIMAL)
132     spamwriter.writerow(['Query', 'sf005', 'sf02','sf05','sf1'])
133     for ex in experiments:
134         spamwriter.writerow([list(ex.keys())[0],
135         ex[list(ex.keys())[1]], ex[list(ex.keys())[2]],
```

```
136         ex[list(ex.keys())[3]], ex[list(ex.keys())[4]]])
```

Listing C.1: Python script used to evaluates the BerlinMOD qeuries

Bibliography

[1] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop gis: A high performance spatial data warehousing system over mapreduce. *Proc. VLDB Endow.*, 6(11):1009–1020, aug 2013.

[2] Mudabber Ashfaq, Ali Tahir, Faisal Moeen Orakzai, Gavin McArdle, and Michela Bertolotto. Using t-drive and berlinmod in parallel secondo for performance evaluation of geospatial big data processing. In *Spatial Data Handling in Big Data Era*, Advances in Geographic Information Science, pages 3–19, Singapore, 2017. Springer Singapore.

[3] Mohamed Bakli, Mahmoud Sakr, and Esteban Zimányi. Distributed spatiotemporal trajectory query processing in sql. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '20, page 87–98, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Mohamed S. Bakli, Mahmoud A. Sakr, and Taysir Hassan A. Soliman. A spatiotemporal algebra in hadoop for moving objects. *Geo-spatial information science*, 21(2):102–114, 2018.

[5] Jonathan Bartlett. *Cloud Native Applications with Docker and Kubernetes: Design and Build Cloud Architecture and Applications with Microservices, EMQ, and Multi-Site Configurations*. Apress L. P, Berkeley, CA, 2022.

[6] Thomas Behr. Secondo berlinmod. `https://secondo-database.github.io/BerlinMOD/BerlinMOD.html`, 2011.

[7] Harrison John Bhatti and Babak Bashari Rad. Databases in cloud computing: A literature review. *International Journal of Information Technology and Computer Science*, 9(4):9–17, 2017.

[8] Adam Broniewski, Mohammad Ismail Tirmizi, Esteban Zimányi, and Mahmoud Sakr. Using mobilitydb and grafana for aviation trajectory analysis. *Engineering proceedings*, 28(17):17–, 2023.

[9] Lei Chen, Christian S Jensen, Cyrus Shahabi, Xiaochun Yang, and Xiang Lian. Trajspark: A scalable and efficient in-memory management system for big trajectory data. In *Web and Big Data*, volume 10366 of *Lecture Notes in Computer Science*, pages 11–26. Springer International Publishing AG, Switzerland, 2017.

[10] Kubernetes client. kubernetes.client.appsv1api. `https://github.com/kubernetes-client/python/blob/master/kubernetes/docs/AppsV1Api.md`, 2023.

[11] Kubernetes client. kubernetes.client.corev1api. https://github.com/kubernetes-client/python/blob/master/kubernetes/docs/CoreV1Api.md, 2023.

[12] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. Citus: Distributed postgresql for data-intensive applications. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2490–2502, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting. Berlinmod: a benchmark for moving object databases. *The VLDB journal*, 18(6):1335–1368, 2009.

[14] Ahmed Eldawy and Mohamed F. Mokbel. A demonstration of spatialhadoop: An efficient mapreduce framework for spatial data. *Proc. VLDB Endow.*, 6(12):1230–1233, aug 2013.

[15] Ahmed Eldawy and Mohamed F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. *2015 IEEE 31st International Conference on Data Engineering*, pages 1352–1363, 2015.

[16] Laurent Etienne, Cyril Ray, Elena Camossi, and Clément Iphar. *Maritime Data Processing in Relational Databases*, pages 73–118. Springer International Publishing, Cham, 2021.

[17] Google. Overview of mig capabilities and common workloads. https://cloud.google.com/compute/docs/instance-groups, 2023.

[18] The PostgreSQL Global Development Group. What is postgresql. https://www.postgresql.org/docs/16/intro-whatis.html, 2023.

[19] Ralf Güting, Thomas Behr, and Christian Düntgen. Secondo: A platform for moving objects database research and for publishing and integrating research implementations. *IEEE Data Eng. Bull.*, 33:56–63, 06 2010.

[20] Shimon Ifrah. *Getting started with containers in Google Cloud Platform : deploy, manage, and secure containerized applications*. Apress, Place of publication not identified, 1st ed. 2021. edition, 2021.

[21] Manolis Koubarakis, Timos Sellis, Andrew U Frank, Stéphane Grumbach, Ralf Hartmut Güting, Christian S Jensen, Nikos Lorentzos, Yannis Manolopoulos, Enrico Nardelli, and Barbara Pernici. *Spatio-Temporal Databases: The CHOROCHRONOS Approach*, volume 2520 of *Lecture Notes in Computer Science*. Springer Nature, Berlin, Heidelberg, 2003.

[22] Kubernetes. Understand kubernetes. https://kubernetes.io/docs/home/, 2022.

[23] Citus Data Microsoft. Microsoft acquires citus data. *TechCrunch*, 2019.

[24] Citus Data Microsoft. Query processing using citus extension for postgresql. https://docs.citusdata.com/en/v11.3/develop/reference_processing.html, 2023.

[25] Citus Data Microsoft. What is citus data. https://docs.citusdata.com/en/v11.3/get_started/what_is_citus.html, 2023.

[26] Jan Kristof Nidzwetzki and Ralf Hartmut Güting. Distributed secondo: an extensible and scalable database management system. *Distributed and parallel databases : an international journal*, 35(3-4):197–248, 2017.

[27] Google Cloud Platform. General-purpose machine family for compute engine. https://cloud.google.com/compute/docs/general-purpose-machines, 2023.

[28] Google Cloud Platform. Python client for kubernetes engine api. https://cloud.google.com/python/docs/reference/container/latest, 2023.

[29] Google Cloud Platform. Regions and zones. https://cloud.google.com/compute/docs/regions-zones, 2023.

[30] Giulia Rovinelli, Stan Matwin, Fabio Pranovi, Elisabetta Russo, Claudio Silvestri, Marta Simeoni, and Alessandra Raffaetà. Multiple aspect trajectories: A case study on fishing vessels in the northern adriatic sea. In *CEUR Workshop Proceedings*, volume 2841, 2021.

[31] Mahmoud SAKR and Esteban ZIMÁNYI. Mobilitydb workshop. https://docs.mobilitydb.com/MobilityDB-workshop/master/mobilitydb-workshop.pdf, 2022.

[32] Mahmoud Sakr, Esteban Zimányi, Alejandro Vaisman, and Mohamed Bakli. User-centered road network traffic analysis with mobilitydb. *Transactions in GIS*, 27(2):323–346, 2023.

[33] Amanpreet Kaur Sandhu. Big data with cloud computing: Discussions and challenges. *Big Data Mining and Analytics*, 5(1):32–40, 2022.

[34] Maxime Schoemans, Mahmoud Attia Sakr, and Esteban Zimányi. Implementing rigid temporal geometries in moving object databases. *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 2547–2558, 2021.

[35] Naresh Kumar Sehgal. *Cloud computing with security and scalability : concepts and practices*. Springer Nature Switzerland AG, Cham, Switzerland, third edition. edition, 2023.

[36] Bharti Sharma, Poonam Bansal, Mohak Chugh, Adisakshya Chauhan, Prateek Anand, Qiaozhi Hua, and Achin Jain. Benchmarking geospatial database on kubernetes cluster. *EURASIP journal on advances in signal processing*, 2021(1):1–29, 2021.

[37] Dimitrios Tsesmelis. Self-scalable moving object databases on the cloud: Mobilitydb and azure. https://github.com/MobilityDB/MobilityDB-Azure, 2021.

[38] Alejandro Vaisman and Esteban Zimanyi. Mobility data warehouses. *ISPRS International Journal of Geo-Information*, 8:170, 04 2019.

[39] Deepak. Vohra. *Kubernetes Microservices with Docker*. The expert's voice in open source. Apress, Berkeley, CA, 1st ed. 2016. edition, 2016.

[40] Yunqin Zhong, Jizhong Han, Tieying Zhang, and Jinyun Fang. A distributed geospatial data storage and processing framework for large-scale webgis. In *2012 20th International Conference on Geoinformatics*, pages 1–7. IEEE, 2012.

[41] Dingju Zhu. Cloud parallel spatial-temporal data model with intelligent parameter adaptation for spatial-temporal big data. *Concurrency and computation*, 30(22):e4497–n/a, 2018.

[42] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. Mobilitydb: A mobility database based on postgresql and postgis. *ACM Trans. Database Syst.*, 45(4), dec 2020.

[43] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. MobilityDB: A mobility database based on PostgreSQL and PostGIS. *ACM Trans. Database Syst.*, 45(4), December 2020.

[44] Esteban Zimányi, Mahmoud Sakr, Arthur Lesuisse, and Mohamed Bakli. Mobilitydb: A mainstream moving object database system. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases*, SSTD '19, page 206–209, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Esteban Zimányi. Mobilitydb 1.0 user's manual. `https://docs.mobilitydb.com/MobilityDB/develop/mobilitydb-manual.pdf`, 2021.

[46] Esteban Zimányi. Berlinmod benchmark on mobilitydb. `https://docs.mobilitydb.com/MobilityDB-BerlinMOD/master/mobilitydb-berlinmod.pdf`, 2022.

[47] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. Mobilitydb: A mobility database based on postgresql and postgis. *ACM transactions on database systems*, 45(4):1–42, 2020.

[48] M. Tamer. Özsu. *Principles of Distributed Database Systems*. Springer International Publishing, Cham, 4th ed. 2020. edition, 2020.

[49] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer International Publishing AG, Cham, 4th ed. 2020 edition, 2019.