CentraleSupélec

université PARIS-SACLAY

Big Data Management and Analytics

Master Thesis

Dimitrios TSESMELIS

Self-scalable Moving Object Databases on the Cloud: MobilityDB and Azure

---

Prepared at Université Libre de Bruxelles

Defended on September 2021

Advisor1:	Esteban ZIMÁNYI	-	Université Libre de Bruxelles	esteban.zimanyi@ulb.be
Advisor2:	Mahmoud SAKR	-	Université Libre de Bruxelles	mahmoud.sakr@ulb.be
Tutor:	Nacéra SEGHOUANI	-	CentraleSupelec	nacera.seghouani@centralesupelec.fr

ULB UNIVERSITÉ LIBRE DE BRUXELLES

BDMA

Acknowledgments

In this short section I would like to express my gratitude to everyone that supported me during the beginning of this long journey that started on September 2019 and finishes on September 2021.

First of all, I'm eternally grateful to my supervisors, Prof. Esteban Zimányi and Prof. Mahmoud Sakr, for their support, motivation and guidance on this master's thesis. Without their help, I would not be able to prepare a work of such quality. I would also like to thank Dr. Mohamed Bakli for initiating me into the field of research. Apart from their invaluable help, they gave me the chance to participate in the preparation of two very challenging community conferences, a unique experience that I truly appreciate.

I sincerely appreciate the decision of the consortium of the Big Data Management and Analytics (BDMA) Erasmus Mundus Joint Master Degree for offering me an Erasmus Mundus Scholarship that made my BDMA experience become true. I also appreciate the quality of BDMA program that equipped me with all the required knowledge to launch a great professional career. I would like to specifically thank Mrs. Charlotte Meurice, member of the International Welcome Desk of ULB, for her continuous support in any procedural issue that occurred during the last two years.

Thanks to Mr. Marco Slot, principal software engineer at Microsoft, for providing very useful insights of Citus internals that played a major role in the final version of this work.

Last but not least, I would like to express my gratitude to my parents Stavros and Alexandra for their day-to-day love, encourage and help that generously give me since the beginning of my life. Both of them have been decisively contributing to the development of my character and my skills. Thanks to my sister Katerina for her real help during my stay in Paris for the second year of this program and for her love all these years.

Dimitrios Tsesmelis

Abstract: PostgreSQL is one of the most promising and quickly evolving relational database management systems. Being a fully extensible system, there are plenty of projects that build functionalities on top of PostgreSQL. MobilityDB is such a tool that enables users to efficiently store, manage and query *moving object data*, such as data produced by fleets of vehicles.

Living in the era of big data technologies and cloud computing, it is vital for cutting-edge database management systems to provide *cloud native solutions* that allow data processing at scale. Citus is such a tool that transforms any PostgreSQL server into a *distributed database*, without preventing any PostgreSQL native functionality.

Deploying a MobilityDB cluster using Citus on the cloud is rather a simple task. It requires deep knowledge of handling the provided infrastructure, configuring the network between the machines as well as time and effort to learn and manage the peculiarities of each cloud provider. In addition, maintaining a scalable system requires continuous monitoring of several factors and metrics that depict the performance of the system. Such a task implies repeating human effort that is sometime prone to errors.

In this work, we aim to target the aforementioned challenges by introducing automation to the rolling out process of a MobilityDB cluster on Microsoft Azure as well as to partially automate the management and maintenance of the deployed solution. Moreover, we provide a tool, called *autoscaler*, that is capable of automatically monitoring the database cluster, analyzing the collected performance metrics and making decisions that adapt the size of the cluster, according to the measured workload.

To assess the performance of our solution, we perform several experiments that combine two different benchmarks, namely BerlinMOD and Scalar benchmarks. The first provides a synthetic dataset that simulates the behavior of moving vehicles across Berlin, while the second one is used to simulate a number of concurrent user requests that query the system under test.

Keywords: Distributed cloud databases, Autoscaling, Database-as-a-Service, MobilityDB, Microsoft Azure

Contents

Chapter 1

Introduction

1.1 Traditional Database Management Systems and Beyond

It is undeniable that our generation lives in the era of big data where new technologies arise to solve critical technological issues. Organisations realize that new sources of unstructured data are precious and they can be exploited to create a complete model of the environment in which the organisation makes business. Having built an accurate view of the business world, the organisation is ready to make better decisions.

Managing and analyzing big data requires respective tools and technologies that enable users to efficiently manage the data. Traditional relational databases, based on SQL, seem not to be able to handle this new type of data, as the supported database schema is mainly static. This means that changing the schema of a SQL database, implies altering one or more tables, which in some cases may not be feasible, especially when tables have billions of rows. On the other hand, NoSQL databases offer dynamic schemas, ideal for storing and managing continuously changing, unstructured data. Such technologies introduce flexibility and can be used to explore the different sources of information as well as the capabilities that such data combination offers.

1.1.1 The Need for Relational Databases and PostgreSQL

The continuously increasing evolution of NoSQL databases does not seem to set bounds to the development of SQL databases. DB-ENGINES [IT21] is a knowledge base of relational and NoSQL database management systems (DBMS) that ranks the different SQL and NoSQL databases according to their popularity. Figure 1.1 depicts the top ten most popular DBMS in June 2021.

Rank			DBMS	Database Model	Score		
Jun 2021	May 2021	Jun 2020			Jun 2021	May 2021	Jun 2020
1.	1.	1.	Oracle	Relational, Multi-model	1270.94	+1.00	-72.65
2.	2.	2.	MySQL	Relational, Multi-model	1227.86	-8.52	-50.03
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	991.07	-1.59	-76.24
4.	4.	4.	PostgreSQL	Relational, Multi-model	568.51	+9.26	+45.53
5.	5.	5.	MongoDB	Document, Multi-model	488.22	+7.20	+51.14
6.	6.	6.	IBM Db2	Relational, Multi-model	167.03	+0.37	+5.23
7.	7.	↑ 8.	Redis	Key-value, Multi-model	165.25	+3.08	+19.61
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model	154.71	-0.65	+5.02
9.	9.	9.	SQLite	Relational	130.54	+3.84	+5.72
10.	10.	↑ 11.	Microsoft Access	Relational	114.94	-0.46	-2.24

Figure 1.1: DBMS ranking of top ten DBMS in June 2021[1]

As we see, the first four positions are occupied by relational DBMS and seven of the ten most popular DBMS are also relational. These numbers prove that the end of relational DBMS is quite far, if not extinct! Why such a statement is true? Actually SQL databases have been developed and optimized for many decades, which makes them very powerful in different scenarios. **SQL databases are the best known way to model, store and query data, the schema of which is known in advance**. This means that if we know exactly what our data represents, then SQL solutions are probably the best alternative, in terms of performance. This happens because SQL technologies provide indexes, table partitioning and many more mechanics that can significantly increase the performance of querying the stored data.

Observing the four biggest players in Figure 1.1, we see that only PostgreSQL [Grob] is an **open source** relational database. More specifically, PostgreSQL is an open source, object-relational database with over 30 years of active development. Figure 1.2 compares the evolution of the top four most popular DBMS from 2013 to 2021. It is clear from the line chart that despite the fact that Oracle, MySQL and MS SQL Server are less popular than before, PostgreSQL gains more and more market share, as its popularity is steadily growing since 2013. Thanks to the fact that PostgreSQL is 100% open source and up to date with the technological changes, it attracts more and more people to build their relational database servers on it.

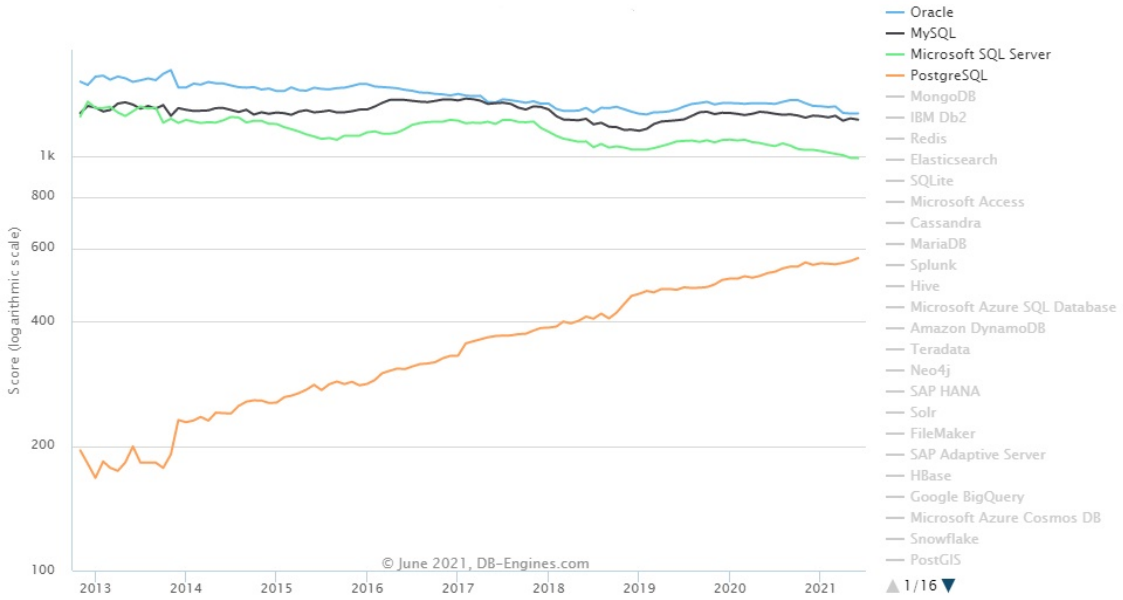


Figure 1.2: Evolution of the top four DBMS from 2013 to 2021[2]

As PostgreSQL is an emerging and growing DBMS, it cannot fall behind the trends of the decade. Cloud computing is a flourishing field that more and more organisations use to reduce cost and add flexibility to their processes. Being a versatile and flexible DBMS, PostgreSQL lives inside the world of cloud native. There are several cloud providers,

[1]DB-ENGINES Ranking https://db-engines.com/en/ranking
[2]DB-Engines Ranking - Trend Popularity https://db-engines.com/en/ranking_trend

with the biggest players being Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform (GCP) that offer a cloud native PostgreSQL distribution in the form of Software-as-a-Service. AWS Relational Database Service [AWSa], MS Azure Single and Flexible Server [Micb] and Google Cloud SQL [Goo21] are the PostgreSQL-as-a-Service versions provided by the most popular cloud providers until now. In addition to them, Azure also offers Hyperscale (Citus) [Hyp21], which is a distributed and scalable version of PostgreSQL on Azure. Although all these services implement **high availability** via replication, **automatic backups** and **several other cloud native features**, they impose several constraints on the users. For instance, none of these services give `SUPERUSER` rights to the service consumers, but only some limited rights that allow creation of new database users, databases etcetera. Another important constraint is the limited set of available PostgreSQL extensions, as none of the cloud providers allows to install every available extension on the database. Section 2.1.1 describes more features and limitations of the aforementioned cloud services.

Therefore, we understand that SQL databases are still vital for several data management scenarios and PostgreSQL DBMS is an emerging relational database. In the next section, we discuss some limitations of PostgreSQL as well as relevant works that extend its functionality.

1.1.2 Self-scalable PostgreSQL Database Solutions

In the previous section we introduced one of the most promising and quickly developing relational DBMS, namely PostgreSQL. Although such a database is very useful when it comes to manage traditional data that can be modeled and **easily queried** with the basic data types that DBMS defines, namely `integer, float, boolean` etcetera, it is a real challenge to use them for moving object data, such as GPS traces. Typical examples of moving data are trips of airplanes, trains and cars. Thinking of the nature of this data, someone can state that it includes two additional dimensions that are not supported by the standard PostgreSQL types, namely **space** and **time**. Indeed, using the known types to model the trajectory of an airplane that makes a trip between two countries, is probably not possible. MobilityDB [ZSLB19], which is an extension of PostgreSQL, adds support for *temporal* and *spatio-temporal* objects, which makes possible and easy for PostgreSQL users to manage moving objects.

A traditional relational database is typically a software designed to **operate on a single server**. No matter if the dataset amounts to 10GB or 1000GB, it is entirely stored and queried using one server. Such an architecture imposes limitations, especially when it comes to cloud computing. When using the cloud, a vital requirement for the application is to be scalable, which means that it can be splitted into several components, and each component can operate on a different server. But as we explained, PostgreSQL by itself is not a scalable application. Citus [Micc] is another PostgreSQL extension that transforms any PostgreSQL server into a distributed DBMS. The core idea behind Citus is table partitioning using sharding, which is basically horizontal partitioning of the target tables. As a result, the different shards of the tables can be independently stored on different servers and hence, PostgreSQL application becomes scalable!

Deploying and managing a cloud application and especially a database cluster, is rather

a simple task. **Scalability** and **high availability** are two features that ideally every scalable application should implement. High availability guarantees that the application is almost always alive and ready to respond to user requests, even when some servers fail. Horizontal scalability is the property of the application to modify the number of its instances that run on independent servers. Although scalability is a quite powerful feature that can significantly increase the performance of the system, it also introduces several challenges. Deciding **when** and **how** the application should scale according to the current traffic and workload, is not a trivial task. A big challenge of this work is to create a **self-scalable** and **highly available** cluster that is able to automatically provision its size according to the observed state of the application.

Consequently, PostgreSQL by it-self is not ideal for every scenario. MobilityDB is an extension that adds support for moving object data and Citus turns PostgreSQL from a single server to a distributed database. The deployment and management of a combination of MobilityDB and Citus cluster is the main scope of this work. In the next section, the main research challenges are presented.

1.2 Research Challenges

This section illustrates the challenges of deploying and managing a PostgreSQL cluster on the cloud that combines both MobilityDB and Citus extensions. A major part of this work is devoted to the automation of the rolling out process of the aforementioned cluster on Azure. MobilityDB and Citus extensions have dependencies on different software that every node of the cluster should have installed in order to host the extensions. Moreover, Citus allows the distribution of a relational database over a cluster of independent nodes, which implies networking configuration that handles the communication between the machines. Furthermore, the target cluster is deployed on MS Azure, which implies that the peculiarities of the specific cloud provider needs to be taken into consideration to efficiently deploy the application on the provided infrastructure. Last but not least, the deployment process of the cluster needs to minimize the interaction with the user. Ideally, the user should be able to complete the process by only initiating it and without interfering while it is in progress, which by definition is not an easy task as many events can cause an abnormal interruption of the process. These are the main points that make the automatic deployment of the cluster a very demanding task.

A number of challenges also arise when building a self-scalable DBMS. With cloud and especially self-scalable databases being an immature field with a lack of previous in-depth research, one major difficulty is the exploration of the factors that affect the performance of a self-scalable database cluster. A wide range of performance metrics is produced and can be extracted from different levels of the application. Using these metrics, we can measure how the system responds to the current workload and acts in such a way that the provisioned resources of the near future will increase the efficiency of the system and reduce the total cost. Hence, defining which performance metrics are ideal and which are mediocre for this purpose as well as how the system should react to the incoming workload is a challenge!

1.3 Objective and Goals

The main objectives of this work are listed in this section. First, we aim to provide a Database-as-a-Service solution on MS Azure, that will enable users to fully exploit the capabilities that MobilityDB and Citus provides. This service introduces **automation**, as the database cluster is deployed on Azure with minimum human intervention. Second, having successfully performed the deployment process, a user can enjoy the experience of a **self-managed deployment** and a **self-scalable DBMS** that automatically adapts its size to the observed workload, in order to achieve the maximum performance and the minimum cost.

Some concrete goals of this work are listed below:

- Explore the existing frameworks and methodologies that enable the management of **elastic** and **scalable** applications on the cloud.
- Model and implement an **automatic** and **configurable** process that initializes a MobilityDB cluster on MS Azure.
- Design and propose a **self-managed** containerized deployment.
- Provide a **highly available** and **self-scalable** MobilityDB cluster on MS Azure.

1.4 Contributions

In this work we provide a Database-as-a-Service tool which is basically a PostgreSQL database cluster that hosts MobilityDB and Citus extensions. The software is meant to be deployed on MS Azure, as it uses several Azure-dependent tools. However, with few modifications, it can be migrated to any other cloud provider, such as AWS and GCP.

The experiments done in this work mainly focus on MobilityDB, as the used dataset includes moving objects. However, the proposed solution **is not bound to any MobilityDB component**, and hence, **it can be used as a distributed PostgreSQL database** for any other purpose.

The contributions done with this software serve fundamentally to the following purposes:

1. **Automation of the Cluster Rolling Out Process**. As we already mentioned, the first goal of this work is to provide a Database-as-a-Service that makes the use of this product easily accessible and used by anyone, regardless of his/her expertise. The set of tools includes the following components:
    - A process that uses *bash scripting* and *Azure Command Line Interface* to automatically deploy a MobilityDB cluster on Azure. The script receives a number of parameters by the users that define the size of the initial cluster. For instance, the user provides the desired number of worker nodes, the size of the virtual machines (VM) as well as some other parameters that are explained in detail later in this work.
    - A Kubernetes deployment that enables the automatic management of the MobilityDB cluster on the cloud. As explained in a future chapter, Kubernetes is an orchestration, production-ready tool that allows the management of containerized applications, deployed via Docker containers.

2. **Implementation of Autoscaling Mechanisms**. This part of the work focuses on the transformation of the static MobilityDB cluster, to a dynamic system that is able to adapt its size to the current state of the environment in which it operates. Specifically, this component is responsible to decide when and how the cluster will grow or shrink, in order to ensure better performance and lower cost. This tool includes:
    - A continuously running process on the background of the master server of the cluster, which is also known as *daemon process*. This tool is implemented using *Python 3* programming language, and its deployment is performed using *virtual environments* to enable quick portability. It basically implements a continuous loop, consisting of four phases, namely monitor, analyze, plan and execute. In the next chapter, we provide more details about relevant works around the field of autoscaling mechanisms, as well as implementation details of our solution.

The rest of this work is structured as follows. Chapter 2 presents the literature review of PostgreSQL solution on the cloud and autoscaling techniques. Chapter 3 introduces the technical ecosystem of this work, where all the used tools and technologies are presented in detail. Chapter 4 details the automated provided solution of deploying and maintaining a MobilityDB cluster on Azure. Chapter 5 lists the available performance metrics that can be used by our autoscalers and presents the implementation steps of the autoscaling mechanisms. Chapter 6 illustrates and compares the experiments and results that were performed in this work. Chapter 7 concludes this work and discusses its limitations as well as future aspects. Finally, Appendix A highlights part of the implementation while Appendix B is a detailed tutorial that guides any user that wants to reproduce the existing implementations and deploy his/her own PostgreSQL cluster on Azure.

Chapter 2

Related Work

This chapter presents a state-of-the-art of the cloud native solutions for PostgreSQL. Moreover, it includes the approaches used to create a system, capable of scaling by itself, also known as *autoscaling system*. We start with an intuitive definition of scalable systems. Next, we formalize the process of an autoscaler, by splitting it into four phases, namely monitor, analyze, plan and execute. Finally, we present a review of the different autoscaling techniques that have been proposed over the last years. This review includes traditional threshold-based techniques, time series analysis, reinforcement learning and queuing theory.

2.1 Cloud Native PostgreSQL

This section describes the different PostgreSQL services offered by the most popular cloud providers, namely MS Azure, AWS and GCP. Furthermore, it presents several available tools and operators that enable users to easily deploy PostgreSQL on Kubernetes. The PostgreSQL operators for Kubernetes are presented at an abstract level, as a detailed description of Kubernetes tool follows in the next sections.

2.1.1 PostgreSQL-as-a-Service

As we already mentioned in the introduction of this work, the cloud providers promote production-ready solutions of PostgreSQL on the cloud. The fact that such implementations are in the form of a service, simplifies the deployment and management of the DBMS for the clients but it introduces several limitations. Table 2.1 summarizes the features offered by Azure Single Server (AZ SS), Azure Flexible Server (AZ FS), AWS RDS and GCP Cloud SQL (G CSQL).

As we understand, the different cloud providers seem to offer similar products. AWS RDS has a significant advantage when it comes to major version updates, as this process is completely handled and executed by the cloud provider. Amazon is the only one that provides such a feature. Almost every cloud provider gives access to the database logs in a way. Azure and Amazon allow the users to directly access the logs on an hourly basis, while Google provides logs analytics via its cloud platform.

On the other hand, we see that the services set barriers when it comes to full management of the database. First, none of them provides `SUPEUSER` access rights, as this privilege is reserved for maintenance by the cloud vendors. Second, management of PostgreSQL extension is partially supported, which means that there are PostgreSQL extensions that cannot be installed on the database server. This is a significant limitation that users needs to consider before using one of these services. Finally, none of the provided services offers data distribution and autoscaling features, which are the main goal of this work.

	AZ SS	AZ FS	AWS RDS	G CSQL
Synchronous standby	Yes	Yes	Yes	Yes
Automatic failover	Yes	Yes	Yes	Yes
Read replicas	Yes	-	Yes	Yes
Scheduled maintenance	-	Yes	Yes	Yes
In-place major version updates	-	-	Yes	-
Support of extensions	Partial	Partial	Partial	Partial
Direct log access	Yes	-	Yes	-
Log analytics	Yes	Yes	-	Partial
`SUPERUSER` access	-	-	-	-
pseudo-`SUPERUSER` access	-	-	Yes	Yes
Data distribution	-	-	-	-
Autoscaling	-	-	-	-

Table 2.1: Comparison of the different available PostgreSQL-as-a-Service provided by the most popular cloud providers

In conclusion, PostgreSQL cloud native solutions seem to be quite mature and production-ready, as the cloud vendors have already implemented several critical features. Amazon RDS is probably the most popular and mature product thanks to the fact that its update cycle is the most sensible and predictable.

In this subsection we described the most popular PostgreSQL-as-a-Service solutions that cloud providers offer. All these services come with a cost, higher compared to a PostgreSQL server deployed on top of an Infrastructure-as-a-Service solution. The next subsection presents some of the most widely used and open source PostgreSQL operators for Kubernetes that can be deployed on any infrastructure provided by the cloud vendors.

2.1.2 PostgreSQL Operators in Kubernetes

This subsection outlines and compares the most popular PostgreSQL operators for Kubernetes. In a few words, Kubernetes is an orchestration tools designed to automate several tasks that are related to management and autohealing processes of containerized applications. Containerized applications are mainly used to speed up the deployment process of the application, as it is totally decoupled from the operation system of the hosting machine. More details about these technologies are presented in Sections 3.4 and 3.5.

All these operators aim to make **cloud native PostgreSQL easily available to anyone**. Deploying a PostgreSQL cluster on the cloud from scratch is a demanding task that requires expertise on several database-related domains. Providing a tool that allows quick deployment and automatic management of a PostgreSQL server on the cloud, is very important feature for non expert users. Furthermore, the PostgreSQL operators provide **high availability via replication**. The architectures of the different solutions propose the existence of one master and several replica nodes, where the master is responsible to serve the users while the replicas are ready to replace the master node, whenever it fails. In

addition, many of the operators offer **automatic backups** that are periodically scheduled by the system and **rolling updates for PostgreSQL major/minor version updates**. Following, we present some of the most well known and production-ready PostgreSQL operators for Kubernetes.

EnterpriseDB Cloud Native PostgreSQL

EnterpriseDB (EDB) Cloud Native PostgreSQL [Ent21] is a PostgreSQL operator that provides a Kubernetes deployment, offering a number of cloud native features. According to the documentation, EDB provides high availability and autohealing via automated recreation of the failed replicas. Second, the product is scalable as it allows easy scale in and out operations and continuous backups along with point-in-time recovery. EDB uses TLS connections and client certificate authentication to ensure the safe communication between the nodes and support for automated PostgreSQL minor version updates. Another advantage of EDB solution is Cloud Native PostgreSQL plugin for *kubectl* that basically forms a user-friendly API of the operator. EDB is planning to implement a Physical Replica Cluster [WB21], which is basically a disaster recovery solution for the cloud that replicates the whole PostgreSQL cluster over different physical regions, to ensure high availability even if the infrastructure of the cloud vendor of a region experiences a physical disaster.

Apart from the key feature that EDB Cloud Native PostgreSQL provides, it also includes some limitations. One major limitation is that EDB does not allow to provide a custom Docker image of PostgreSQL. This means that if users want to install PostgreSQL extensions, they need to manually perform the installations on every single server of the cluster. In addition, EDB does not actually implement data distribution, but data replication. In other words, the system does not scale when more nodes are added, as the query answering is done using a single server. This is a major disadvantage as we have to increase the cost of the infrastructure only to ensure high availability, but without any actual computational gain.

Crunchy PostgreSQL Operator

Crunchy PostgreSQL Operator [Data] is another operator for Kubernetes provided by Crunchy Data. Figure 2.1 illustrates the architecture of a PostgreSQL cluster deployed via Crunchy Data solution. Similar to EDB Cloud Native PostgreSQL, it offers high availability via replication. However, in the case of Crunchy Data, this feature is supported by a distributed consensus based high-availability solution [Datc], which seems to be a more advanced and sophisticated high availability algorithm. The product also offers backups with different strategies [Datb], namely full, differential and incremental backups. Finally, it seems that the tool is more customizable compared to EDB as it allows use of custom Docker images of PostgreSQL and Custom Resources Definition (CRD) for Kubernetes objects. Similar to EDB, Crunchy PostgreSQL Operator does not provide data distribution, but data replication.

[1] Figure taken from https://access.crunchydata.com/documentation/postgres-operator/latest/?_gl=1*1gmxdd4*_ga*MTU1ODE5MDE0MS4xNjIzOTIwNjAx*_ga_ZQQNV12TZ5*MTYyNDk1NDg0MS4zLjAuMTYyNDk1NDg0MS4w

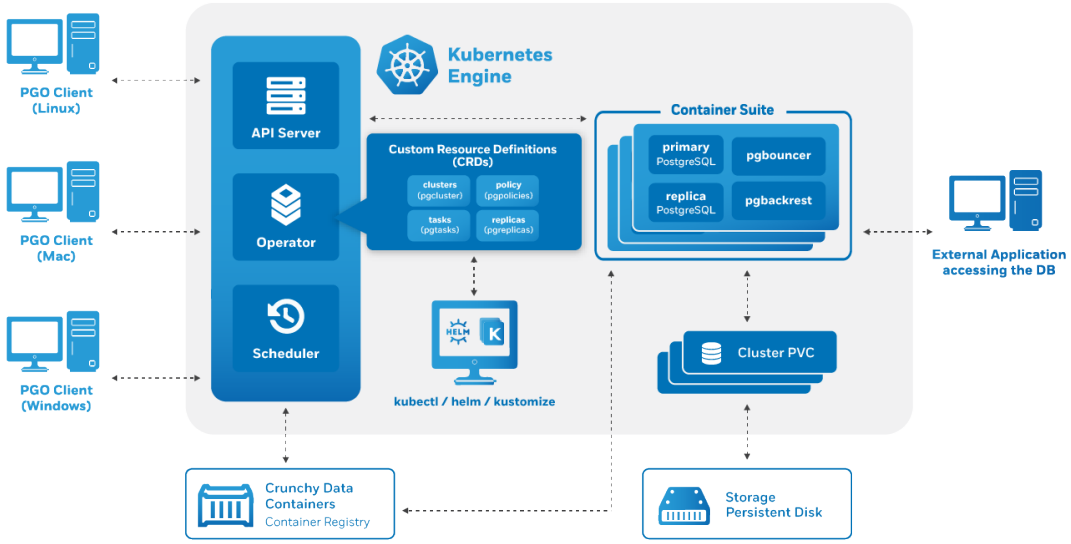


Figure 2.1: Crunchy PostgreSQL operator architecture[1]

StackGres Operator

StackGres operator [OnG21] is the newest PostgreSQL operator for Kubernetes. StackGres has been designed to make PostgreSQL available to everyone [Her21]. Indeed, this operator enables the deployment of PostgreSQL on the cloud in just a few steps. Unlike the cloud native PostgreSQL solution of the cloud providers described in Section 2.1, StackGres gives full access rights to the users of the tool, putting them in full control. It also guarantees high availability with Patroni software [Zal21] as well as automated backups. Finally, StackGres comes with an advanced fully-featured management console, that includes both a bash console and a graphical interface. Using the latter, users can get valuable information from the built-in Grafana dashboards, that is integrated into the web console. The architecture of the StackGres operator is depicted in Figure 2.2. The lack of data distribution is a major disadvantage, common for all the three operators.

Comparing the Different PostgreSQL Operators

As we understand, there have been proposed different operators for deploying PostgreSQL on Kubernetes. Table 2.2 summarizes the different available features that each operator includes. It is true that creating a PostgreSQL operator is not a trivial task. There are different available tools that allow the deployment of a PostgreSQL cluster, and the choice of a system depends on the requirements of each scenario as not all the operators are ideal for each use case.

Thinking of the goals of this work, we mainly want to achieve the following: 1) Automate the deployment and management of a MobilityDB cluster with Citus extension on Azure and 2) Create an autoscaling mechanism that automatically adapts the size of the

[2]Figure taken from https://stackgres.io/doc/latest/01-introduction/03-architecture/

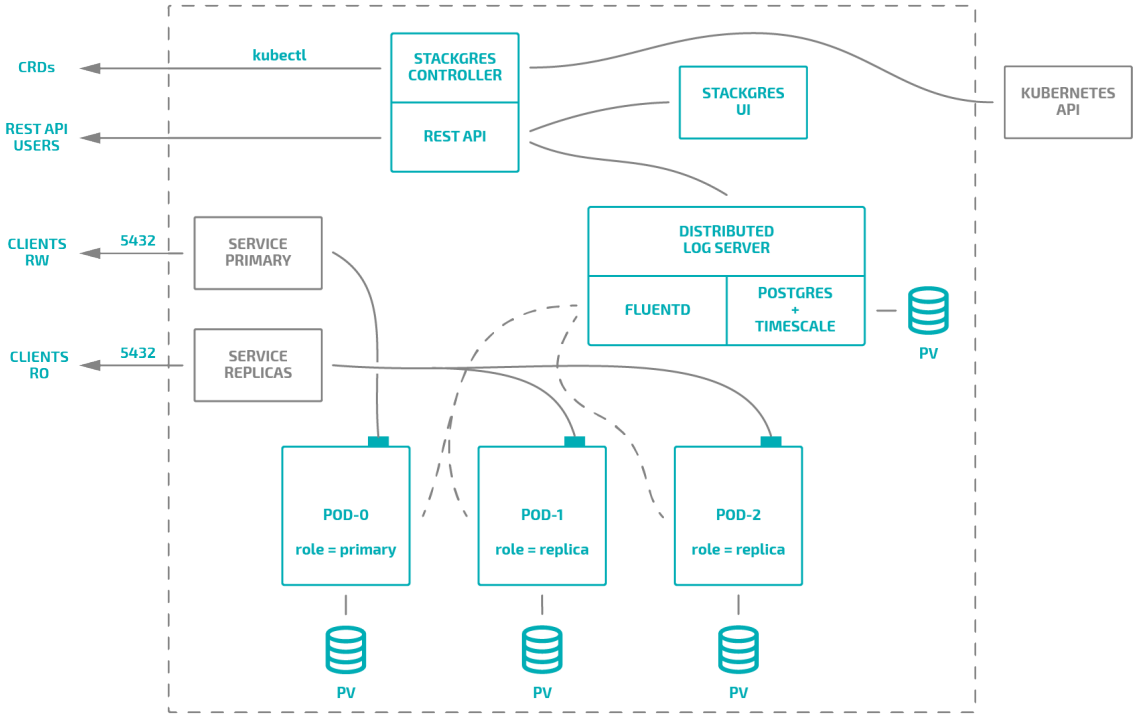


Figure 2.2: StackGres PostgreSQL operator architecture[2]

cluster. Regarding the first goal, some of the existing operators can partially support its implementation. For example, Crunchy operator can be used to deploy a custom Docker image of PostgreSQL with MobilityDB and Citus extensions installed, but the other two tools cannot, as they do not currently support custom images. However, there is a major issue that is closely related to the architecture of these solutions and the first goal of this work. We explained that none of them applies data distribution, but they implement data replication, meaning that **each node of the cluster is meant to store exactly the same data**. This idea is completely opposite to Citus architecture that distributes the data over the nodes using sharding. This means that **each individual node is designed to store different partitions of the data**, which seems that cannot be supported by the current operators.

Concerning the second goal of this work, none of the aforementioned tools provides any autoscaling feature. Most of them provide a way to scale in and out the cluster, but this process needs to be done manually by the user. Consequently, these two last observations **highlight the need for a new tool that is able to cover the goals of this work**. Some components of this new tool, for instance containerization and orchestration with Kubernetes, remain the same, but another implementation is needed to cover the aforementioned gaps.

In this section, we covered a state-of-the-art of the existing cloud services, tools and operators that allow the quick deployment of PostgreSQL on the cloud. The next section introduces the meaning of autoscaling systems and provides a state-of-the-art of the different available autoscaling techniques in a wide range of applications.

	EDB	Crunchy	StackGres
High availability with streaming replication	Yes	Yes	Yes
High availability with synchronous replication	Yes	Yes	Yes
Automated backups	Yes	Yes	Yes
`SUPERUSER` access rights	Yes	Yes	Yes
Command line interface	Yes	Yes	Yes
Advanced web console with monitoring capabilities	-	-	Yes
Custom Docker images	-	Yes	-
Data distribution	-	-	-
Autoscaling	-	-	-

Table 2.2: Features comparison of the different available PostgreSQL operators

2.2 Autoscaling Definition

In this section, we describe the concepts of *scalable systems* and *autoscaling mechanisms*. We provide both strict definitions of the terms as well as intuitive examples to better understand the hidden capabilities that such systems offer.

Nowadays, autoscaling is a widely used term when it comes to cloud computing technologies. Before defining this term, we first need to clearly state the significance of scaling an application. The following example illustrates the meaning of scaling with a realistic scenario. Assuming a newly created startup Company A that designs, develops and releases a specific web application to the public. During the very first days of releasing, such a company would expect relatively low traffic to its web servers, as the application is not popular in the beginning. However, after some months, Company A achieves, by adopting a good strategic marketing and making appealing offers, to attract more and more users and its sales skyrocket. At this point, Company A faces a big challenge of provisioning the right hardware resources to guarantee **low response time** and **continuous availability** to its users. To tackle this issue, Company A needs to own a scalable application, that is able to either ***scale up*** or ***scale out*** (Figure 2.3). In other words, Company A can apply vertical scaling by adding more resources (e.g. more RAM and CPUs) to the machine(s) that already serve(s) the application or by applying horizontal scaling, meaning that more identical machines will be deployed and the whole cluster of machines will serve the application. Horizontal scalability requires that the application is designed to be distributed on different machines, otherwise it cannot be applied.

In the previous scenario, we saw that Company A has two alternatives to make its product available to more users. However, in the case that Company A has its own **on-premises** servers, scaling out the application can take months, as the company has to first decide the amount of the resources that it needs, physically acquire these resources

[3]Figure taken from https://insights.illumio.com/post/102foor/scaling-up-vs-scaling-out-your-security-segmentation

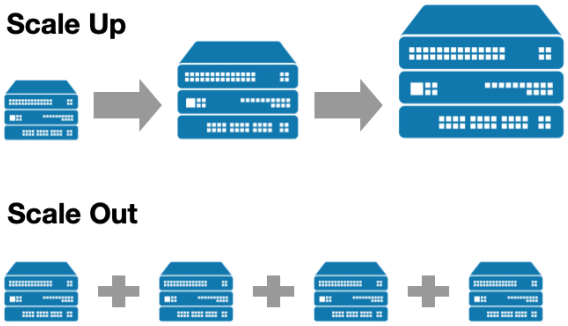


Figure 2.3: Scale Up vs Scale Out[3]

by buying them and installing them on its premises etcetera. During those months, it is highly likely that this company will lose many of its clients, as the quality of service of its application will not be the expected one. In addition, Company B is another case where scaling the on-premises resources is not possible. Company B owns an e-shop that has very high demand during specific periods (e.g. during summer holidays and during December) and the available resources cannot handle the traffic. In such scenario, the previously described process of scaling is not affordable, as Company B would invest in buying resources that would not **fully utilize** during the whole year. Hence, there is an emerging need for a scalable system that would be able to scale out when there is an increasing demand of services and to scale in when the demand decreases.

Cloud computing effectively addresses the aforementioned problems, by enabling the users to dynamically acquire and release resources **in a few minutes**. In this way, companies that host their applications on the cloud are able to build *elastic* applications, that can be adapted to the current workload. Although scalable applications have many benefits, they also add extra human effort, as people need to decide when the scaling operation should be performed. **A system that can achieve the right scaling with minimum human intervention or, even better, without it, would be called an *autoscaling* system and it is the main goal of this work**.

In this section, we explained the concepts of scalable system and autoscaling mechanisms. In the next section, we provide a state-of-the-art of the different autoscaling mechanisms that have been proposed over the last years for a wide range of applications.

2.3 A Review of Autoscaling Techniques

This section includes a state-of-the-art of the autoscaling techniques that have been applied and tested on different applications. First, we formalize the process that an autoscaler performs. Then, we classify the available autoscaling techniques, according to the core idea that is behind the algorithm.

The main goal of an autoscaling process is to adapt the allocated resources that are attached to an elastic application, with respect to the current workload. Another factor that was not taken into account in the previous section, is related to the cost of the allocated resources. As we already discussed, the more cloud resources the application

uses from a cloud vendor, the higher the cost will be. Ideally, a scalable application would always allocate more resources than required to ensure the Service Level Agreement (SLA) with the users but such an assumption is not realistic as everything comes with a cost. For each scalable application, we would like to design an autoscaler, the goal of which is to provision the desired resources with the least cost. Defining such a mechanisms hides several pitfalls:

- Under-provisioning: In this case, the autoscaler has not allocated enough resources to efficiently manage the incoming requests, which leads to a poor user experience. When the autoscaler detects that the provisioning is not correct, the system tries to bind more resources, but the allocation process in cloud environments is not instantaneous. Some minutes are needed before the new machines are ready to be used, but during this time the application is not responsive for some users or it does not meet the SLA.
- Over-provisioning: In this case, the user experience is the desired one, as there are more than enough machines to serve the clients. On the other hand, extra money are spent, duo to unused resources.
- Oscillation: Oscillation occurs when both previous cases happen at the same time. In other words, the autoscaler is not able to correctly provision the workload, as in some cases it underestimates the incoming requests and in some other cases, it overestimates them.

All the above cases are challenges for the creation of autoscaler. A performant autoscaler minimizes the existence of such behaviors.

2.3.1 Formalizing the Autoscaling Process

As we clearly understand, the process of defining an ideal autoscaler is not trivial and a lot of research has been done on this field the last two decades. Lorido-Botrán et al. [LBMAL14] review the available autoscaling techniques for elastic cloud applications. Firstly, the autoscaling process is related to the MAPE loop of autonomous systems [MBEB11, RO04]. MAPE loop consists of the following four phases and it illustrates the basic operations that each autoscaler performs in order to correctly adapt the resources of the system:

1. **Monitor** is the first phase of the MAPE loop. During this step, the autoscaler collects several metrics that sketch the current state of the application. In order to be able to predict the next task to perform, the autoscaler needs to have updated data of high quality. This data is called *performance metrics* and they can be metrics like CPU utilization, disk and network access, memory usage etcetera. Ghanbari et al. [GSLI11] provide an extensive list of *performance metrics* that are listed in Table 2.3. Most of these metrics are not generated by the autoscaler but from other monitoring systems that can be provided by the cloud provider, by the operating system or by the running application. Casalicchio et Perciballi [CP17] split the different available *performance metrics* into two groups, namely *relative* and *absolute metrics*. *Relative metrics* refer to the resources that are allocated to a specific Docker container. For example, in a Docker set up where one node hosts two containers, the maximum CPU utilization of each container can be at most 50%. On the contrary, *absolute*

metrics measures the actual state of the infrastructure.

2. **Analyze** is the next step of the MAPE loop. After the *performance metrics* have been gathered, the autoscaler has to analyze them before defining the provisioning strategy. The autoscalers can be classified into two groups, namely *reactive* and *proactive* autoscalers. A *reactive* autoscaler reacts to the current system status by only taking into consideration its current state. On the other hand, *proactive* autoscalers are more sophisticated mechanisms that predict part of the future state of the system in order to better respond to the current status. Selecting which of these two types of autoscalers will be used, depends on the type of the application as reactive autoscalers are not a good choice for applications that have fluctuating demands. However, if the application usually shows long-term high demand of requests, then reactivity is probably a better choice.
3. **Plan** is the phase of the MAPE loop that defines how the autoscaler will adapt the system to the incoming workload. If the application is designed to scale in/out, the planning phase defines how many new machines are going to be removed/added to the current cluster in order to efficiently respond to the future workload. During this phase, the autoscaler considers the trade-off between the user experience and the cost of the allocated resources.
4. **Execute** is the last phase of the MAPE loop. The actions that were decided in the previous step are actually executed via the cloud provider's API. As we mentioned before, it always takes some time (minutes) before these actions are actually reflected on the system as there is a delay between the time that a user requests the allocation of a new VM with the required software installed, and the time that the VM is actually available to the user.

Source	Metrics
Hardware	CPU utilization, disk access, network interface access, memory usage
General OS Process	cpu-time, pagefaults, real-memory (resident set) size
Load balancer	request queue length, session rate, number of current sessions, transmitted bytes, num of denied requests, num of errors
Web server	transmitted bytes and requests, number of connections in specific states (e.g. closing, sending, waiting, starting, ...)
Application server	total threads count, active threads count, used memory, session count
Database server	number of active threads, number of transactions in (write, commit, rollback, ...) state
Message Queue	average number of jobs in the queue, average job's queuing time

Table 2.3: List of performance metrics for defining autoscaling rules

2.3.2 A Classification of Autoscaling Techniques

Although we have already made a first classification of the different autoscaling techniques based on how reactive the selected strategy is, a more focused categorization is needed as the problem of autoscaling cloud applications is complex and needs multifaceted solutions. In addition, such a classification would not be enough as there are cases where it is not clear whether the autoscaler is reactive or proactive. Both previous works, [LBMAL14, RS21], classify the different available autoscaling techniques in the following categories:

1. Threshold-based rules
2. Time series analysis
3. Reinforcement learning
4. Queuing theory

Such a classification seems useful to compare the different approaches proposed in the literature. However, the predefined problem has many aspects and hence, each solution targets to partially optimize them. For instance, one greedy autoscaler may focus on optimizing the decisions that are taken for the current state, without considering possible states of the future while another autoscaler predicts future states of the system in order to minimize the overall cost. With the following grouping, we will be able to identify common elements in the proposed solutions as well as interesting insights on how the different cloud providers address the autoscaling challenge.

2.3.2.1 Threshold-based Rules

Autoscalers that are based on threshold rules are the most popular category of autoscalers, especially when it comes to autoscaling services provided by the cloud providers. Indeed, Amazon Web Services, MS Azure, Google Cloud Platform and many other cloud providers mainly propose autoscaling mechanisms that define quite simple rules and policies that are appealing to cloud clients. This kind of systems can be considered as reactive autoscalers and they are basically used during the planning phase of the MAPE loop, as they define when and how many resources will be allocated. However, the fact that they are very simple makes this kind of autoscalers prone to errors and cannot be used for every type of application. **Defining a good threshold-based autoscaler implies fine tuning of the rule thresholds and extensive experiments**.

One example that can help us understand how this type of autoscaler behaves, is the following rule: *whenever the average CPU utilization (%) is more than 85% for more than 5 minutes, add 3 new VMs* or *whenever the active user transactions of the current time window is 50% less than the active user transactions of the previous time window, reduce the number of VMs to the half*. With these two examples, it is evident that the performance of such an autoscaler is coupled with the fine tuning of the defined thresholds. In other words, the application administrator needs to carefully tune these parameters in order to increase the efficiency of the autoscaler. Dutreilh et al [DMM+10] highlight that careful tuning of the threshold can result in avoiding system oscillations. Another common practice to tackle oscillation is to introduce a *cooldown period* after each execute phase. In this way, the autoscaler waits until the changes are effective to the system before taking the next decision.

Thanks to their simplicity, threshold-based autoscalers are widely used by the major

cloud providers. AWS Auto Scaling [AWSc] is the standard autoscaling service provided by AWS. AWS Auto Scaling allows building scaling plans for resources like Amazon EC2 instances or Amazon DynamoDB by setting target utilization levels. MS Azure Autoscale [Mica] is the equivalent service porivded by MS Azure cloud provider. Similarly, it enables the user to scale MS Azure resources by pre-defining rules in the MS Autoscale settings panel.

The Kubernetes Horizontal Pod Autoscaler [Kubb] is an available autoscaler provided by Kubernetes. Kubernetes is a tool that manages deployment components of scalable applications. One basic difference with the previously described autoscalers is that Horizontal Pod Autoscaler works at the container level and not at the infrastructure level. It supports reactive thresh-hold based rules for CPU utilization metric and it can be extended by using its plugins. As Kubernetes operates closer to the application level, Horizontal Pod Autoscaler should be able to deploy new VMs on the cloud when the existing one are not enough or to deallocate them when they are not necessary anymore. This feature is available through Kubernetes Cluster Autoscaler [Kuba].

Beloglazov and Buyya [BB10] propose a novel approach for dynamic consolidation of VMs based on adaptive utilization thresholds. In this work, the authors illustrate the massive energy consumption of the big cloud data centers and propose a method that is able to limit the consumption of the system as low as 1%. More specifically, Dynamic Utilization Threshold algorithm is proposed, which is based on a statistical analysis of the historical data collected during the lifetime of VMs. Although the CPU utilization cannot be predicted, it is modeled as a random variable and hence, characteristics of the distribution over some recent period of time can be calculated. As some statistically computed values of the future are known, more confident decisions can be taken for the future provisioning. Such an approach, can be considered as less reactive than the previously described one, as the defined policies are dynamically changed according to statistical knowledge of the future state. As stated in [RS21], little research has been done in the field of adaptive thresholds definition.

2.3.2.2 Time Series Analysis

The reactive nature of the threshold-based approaches is one of the major drawbacks which makes them inappropriate for many applications. Time series analysis is a wide field that can introduce proactivity to an autoscaling system. Time series are generally used to represent the change of a measurement over time. Figure 2.4 depicts a typical example of a time series, where the CPU utilization of a system is plotted over the last 60 seconds.

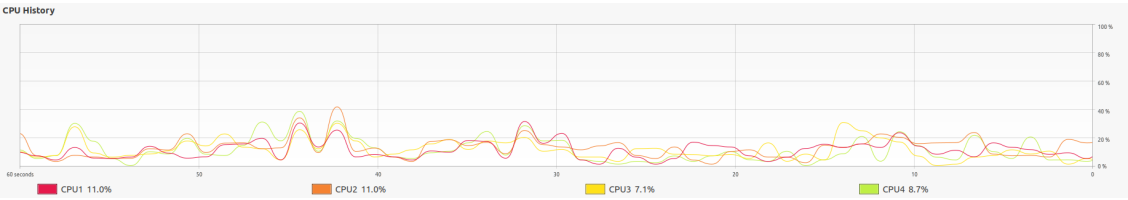


Figure 2.4: An example of time series extracted from the CPU monitoring system of my personal computer

A time series is a sequence of data points listed in time order. Consequently, any of the metrics used by the autoscaling mechanisms can be considered as a time series, as the the VMs are continuously evolving systems that produce data points, measured at successive time instants and spaced at uniform time intervals. As we understand, the creation of the time series occurs during the monitor phase of the MAPE loop. Time series analysis can be performed during the analysis phase in order to predict future values of the monitored metric. By knowing these values, a more sophisticated autoscaling system can be designed that exploits both time series analysis (analysis phase) and threshold-based rules (plan phase) or any other method, to proactively respond to the current state of the system. The remaining part of this subsection is dedicated to the different existing time series analysis methods, varying from simple to more sophisticated one.

- **Moving average methods:** Assuming that the monitor phase produces o observations over some time. In time series analysis, we typically use the last w observations in a time window, where $w \leqslant o$. The desired forecast value x_{t+1} can be computed as a weighted sum of the past w values as $a_1x_t + a_2x_{t-1} + ... + a_qx_{t-q+1}$, where $a_1 + a_2 + ... + a_q = 1$ and each $a_i > 0$. There are many variants of the method to define the a values. Most of the times, more significance (greater weight) is given to the most recent observations. It should be noted that moving average methods are mostly used to remove noise from time series by eliminating potential outliers.
- **Auto-regression methods** Auto-regressive models can be used to forecast future values of a time series, by using a linear combination of past values. An auto-regressive model of order p, also called *AR(p)* can be written as $x_{t+1} = \phi_1x_t + \phi_2x_{t-1} + ... + \phi_px_{t-p+1} + \varepsilon_t$, where p is the size of the time window and ε_t is white noise. There are many different ways in the literature to compute the ϕ weights. For instance, auto-regressive moving average (ARMA) is a known method that combines moving average and auto-regression, ideal for stationary data points, i.e. the mean and the variance of the time series does not fluctuate. Auto-regressive integrated moving average (ARIMA) is an extension of the ARMA model, suitable for non-stationary time series.
- **Machine learning methods:** Machine learning is a flourishing field that is being used the last years to tackle a wide range of problems. This field also includes regression techniques that use past observations to predict future values. In this case, the algorithm tries to approximate a polynomial function that fits to the discrete data observations. Typical examples of this type of algorithms are linear and logistic regression. Another type of algorithms that are also widely used for time series forcasting are neural networks. Shahin et al. [Sha16] use long short-term memory (LSTM) recurrent neural networks to forcast the CPU usage of the VMs. LSTM are ideal for data that includes the meaning of time as their architecture has memory that captures the behaviour of the system over time.

Another big challenge of time series analysis is the storage and management of the collected data. Time series are typically continuous data that are received by one or more source systems, like sensors. These systems produce vast amounts of data at high velocity, which introduces new challenges for the DBMS. Jensen et al. [JPT17] make a survey of the available time series management systems (TSMS) that have been developed to overcome the limitations imposed by the traditional DBMS for time series management. ModelarDB

[JPT18] is a distributed TSMS, developed on top of Apache Cassandra for storing and Apache Spark for processing the data. This tool uses models to store the time series data, by segmenting it into two static models, namely Gorilla and constant function model.

2.3.2.3 Reinforcement Learning

Reinforcement learning (RL) is another set of algorithms that has been used to design autoscaling mechanisms. RL is a type of self-learning algorithms that are able to choose the best performing **action** for a given **state** after some time of training. RL algorithms are mainly repeating algorithms. They include the meaning of the **agent** (the autoscaler) and the **environment** (the application). In every iteration, the agent makes an *action* by observing the current *state* of the environment and it receives a *reward* that determines how good or bad the action was. The objective of the agent is to find the optimal policy π that constitutes a mapping from the *state* to the *action* space, which is based on a Q-value function (Q(s,a), where s stands for state and a for action). To find the optimal policy, the goal is to maximize the **expected long term rewards**.

There have been proposed different algorithms to obtain the Q-value function. Q-learning is a widely used algorithm that creates a lookup table (Q-table) that maps every *state s* to the best *action a*. The core idea of the algorithm is an iterative process that enables the agent to explore the environment and try many actions to find out which series of actions lead to better results. As the agent explores the different alternatives, it continuously updates the Q-table whenever a better choice is made.

One challenge that RL techniques introduce, when it comes to autoscalers, is the modelling of the problem. As we understand, RL implies the definition of the *states*, the *actions* and the *rewards*. Depending on the nature of each application, different models can be defined. Tesauro et al [DMM+10] use the combination of the *total number of requests, the current number of VMs and the number of VMs in the previous time interval* as the *state*. Dutreilh et al [TJDB06] define the *state* as the *total number of requests, the current number of VMs and the average response time of the system*. Regarding the *actions* of an horizontal autoscaler, the available actions are mainly three: add, remove VM(s) and do nothing.

Although RL approaches facilitate the training phase of the algorithm, as they do not require the collection of a training set, as most of the machine learning techniques do, they also introduce several challenges. Specifically, the fact that the autoscaler learns by it-self which actions are the best according to each state, results in poor performance at the early stages. Most of the times, the Q-values of the agent are randomly initialized which means that the autoscaler will receive naive decisions until it starts to be more intelligent. Considering that the actual execution of the autoscaler decisions takes some minutes, we understand that the total training time of the agent can grow a lot, which may be unacceptable for some applications.

2.3.2.4 Queuing Theory

Queuing theory has been used to model and analyze autoscaling systems. In queuing theory, the model places the users in a traditional queue from which several performance metrics can be extracted, namely average response time of the requests, the mean waiting

time in the queue, the length of the queue and many more. These metrics can be used during the analysis phase of the MAPE loop in order to take decisions for the future actions that will be taken.

In this section, we presented a state-of-the-art of the autoscaling techniques. The next chapter provides the technical ecosystem of this work, in terms of background technologies that the reader needs to know in order to be able to fully comprehend the aspects of this work.

Chapter 3

Technical Ecosystem

This chapter introduces the technical background that is needed for the comprehension of this work. First, we motivate the use of cloud computing and we explain the different alternatives that cloud provides. Following, we give the big picture of MobilityDB, which is the database management system used in our experiments as well as Citus, a PostgreSQL extension that introduces scalability. Finally, we demonstrate Docker fundamentals and Kubernetes platform that are used for the deployment and management of cloud applications, as well as DevOps principals.

3.1 Cloud Computing and Microsoft Azure

Nowadays cloud services is a flourishing domain with more and more companies migrating their data from their own data centers to the cloud. There are several cloud providers that are providing different cloud services, with the biggest players being Amazon Web Services, Microsoft Azure, Google Cloud Platform , Alibaba Cloud and IBM Cloud. This work is based on Microsoft Azure cloud vendor but the implemented concepts can be easily migrated to other cloud environments. Although each cloud provider offers a wide range of services, most of them have many elements in common while some of them being exactly the same services, baptized with different names. All these services are designed to provide access to applications, platforms and resources, without the need for on-premises infrastructure or hardware. But why cloud services become more and more popular? There are several advantages that appeal customers to use cloud with some of them being listed below.

3.1.1 Cloud Benefits

This section lists some of the reasons that render cloud services one of the most emerging technologies nowadays. It also groups the different forms of services that cloud offers.

The first benefit of the cloud is that it is **simple to use**. Imagine a company that conceptually creates an innovative product or service but it lacks the infrastructure to host it. By considering the cloud solution, this company is able to allocate the required resources with just a few clicks and immediately start developing its ideas. On the other hand, if that company had followed the traditional way, it should had spent a lot of time in physically acquiring and maintaining the hardware. Hence, **it is clear that cloud dramatically decreases the time-to-market**.

Moreover, cloud services have **lower cost**, when consciously used and there are plenty of reasons that support this statement. Once a company starts using the cloud, it stops caring about paying for software licenses, hardware updates, infrastructure maintenance, upgrades, and all the other expenses that hosting a data center includes, as the cloud

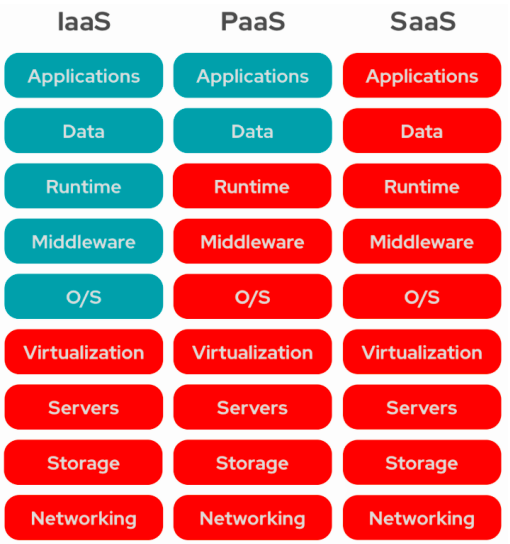
vendor is responsible for them. The company has only to pay the provisioned monthly or annual cost according to the consumed services. Cloud services follows the *pay-as-you-go* model which means that customers pay only for the resources that they are using. We should notice that the customers are responsible for monitoring their expenses, as they can easily skyrocket. Cloud providers can theoretically provide infinite amount of resources.

Scalability is another fundamental advantage of the cloud. It is true that when an organisation is using the computational power offered by the cloud, it can very easily scale the allocated resources according to the current incoming workload. That means that in case of high traffic, more resources can be allocated to serve the clients while in case of low traffic, some resources can be freed to decrease the cost. We should not forget that cloud cost is a linear relation between the number of used resources and the time that they are allocated. With an on-premises data center, such a flexibility would be impossible as the physical resources cannot be instantaneously acquired or released. As a result, **cloud computing introduces extra flexibility to the organisations enabling them to quickly adapt to the changes of the market**.

Disaster recovery is another key asset. Data loss is a major concern for every organisation. When maintaining an on-premises data center, disaster recovery is crucial for the companies, as they should have protocols to recover from potential total disaster of their data center. When using the cloud, it is under the responsibility of the vendors to guarantee that data is always available. Cloud-based solutions provide efficient disaster recovery for every kind of scenarios, varying from physical disasters of the cloud data center to power outage and more.

3.1.2 Types of Cloud Services

There are mainly three types of cloud services, namely Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS). Apart from these three types of cloud services, there are more that fall under one of these categories. For instance, Database-as-a-Service is classified to SaaS as it is a specific application of SaaS. Figure on the right illustrates the differences between the three types. The shapes in blue represent the tasks managed by the cloud provider and the shapes in red the tasks managed by the customers.



- **Infrastructure-as-a-Service (IaaS)** provides the infrastructure on top of which customers can build their own applications. When it comes to IaaS, the cloud vendor provides the virtual machines, the storage, the networking and the rest of the stack that is related to the infrastructure, while the customer manages the operating system, the middleware, the runtime and whatever is related to the application that is running on top of the provided infrastructure.

- **Platform-as-a-Service (PaaS)** serves as a web-based environment where the cloud vendor maintains the infrastructure, the operating system, the database and the customer has only to develop and operate the application.
- **Software-as-a-Service (SaaS)** refers to any type of application that is offered by any company that is hosted on the cloud. This broad category includes many different services, like cloud storage, email, analysis tools etcetera.

In this work, we adopt an IaaS approach, as we allocate the hardware resources by Azure and we develop our application on top of it.

In this section, we illustrated the benefits of cloud services and we divided cloud resources into three categories, namely IaaS, PaaS and SaaS. In the next section, we introduce MobilityDB as a PostgreSQL extension for managing moving objects.

3.2 MobilityDB

This section is used to motivate the use of MobilityDB. MobilityDB [ZSLB19] is a database management system for moving object geospatial trajectories, such as GPS traces. It is freely accessible in GitHub [Mobb] and it comes with detailed documentation that analyzes its functionality and capabilities. It adds support for *temporal* and *spatio-temporal* objects to the PostgreSQL database and its spatial extension PostGIS. The behaviour of a *temporal* object can be described by a temporal datatype. For instance, the fluctuation of the speed of a vehicle can be perfectly expressed using *temporal float* data type. A *spatio-temporal* object can be defined as an object that is moving in a region, over time. Typical examples of *spatio-temporal* objects are cars, ships and airplanes. This type of database management system can be widely used by companies that manage fleet of vehicles. We can consider a company that owns hundreds of ships and it would like to continuously monitoring the position of them to perform further analysis, like fuel consumption reduction, accident prevention and more. Thinking of the tools and capabilities of a traditional database system, storing and managing such data seems to be challenging. MobilityDB is a management system that provides a new set of *temporal* and *spatio-temporal* types and operations that facilitates the manipulation of *spatio-temporal* data.

MobilityDB is a big, well elaborated system with a lot of research being performed around it. Bakli et al. [BSZ19], [BSZ20a], [BSZ20b] explore the capabilities of MobilityDB on distributed environments and with a wide range of external tools. Zimányi et al. [ZSL20] provide more insights on the performance of MobilityDB, by benchmarking it using BerlinMOD benchmark. Schoemans et al. [MSaZ21] introduce an implementation of rigid temporal geometries for MobilityDB and propose efficient algortihms for the operations defined in ISO 19141. MobilityDB is an OSGeo community project, compliant with the Moving Features standards[1] from the Open Geospatial Consortium (OGC).[2] Apart from the scientific papers, MobilityDB team actively participates in several community presentations, with the latest one being PGConf.Russia [ZS21a], a leading Russian PostgreSQL international conference and POSTGRES VISION 2021 [ZS21b], a free virtual global event that brings together the world's leading PostgreSQL experts, users, and community members.

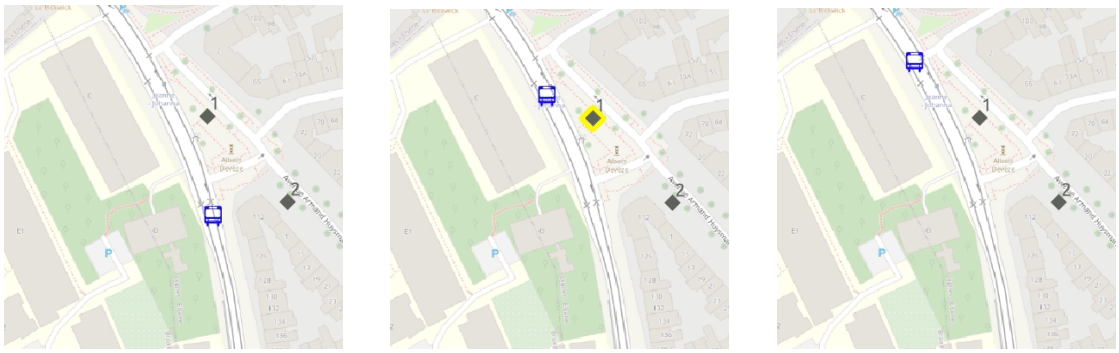
[1] https://www.ogc.org/standards/movingfeatures
[2] https://www.ogc.org/

3.2.1 MobilityDB Data Types

MobilityDB is an extension of PostgreSQL and PostGIS. This means that a user can leverage the benefits of a PostgreSQL environment as well as the ones provided by MobilityDB. This DBMS introduces the following set of data types to assist the manipulation of data representing moving objects: `tbool, tint, tfloat, ttext` that are based on the `bool, int, float`, and `text` base types, respectively. These new types constitute the *temporal* version of the base type. For example, `tint` shows how an integer value evolves over time. Moreover, `tgeompoint`, and `tgeogpoint` are defined which are based on `geometry` and `geography` base types provided by PostGIS. Furthermore, MobilityDB uses four time types to represent extents of time: the `timestamptz` type provided by PostgreSQL and three new types which are `period, timestampset,` and `periodset`. In addition, two range types are defined: `intrange and floatrange`.

3.2.2 Illustration Case

In this subsection, we will try to motivate the use of MobilityDB with a more realistic scenario, which was originally presented by Sakr Mahmoud [Mah]. Our example is related to the visibility of advertising billboards to passing vehicles. Assuming that an advertising company would like to optimize the location of billboards in such a way that billboards with higher cost are placed on streets where more buses pass daily and the duration of the visibility of the billboards to the passengers is high. Figure 3.1 illustrates three continuous captures of a bus passing close to a billboard. Whenever the bus is within 30 meters to the billboard, it is visible to the passengers of the bus and it is highlighted with yellow stroke (Fig. 3.1b) while in the rest of the cases (Figs. 3.1a and 3.1c), the billboard is not visible. We could easily create a database (Listing 1) using PostGIS extension that stores the aforementioned data.



(a) (b) (c)

Figure 3.1: Billboards illustration example[3]

The trajectory of the bus is **continuous**. However, PostGIS extension and in general traditional DBMS cannot effectively handle continuous data, as a continuous trajectory

[3]All figures of this subsection are taken from https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/analyzing-gps-trajectories-at-scale-with-postgres-mobilitydb-amp/ba-p/1859278

```
CREATE EXTENSION PostGIS;
CREATE TABLE gpsPoint (tripID int, pointID int, t timestamp, geom
    geometry(Point, 3812));
CREATE TABLE billboard(billboardID int, geom geometry(Point, 3812));

INSERT INTO gpsPoint Values
(1, 1, '2020-04-21 08:37:27', 'SRID=3812;POINT(651096.993815166
    667028.114604598)'),
(1, 2, '2020-04-21 08:37:39', 'SRID=3812;POINT(651080.424535144
    667123.352304597)'),
(1, 3, '2020-04-21 08:38:06', 'SRID=3812;POINT(651067.607438095
    667173.570340437)'),
(1, 4, '2020-04-21 08:38:31', 'SRID=3812;POINT(651052.741845233
    667213.026797244)'),
(1, 5, '2020-04-21 08:38:49', 'SRID=3812;POINT(651029.676773636
    667255.556944161)'),
(1, 6, '2020-04-21 08:39:08', 'SRID=3812;POINT(651018.401101238
    667271.441380755)'),
(2, 1, '2020-04-21 08:39:29', 'SRID=3812;POINT(651262.17004873
    667119.331513367)'),
(2, 2, '2020-04-21 08:38:36', 'SRID=3812;POINT(651201.431447782
    667089.682115196)'),
(2, 3, '2020-04-21 08:38:43', 'SRID=3812;POINT(651186.853162155
    667091.138189286)'),
(2, 4, '2020-04-21 08:38:49', 'SRID=3812;POINT(651181.995412783
    667077.531372716)'),
(2, 5, '2020-04-21 08:38:56', 'SRID=3812;POINT(651101.820139904
    667041.076539663)');

INSERT INTO billboard Values
(1, 'SRID=3812;POINT(651066.289442793 667213.589577551)'),
(2, 'SRID=3812;POINT(651110.505092035 667166.698041233)');
```

Listing 1: Billboards PostGIS database

includes theoretically infinite points. One solution, that is used in Listing 1, is to represent it with discrete points. In this database, we use `geometry(Point)` type which basically describes a point on the map. To model a trip, we use a set of `geometry(Point)` along with a `timestamp` to record the moment that the bus is located every time. Using the query in Listing 2 we can retrieve the locations of the buses that are within 30 meters from the billboards. Figure 3.2 visualizes the database as well as the result of the previous SQL query (circled point). With the previous SQL query, we retrieved the interesting bus locations but we did not take into account the duration that the billboards are visible to the bus, which was our initial question. To get this information from the current database, we need to write a much more complex SQL query (Listing 3) that splits the query into several sub-queries.

Figure 3.3 visualizes the result of the previous query as well as the intermediate CTEs. In Figure 3.3a, we see the interpolation of the database points. This is basically a connec-

```
SELECT tripID, pointID, billboardID
FROM gpsPoint a, billboard b
WHERE st_dwithin(a.geom, b.geom, 30);
```

Listing 2: SQL query to retrieve the locations of the buses within 30 meters from billboards

tion of every point with the previous and the next one, to introduce the continuity of the trajectory. In Figure 3.3b, it is depicted the start and end points of the bus from where the billboard were visible. In addition, the timestamp of both locations are marked in order to be able to compute the duration.

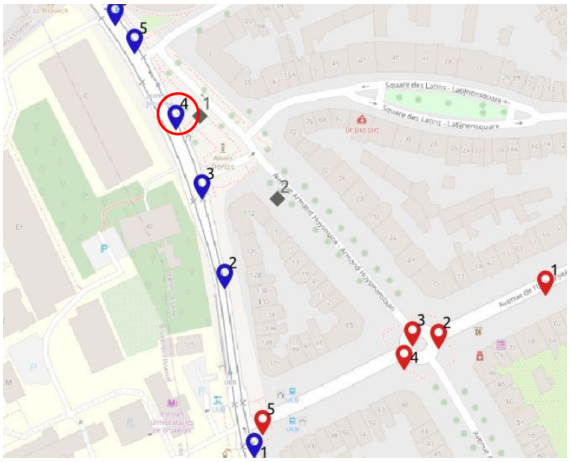
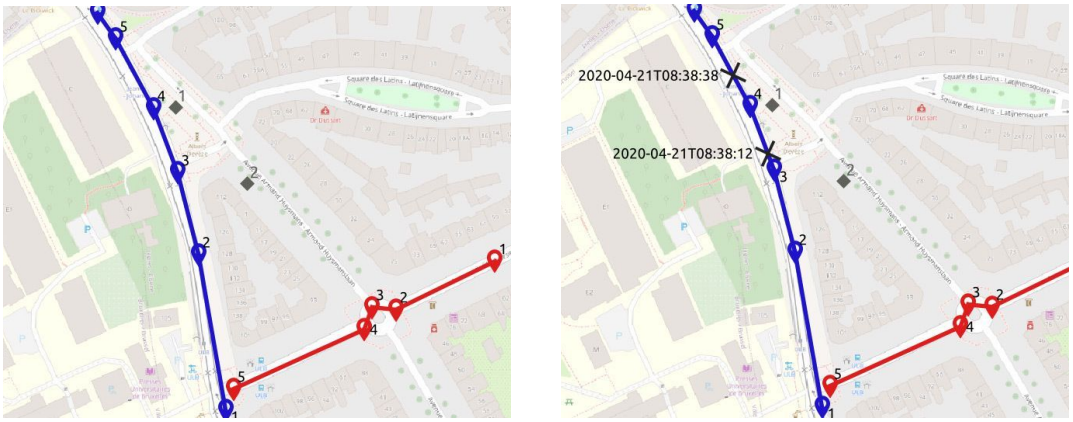


Figure 3.2: Database visualization

Consequently, although PostGIS extension enables the users to manipulate moving data, it is clear that writing analysis queries that include the concepts of continuity and spatio-temporal proximity is not an easy task, especially for non-experts. MobilityDB can simplify the management of moving trajectories. Now that we know one possible implementation of the billboards problem using PostgreSQL and PostGIS extension, we can see how the same problem can be modelled and solved using MobilityDB. First, we need to redefine the database schema. As we see in Listing 4, `busTrip` table includes two columns, `tripID` which is a unique identifier of the trip and `trip` which is the column that holds the information of the trips. We observe that for each trip, the table has only one row, which is not the case in the PostGIS database, where for each trip, the database stores one row for each instant of the trip. In the new database, the type of the `trip` column is `tgeompointseq`, which is a spatio-temporal type that describes both time and location. Listing 4 shows a sample of the table. Each trip is stored as a list of points along with a timestamp.

Having defined the database, we can see how we can find the buses that were visible from the billboards as well as the specific amount of time. Listing 5 illustrates how we can retrieved the data. The SQL is a simple join operation between the `busTrip` and the `billboard` tables. The join condition uses `dwithin` predicate that expresses that the distance between each location of the `trip` column may be within a distance of 30



(a) (b)

Figure 3.3: Query result visualization. (**a**) Visualization of `pointPair` and `segment` CTEs; (**b**) Visualization of `approach` CTE and final result;

meters to the `geometry` of the billboard. The nesting of `tdwithin->atValue->getTime` will return the corresponding time periods that the `where` clause is true and the function `atperiodset` will restrict the bus trip to only these time periods. As we see from the query result, the bus is visible from the moment `2020-04-21 08:38:12.507515+02` until `08:38:38.929465+02`.

Comparing the two equivalent SQL queries, namely Listings 3 and 5, we can clearly see that **MobilityDB facilitates the analysis of moving objects** and it is a DBMS that can be used by non-expert users after practicing and understanding the newly defined data types and operations.

In this section, we described the fundamentals of MobilityDB, which is a DBMS for moving objects. We also motivated its use by providing an illustration case where MobilityDB facilitates the management of trips of vehicles. In the next section, we present Citus, an extension that allows PostgreSQL at scale.

3.3 Citus

This section introduces Citus by describing its architecture and capabilities. Second, it presents the compatibility of MobilityDB and Citus extensions. Finally, it motivates the use of MobilityDB at scale, using Citus solution.

Citus Data [Micc] is a PostgreSQL extension that transforms any PostgreSQL server into a distributed relational database. Citus is an extension of PostgreSQL, hence, the user can leverage all the features, tooling and reputation of PostgreSQL. Citus seems to be a promising tool, ideal for any user that wants to use PostgreSQL at scale. By using Citus, scaling out a PostgreSQL server becomes very simple by distributing the data and the queries over a cluster of nodes using *sharding*. Such operations can be easily performed by any user with simple SQL commands that Citus provides. By introducing parallelization and depending on the size of the cluster, the user can experience 20x of

```
1  WITH pointPair AS(
2    SELECT tripID, pointID AS p1, t AS t1, geom AS geom1,
3      lead(pointID, 1) OVER (PARTITION BY tripID ORDER BY pointID) p2,
4      lead(t, 1) OVER (PARTITION BY tripID ORDER BY pointID) t2,
5      lead(geom, 1) OVER (PARTITION BY tripID ORDER BY pointID) geom2
6    FROM gpsPoint
7  ), segment AS(
8    SELECT tripID, p1, p2, t1, t2,
9      st_makeline(geom1, geom2) geom
10   FROM pointPair
11   WHERE p2 IS NOT NULL
12 ), approach AS(
13   SELECT tripID, p1, p2, t1, t2, a.geom,
14     st_intersection(a.geom, st_exteriorRing(st_buffer(b.geom, 30)))
   visibilityTogglePoint
15   FROM segment a, billboard b
16   WHERE st_dwithin(a.geom, b.geom, 30)
17 )
18 SELECT tripID, p1, p2, t1, t2, geom, visibilityTogglePoint,
19   (st_lineLocatePoint(geom, visibilityTogglePoint) * (t2 - t1)) + t1
   visibilityToggleTime
20 FROM approach;
```

Listing 3: Complex SQL query to retrieve the duration and locations of the buses within 30 meters from billboards

speed up comparing to a single node server. Another positive point of Citus is that it is open source and available in GitHub [Micd]. Apart from the open source version, Microsoft offers Citus as a service, via Azure Hyperscale [Hyp21]. **During the preparation of this master's thesis, we have been in close contact with the Citus engineering team, and more specifically with one of the inspiring members, Mr. Marco Slot, principal software engineer at Microsoft. Through this communication, we managed to get useful insights of Citus that lead us to make significant design choices.**

3.3.1 Citus Architecture

A Citus cluster mainly consists of two components, namely Citus coordinator(s) and Citus worker(s). Figure 3.4 illustrates the architecture of a Citus cluster. In a Citus cluster, all the queries are managed by the coordinator node. This node is responsible to receive the user query and create a query plan that splits the query into smaller query fragments that can be executed **independently** on the worker nodes. After the generation of the distributed query plan, the coordinator sends the sub-queries to the corresponding worker nodes and it monitors the process. When the worker nodes have locally fetched the results, they send them back to the coordinator where the sub-results are merged into one single result that will be returned to the user.

[4]Figure taken from https://www.citusdata.com/product#capabilities

```
CREATE EXTENSION MobilityDB CASCADE;

CREATE TABLE busTrip(tripID, trip) AS
  SELECT tripID,tgeompointseq(array_agg(tgeompointinst(geom, t) ORDER BY t))
FROM gpsPoint
GROUP BY tripID;

SELECT tripID, astext(trip) FROM busTrip;

--Query result:
1  "[POINT(651096.993815166 667028.114604598)@2020-04-21 08:37:27+02,
      POINT(651080.424535144 667123.352304597)@2020-04-21 08:37:39+02,
      POINT(651067.607438095 667173.570340437)@2020-04-21 08:38:06+02,
      POINT(651052.741845233 667213.026797244)@2020-04-21 08:38:31+02,
      POINT(651029.676773636 667255.556944161)@2020-04-21 08:38:49+02,
      POINT(651018.401101238 667271.441380755)@2020-04-21 08:39:08+02]"
2  "[POINT(651201.431447782 667089.682115196)@2020-04-21 08:38:36+02,
      POINT(651186.853162155 667091.138189286)@2020-04-21 08:38:43+02,
      POINT(651181.995412783 667077.531372716)@2020-04-21 08:38:49+02,
      POINT(651101.820139904 667041.076539663)@2020-04-21 08:38:56+02,
      POINT(651262.17004873 667119.331513367)@2020-04-21 08:39:29+02]"
```

Listing 4: Billboards MobilityDB database

Until now we described how Citus handles the distributed query execution but how is the data partitioned and spread over the cluster? Citus uses **sharding** to enable data distribution, which is a feature that can be used by running the *create_ distributed_ table* SQL command. By providing a sharding key, which is a column of the table that is meant to be distributed, Citus coordinator splits the table into shards that are assigned to the worker nodes. The coordinator only stores light weight meta-data about the location of each shard. The shards are assigned to co-location groups, meaning that tuples with the same shard key will be physically located on the same machine. Shading is basically horizontal partitioning of the target tables. Horizontal partitioning implies split of a table into fragments. In Citus extension, this operation is performed via hashing, which means that the data is distributed over the nodes according to the hash values of the selected column. Apart from the distributed tables, Citus also provides reference tables, which are typically small tables that are replicated on every node machine and can be used in combination with distributed tables.

Another Citus capability is **high availability**. Whenever a Citus worker node or the coordinator fails, Citus is able to recover. Regarding the managing of worker nodes failure, Citus proposes two alternatives. The first solution is to use PostgreSQL streaming replication [Groa] and it is mainly proposed for heavy OLTP workloads. With this approach, the system replicates the entire worker node by periodically syncing the data. Alternatively, Citus proposes shard replication, a mechanism to replicate the distributed shards over multiple nodes. In this way, once a worker node fails and does not respond to the coordinator, the sub-query can be re-routed to the other node that hosts the replica

```
SELECT astext(atperiodset(trip, getTime(atValue(tdwithin(a.trip, b.geom, 30),
    TRUE))))
FROM busTrip a, billboard b
WHERE dwithin(a.trip, b.geom, 30)

--Query result:
{[POINT(651063.737915354 667183.840879818)@2020-04-21 08:38:12.507515+02,
 POINT(651052.741845233 667213.026797244)@2020-04-21 08:38:31+02,
 POINT(651042.581085347 667231.762425657)@2020-04-21 08:38:38.929465+02]}
```

Listing 5: SQL query to retrieve the duration and locations of the buses within 30 meters from bill-boards using MobilityDB

shards. The Citus coordinator can also be an one-point failure for the cluster, as the whole architecture is based on it. To ensure high availability, several coordinator replica nodes can be maintained across the cluster by using PostgreSQL streaming replication. As the coordinator node does not store any data, its size is relatively small (typically a few MBs in size) and hence, it can be quickly replicated.

3.3.2 MobilityDB in Citus

As we have already discussed, both MobilityDB and Citus are extensions of PostgreSQL. However, deploying a MobilityDB server on a Citus cluster is not trivial and currently partially supported. This happens because Citus query planner is agnostic to the data types that Mobility defines on top of PostgreSQL. Bakli et al. [BSZ20b] recently researched to what extent Citus supports MobilityDB operations. Thanks to the fact that MobilityDB operations are built using the extensibility features of PostgreSQL and that Citus accounts for such an extensibility, we can say that Citus supports at a satisfactory level MobilityDB.

As explained by Bakli et al. [BSZ20b], MobilityDB operations can be classified into four classes, namely *spatiotemporal joins*, *temporal aggregations*, *spatiotemporal index access* and *limit operations*. Starting with the first class, *spatiotemporal joins* can be splitted into two distinct classes, **the co-located joins** and the **non co-located joins**, and a query is classified to one of these categories depending on the MobilityDB predicates that are used in the join operation. In **co-located joins**, every worker can locally perform the join operation using its own shards and hence **this type of queries are fully supported**. On the other hand, **non co-located joins** require redistribution of the shards, as the partial join operations cannot be performed on each node locally. **This type of queries cannot be executed with the current version of Citus**. Listing 6 shows an example of these types of queries. In this example, `trips` table is distributed while `municipalities` is a reference table, replicated to every worker node. The first query of Listing 6 is a co-located join query that includes several MobilityDB-defined operations and is accepted by the Citus query planer, as the partial join operations can be executed on each node. This is true as the entire `municipalities` table is located on every worker, hence, the partial joins between the shards of `trips` and `municipalities` can be computed on every node and the results can be merged in the coordinator. On the contrary,

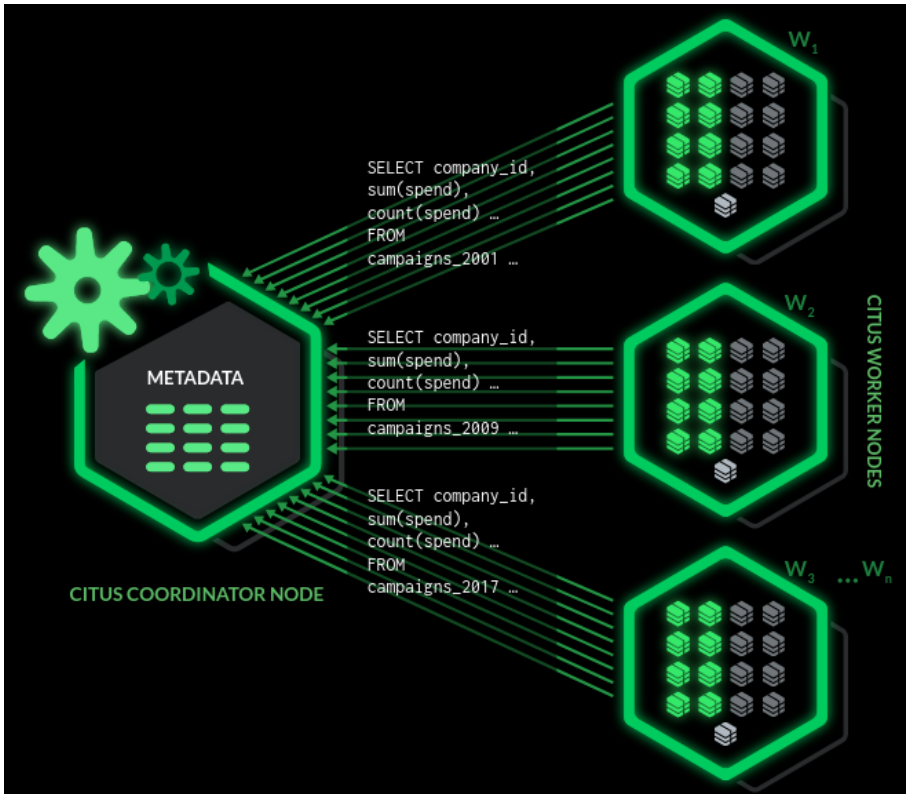


Figure 3.4: Citus architecture[4]

the second query of Listing 6 is a non co-located join query and is rejected by the Citus query planner. In this case, we try to perform a self join of a distributed table, which will require data re-distribution, as it is highly likely that shards belonging to different nodes should be joined together.

Moving on to the *temporal aggregations*, MobilityDB defines a variety of temporal aggregation functions that are not supported by Citus. Bakli et al. [BSZ20b] proposes a way to enable the execution of temporal aggregations that are both commutative and associative. Finally, both *spatiotemporal index access* and *limit operations* are supported out-of-the-box, as the queries can be pushed down to individual shards.

3.3.3 Illustration Case

In Section 3.2.2, we solved the billboards problem using two different PostgreSQL extensions, namely one with PostGIS and another one with MobilityDB extension. In this subsection, we will see how we can easily use MobilityDB at scale, exploiting the capabilities that Citus provides.

We remind that in the database of Listings 1 and 4, we define two different tables, one containing the locations of the billboards and another one the trips of the buses. Working on a big data environment, we would like to analyze vast amounts of trips, hence, the `busTrips` table can contain millions of trips. In addition, the `billboard` table is relatively small, as it contains only some thousands rows. In such a scenario, we can easily understand that the computational power of a single PostgreSQL server would not

```
--Example of co-located join--
SELECT avg(duration(t.trip))
FROM Trips t, Municipalities m
WHERE m.name like '%Uccle%' AND
    ST_Intersects(startValue(t.trip), m.geom);

--Example of non co-located join--
SELECT a.vehicle, b.vehicle, a.day
FROM Trips a, Trips b
WHERE a.vehicle < b.vehicle AND
     a.day < b.day AND
     a.seq < b.seq AND
     tdwithin(a.trip, b.trip, 10) %= TRUE;
```

Listing 6: Co-located vs non co-located joins

be enough to handle multiple analysis requests. Below, we describe how we can easily migrate the previous database to a Citus cluster.

Assuming we have installed and set up a Citus cluster, after connecting to the coordinator node, we can perform the following steps to create a distributed database. First, we create the database in the same way that we did in Listings 1 and 4. Then, we simply execute the SQL commands of Listing 7 to create and distribute the data over the Citus cluster. The `create_reference_table` function is used to replicate the `billboard` table over the cluster nodes. Typically, when the size of a table is small, we replicate it over every node, as the cost is low and the performance gain, in terms of speed up, is high. The `create_distributed_table` function distributes the `busTrip` table using the `tripID` as distribution column. With this action, the table will be splitted into shards that will be assigned to different worker nodes.

```
SELECT create_reference_table('billboard');

SELECT create_distributed_table('busTrip', 'tripID');
```

Listing 7: Billboards distributed database

After having run these two functions, **the user can keep executing the same SQL commands as before**. For instance, the table can be loaded with millions of data with the traditional `INSERT` commands or with `BULK` load and then, the query in Listing 5 or any other valid SQL query can be executed on the coordinator node. **Hence, the user can experience all the benefits of Citus, especially the speed up, with minimal effort**.

In this section we described the capabilities of Citus, as a distributed version of PostgreSQL. In the next section, we introduce Docker, a platform that facilitates the deployment of application on different machines.

3.4 Docker

This section presents the core idea behind Docker, which is a platform that facilitates the deployment of applications using containers.

One major issue in computer science is the portability of the applications from one machine to another. When developing a complex application, it is usually coupled with many other software and libraries that need to be installed on a system to run it. Before running the same application on a different machine, the dependent software needs to be installed first. The imperative need for quick portability becomes even bigger within cloud environments, where customers would like to reap the benefits of scalability. As cloud services enable quick scalability, it is evident that the application and all its dependencies should be easily transferred to the newly created machines. Docker, which was originally introduced by Merkel [Mer14], is an efficient way of making applications portable.

Docker is an open-source platform for developing, shipping and running applications. It enables users to decouple the application from the underlying infrastructure. In this way, the same application can normally run on every machine that runs Docker engine. The core ideas behind Docker are the Docker containers and Docker images.

- A **Docker container** provides the ability to package and run an application in a loosely isolated environment. This means that many containers with different applications can run on the same host. Containers are lightweight and they do not depend neither on the operating system of the host machine nor on the installed software. They contain everything needed to run the application.
  A Docker container constitutes a runnable instance of a Docker image. We can think of containers like very lightweight virtual machines that hosts applications. Figure 3.5 depicts the difference between containers and virtual machines. The major difference, which makes Docker containers much less bulky compared to the virtual machines, is the lack of the operating system. Virtual machines require their own operating system which makes them bigger in size and less performant. In contrast, Docker containers share the underlying OS kernel which allows them to start almost immediately, as their size is small.
- A **Docker image** is a read-only template containing instructions of how to create a Docker container. It is often that an image is an extension of another image. For instance, it is possible to create a container that is based on an Ubuntu image and it also installs Apache web server and PostgreSQL database. In this work, we create an image based on MobilityDB and then we install Citus extension as well as some additional extensions to monitor different database metrics. Docker images can be shared in a public registry and then being pulled by several machines at any time.
  Docker images are quite powerful especially when software updates are introduced. Imagine a scenario where a web application runs for over a year and its back-end is a PostgreSQL 9.0 server. After one year in the production, we decide to upgrade the server to version 11.0. Such a change can be very easily performed when the PostgreSQL server runs on Docker, as the only required change is related to the image. More specifically, a new Docker image should be created, based on the image of PostgreSQL 11.0, and a new container should be launched by the newly created image. In this way, the users will experience almost zero downtime of the application,

as the switch between the two containers can be performed in a few minutes.

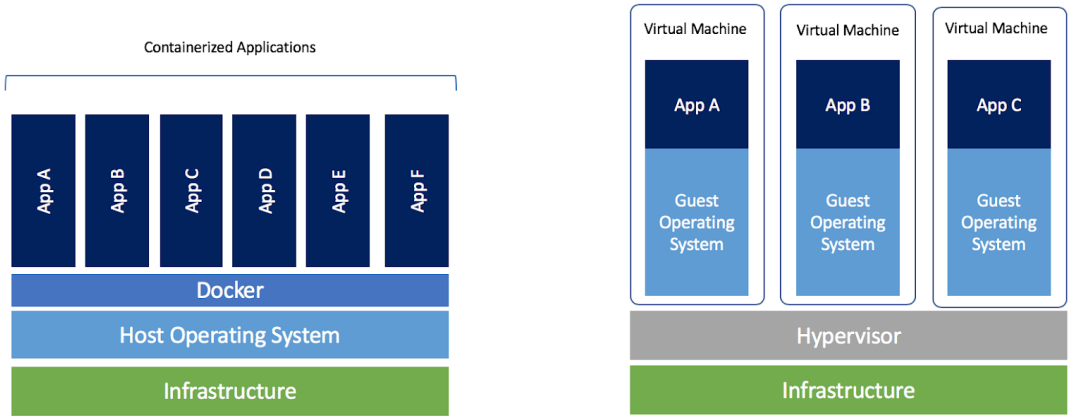


Figure 3.5: Containers vs virtual machines[5]

In conclusion, Docker is a container-based platform that facilitates the deployment of portables applications. A major task of this work is the automation of several tasks that are related to the deployment of MobilityDB. Hence, Docker containers are mainly used to host the instances of MobilityDB and allow fast scaling. In the next section, we present Kubernetes tool, which is a framework used to manage containerized applications deployed on Docker.

3.5 Kubernetes

This section describes Kubernetes platform. First, it presents some of the reasons that make Kubernetes one of the most widely used tools in production environment today. Moreover, it describes some of the core components of Kubernetes as well as some workloads that Kubernetes provides in order to host user-defined applications.

As discussed until now, Docker seems to be the ideal technology for shipping and running containerized applications thanks to its portability and scalability. Although Docker containers introduce many advantages, they also add extra complexity and several challenges that we need to consider before using them. The biggest challenges can be detected when an applications is comprised of many containers. Docker runtime environment is well suited to handle single-container applications but struggles when many containers exist for the same application. Tasks like scheduling, load balancing and autohealing are not placed under the umbrella of Docker runtime. Such tasks require an external orchestration tool, responsible of managing the set of containers. Kubernetes is an ideal open-source platform for this purpose.

Kubernetes (K8s) [Kube] was originally proposed by Google in 2014. The name Kubernetes originates from Greek, meaning helmsman or pilot. As its name says, Kubernetes is a platform designed to automatically manage containerized applications. It is used to

[5]Figure taken from https://www.docker.com/blog/containers-replacing-virtual-machines/

schedule such applications into a computer cluster and ensure that the deployed applications are working as intended. To describe how the deployment is expected to operate and be managed, the developer uses a declarative and infrastructure-agnostic language, written in *yaml* or *json* files. To better understand the capabilities of Kubernetes, the following subsection highlights some of the most useful features.

3.5.1 Kubernetes Features

Kubernetes provides a platform to run distributed systems resiliently. Assuming a company that runs a distributed application, consisted of many containers, and one of them fails. Some mechanism should interfere by restarting the failed container in order to keep the application alive. Kubernetes is such a framework that provides automatic failover and other features [Kubf], some of them being listed below:

1. **Service discovery and load balancing**. K8s provides a way to expose the containerized applications via *services* by using the distinct DNS name or the IP address of each application. Depending on the traffic of the containers as well as other metrics that the user can choose, K8s is able to load balance and distribute the workload to increase the efficiency of the system.
2. **Autohealing**. K8s maintains the health of the system according to the user-defined health checks. Each time that a container does not respond or has failed, K8s tries to restart or replace it in order to keep the application running. All these actions are done without human intervention.
3. **Scalability**. K8s can be used to easily scale in or out the application, by simply removing or adding new containers to the cluster.
4. **Storage orchestration**. Containers are known to be stateless components as they are launched in the computer's memory and once they are terminated, they do not store any state to the hard disk. K8s allows to automatically attach external storage to the containers, as there are containerized applications that are meant to be stateful. The storage can be placed locally on the same machine that hosts the container or it can be hosted on the cloud.
5. **Version control**. With K8s and Docker containers, updating the application to a newer version becomes trivial. It is enough to create the new Docker image that includes the new features of the application and launch a new container by using the newly created image. All these actions can be scheduled with K8s to be automatically performed. In case that the new deployment is not stable, K8s can decide to switch back to the old deployment to keep the application alive.

3.5.2 Kubernetes Components

Kubernetes architecture is based on a cluster. A K8s cluster may include several components [Kubc] depending on the requirements of the application. This subsection is devoted to an overview of the available components of a K8s cluster.

K8s follows a master-slave architecture where the cluster consists of at least one **worker node** machine. There is always one or more **Control Plane(s)** that manages the deployed applications. The applications are running inside **Pods**. A Pod represents a set of running

containers in a cluster. Both Control Plane and Pods are hosted by the worker node machines. Figure 3.6 is a sample diagram of a K8s cluster.

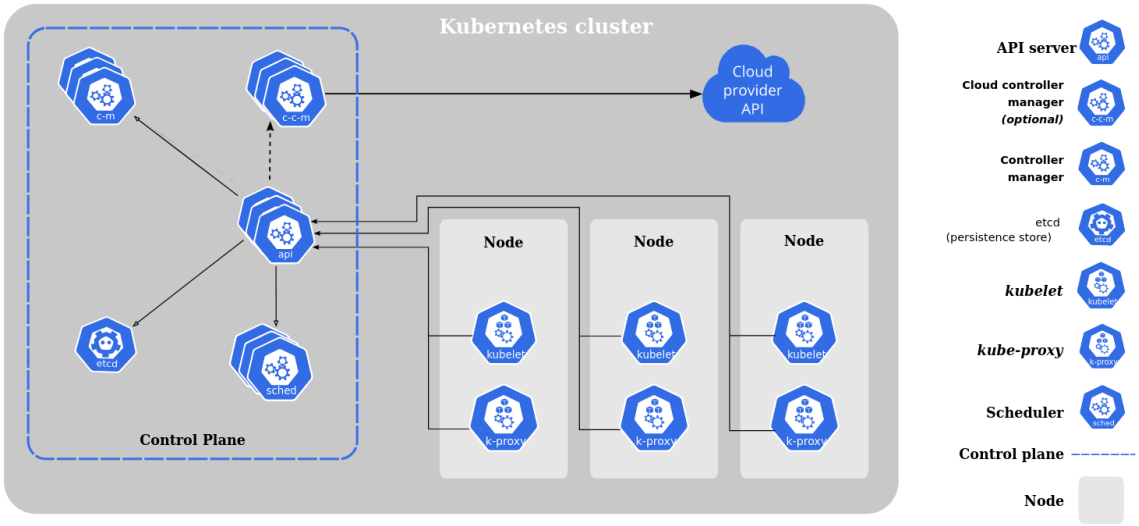


Figure 3.6: Kubernetes cluster diagram[6]

3.5.2.1 Control Plane Components

Control Plane is the master of a K8s cluster. It is responsible for the smooth operation of every component, it has all the information for the cluster members and it continuously monitors them. The components of a Control Plane can be assigned to the same worker node but typically for production environments, they are spread across different worker nodes to ensure fault-tolerance and high availability. In addition, it is possible to maintain more than one Control Plane to further reinforce the highly available nature.

1. **Kube-apiserver**. This component is the front end for the K8s Control Plane. It it responsible for the communication between the applications that run on top of the cluster. It also takes care of the interaction with the users, as all the user operations pass from the API and then they are applied to the cluster. The API server is designed to scale horizontally, hence, it scales by deploying new instances.
2. **Etcd** is the back end of the cluster. It stores critical configuration information, like the desired and the actual state of the cluster as well as changes on both states in order to monitor the system. As K8s is a distributed cluster, etcd is also a distributed key-value store that is spread across the cluster nodes.
3. **Kube-scheduler** is a key component of the Control Plane. It is a service that distributes the workload across the available worker nodes of the cluster. Each time that a newly created Pod arrives, the scheduler looks for available resources and assigns the Pod to the corresponding worker node. The assignment of a Pod to a worker node is performed based on several factors. Someone may define some hardware constraints, where a Pod cannot be assigned to a node with less than 2GB of RAM available or another affinity specification, where two Pods of the same

[6]Figure taken from https://kubernetes.io/docs/concepts/overview/components/

deployment cannot be hosted by the same node. In Section 4.2, we describe the specific scheduler constraints for the deployment of MobilityDB on Kubernetes.

4. **Kube-controller-manager** is the component that runs all the **controller** processes of the cluster. Controller is a daemon loop that monitors the shared state of the cluster and attempts to move the current state of the system towards the desired state. The following example contributes to better understanding the difference between the current and the desired state. **K8s works in a descriptive manner**, as the user provides several *yaml* files that express the desired state of the deployment, and K8s continuously tries to turn the system as closer as possible to this state. For instance, supposing that a user manifests that he/she wants to deploy three instances of a PostgreSQL server on three different worker nodes, but currently there are only two available worker nodes on the cluster. K8s scheduler is going to detect that there are only two machines available and it will create only two servers. Then, the controller will periodically try to modify the current state of the system, by creating one more Pod once the third server is available. This back and forth move between the two mechanisms will continue until a third worker node is assigned to the cluster and a new Pod is created or until the user changes the manifest to a smaller number of PostgreSQL instances.
5. **Cloud-controller-manager** is similar to the Kube-controller-manager, but it is responsible for controlling the components provided by the cloud provider. The Cloud-controller-manager is an optional component as it does not exists for on-premises clusters.

3.5.2.2 Worker Node Components

Worker nodes are the part of the cluster that actually hosts the deployed applications. Worker node components run on every node and are responsible for maintaining running Pods and providing the Kubernetes runtime environment.

1. **Container runtime** is the underlying software on top of which containers are running. K8s supports several runtimes, including Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container runtime interface).
2. **Kube-proxy** is the mechanism that handles the networking of the worker nodes. It makes the services of the Pods available to the external hosts. In this way, the exposed services can be accessed by network sessions that are both inside and outside of the cluster. Kube-proxy takes over the redirection of the incoming requests to the right container and sends back the response to the kube-apiserver.

3.5.3 Kubernetes Workloads

K8s provides to the users different types of workloads in order to be expressive enough to describe a wide range of applications. Workload resources are used to automatically manage a set of Pods that represents the user applications. These resources are using controllers to systematically monitor the state of the application and ensure that the Pods are running correctly. The following list provides explanations on some of the most popular K8s workloads. A combination of the available workloads can also be considered to express more complex applications.

- **Deployment** is one of the simplest resource objects of K8s. A Deployment allows to define the application's life cycle, such as how many Pod replicas will be used, which image will instantiate the container, which commands will be executed on the container right after its initialization and more.
  A Deployment ensures that the desired number of replicas are running and being responsive at all times. The operations that can be performed on a Deployment are the following: deploy, pause, continue and terminate a set of Pods, update it, roll back to a previous version and horizontally scale the set.
- **StatefulSet** is a K8s resource object mainly used for stateful applications. Similarly to a Deployment, StatefulSets can be used to manage set of Pods with identical specifications. This means that this object enables replication. On the contrary, StatefulSets maintain a sticky identity for each of their Pods, which is not the case for the Deployments, where each Pod is interchangeable.
  For a StatefulSet with N replicas, the N Pods are created sequentially, one after the other. Hence, Pod 3 cannot be created unless Pod 2 is ready and running. For example, in the following scenario where the current state of the StatefulSet is equal to the desired state and has two replicas, if the user requests to scale it to four replicas, then the process goes as follows. First, the third Pod will be launched and the controller will wait until it is ready and running. Then, the forth Pod will be instantiated as well. Similarly, Pods are deleted in reverse order, one after the other.
  As StatefulSets are designed for stateful applications, the existence of disk space is required. K8s has introduced **Persistent Volumes** (PV), which are resource objects that can be attached to Pods and play the role of hard disks. Persistent Volumes cannot be accessed directly by the Pod, but they can be accessed by using **Persistent Volume Claims** (PVC). PVCs are requests for storage by users.
- **DaemonSet** is another resource object available in K8s that ensures that some or every worker node has a copy of the deployment's Pods running on it. As nodes are added to the cluster, Pods are assigned to them, while as nodes are deleted, the corresponding Pods are garbaged collected. DaemonSets are working with DaemonSet controllers that are responsible for managing the aforementioned process.
  Some use-cases of DaemonSet are node monitoring processes, logs collection and several other processes that the user desires to be executed across every node of the cluster.

Apart from the different types of workloads, K8s includes several other features that further assist the expressiveness of the tool. Below, we describe Node Affinity feature that will be later used in our application.

- **Node Affinity**. By default, when using a Deployment or a StatefulSet, K8s scheduler tries to assign the newly created Pod(s) to the less overloaded node(s). Hence, each Pod can be assigned to any available node. However, there can be scenarios where the user does not want several Pods to be hosted by particular nodes or it can be required that a specific Pod should be assigned to a specific node. In K8s, it is possible to impose such constraints by using Node Affinity/Anti-affinity. In practice, the user links the node(s) with labels, represented as key-values, and he/she uses these labels in the definition of the Deployment/StatefulSet to determine whether the deployed Pod(s) should/should not be assigned to the node(s) with the given

labels.

This section included the Kubernetes fundamentals that allow the automatic orchestration of containerized applications. The next section introduces DevOps principals as well as DevOps lifecycle.

3.6 DevOps

This section provides the fundamentals behind development and operations (DevOps) principals. First, it explains what is DevOps and how it facilitates different procedures of IT projects. Moreover, it presents the lifecycle of a typical DevOps project. It is widely known that the era of new technologies has introduced new opportunities and challenges for tech companies. Continuous development is a major challenge that most companies struggle to deal with. Specifically, new demands and specifications arise daily from the business and hence, new technical implementations are required to cover these needs. Although there are known techniques since 90's that assist companies on being more agile and ready to efficiently solve such issues, they have been applied and developed only the last decade. DevOps model comes to bridge this gap by enabling teams to deliver the code faster and with better quality.

3.6.1 DevOps Definition

A formal definition of DevOps provided by [BWZ15] is *"a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality"*. Another definition, given by Microsoft,[7] characterises DevOps as *"a compound of development (Dev) and operations (Ops). DevOps is the union of people, process, and technology to continually provide value to customers"*. From the first definition, we can understand that DevOps has to do with a set of practices that increase the efficiency when modification needs to be applied to a system while from the second one, the notion of mixing two teams, namely development and operation, is introduced. Indeed, for many years, tech companies have been spending a lot of time and effort before launching their products because of the orchestration of the different departments that were involved in the process. The following example highlights the value of DevOps nowadays. Imagine Company C that sells software to its clients. This company splits the development of each new product in the following stages, namely *development, operations and quality assurance*, and each stage has a dependency on the previous stage. This means that the operation processes start when the development is done and the quality assurance when the operation processes end. Such a workflow introduces many challenges, such as back and forth moves between the stages when bugs are detected or when the clients request new features to be added to the product. All these issues add extra cost to Company C and delay the final release of the product. A more efficient way to produce it would be to combine and coordinate all the different teams that are involved in the production process. In this way, altogether will contribute to **incrementally build** better and of high quality products. Such an orchestration is achieved when DevOps practices are adopted.

[7] DevOps (Microsoft) definition https://azure.microsoft.com/en-us/overview/what-is-devops/

3.6.2 DevOps Lifecycle

When it comes to DevOps practices, it is useful to divide the whole pipeline of tasks into different phases. Figure 3.7 is a well known diagram that depicts all the processes that form the DevOps lifecycle. As we see, DevOps lifecycle consists of the following eight continuous steps:

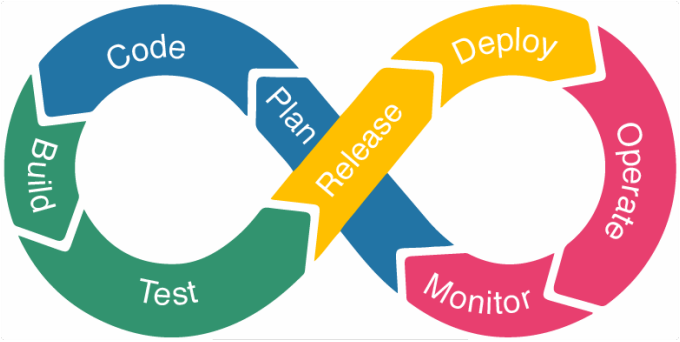


Figure 3.7: DevOps lifecycle[8]

1. **Plan** stage includes every action that takes place before the coding of an application starts. During this step, the organisation needs to gather and analyze all the customer specifications, in order to plan the next phases in the best possible way. The product of this stage is a roadmap that will guide the future development, by defining the order in which each feature of the application will be implemented, the estimated effort that will be needed and all these details that are related to the planning of a project.
2. **Code** is the step of the lifecycle where the actual development work happens. During this stage, the members of the development team collaborate in order to create the expected deliverable of the current sprint. A **development environment** is used with the required plugins installed. One important part of the development process is the storage of the source code. More specifically, a shared code repository is used by the team, where each member works locally in his/her repository and pushes the produced code, when it is ready, to the shared one. In this way, there is always a single source of truth, meaning that every member has access to the same source code.
3. **Build**. As we already mentioned, once a member of the team has completed a feature, the local code needs to be submitted to the shared repository. To do so, the developer needs to submit a pull request, which is a merge request of the new code with the shared codebase. The merge of the code will actually happen after another member of the team has reviewed and accepted the new code. In this way, naive errors are avoided and identified at an early step.
   At the same time, another automated processed has been triggered, the aim of which is to build the new codebase and run a set of predefined tests to identify potential code issues. If the tests fail at any step, the developer is notified to resolve the issues

[8]Figure taken from https://medium.com/@yildirimabdrhm/devops-lifecycle-continuous-integration-and-development-e7851a9c059d

and rerun the process until no problems arise. Similarly with the previous step, the goal is to avoid early mistakes that would cause more severe problems later.

4. **Test**. All the previous steps are done in a **development environment**. Once the built is successful, it is deployed on a **staging environment**. Typically, different environments are used for developing and testing purposes, as different teams may be responsible for each phase. However, most of the times, the replication of an environment is a difficult process, especially for complex applications with many modules installed. **Infrastructure-as-Code** (IaC) is a common practice of automatically provisioning a new environment at the time of deployment and is a crucial part of DevOps.
   During this stage, several manual and automated tests are performed. User acceptance test (UAT) team is responsible to perform the manual test cases, where users use the application as customers in order to identify potential bugs. In addition, several automated tests are performed in order to find security threats, compliance or performance issues and many more.
5. **Release**. Once the test phase has been successfully passed, the build is ready for deployment on the **production environment**, by following the same steps that the team performed to deploy the build on the staging environment. Taking as granted that the staging environment is identical to the production one and that all the previous tests were passed, the DevOps team can be confident that no breaking issues will arise.
6. **Deploy**. We previously talked about the power that IaC gives to the DevOps team. Such a capability can further enhance the performance of the team by assuring almost no downtime to the application. As we already mentioned, IaC can be used to create the staging environment where all the tests are performed. As we have already passed this stage, we are sure that the staging environment was built successfully and hence, the same code can be reused to create a new production environment. This new production environment will include all the newly developed features and hence, all the new incoming customer requests can be redirected to it. If at any point a bug is found, the incoming requests can be redirected back to the old production environment until the issue is fixed.
7. **Operate**. As the new features have been released to the production environment, the customers are able to use them and provide feedback. Such feedback is collected as it is very important for the DevOps team. The most important thing is to keep the customer satisfaction high and the only way to achieve it is by receiving their feedback and act accordingly.
8. **Monitor** phase is where the previously collected feedback is analyzed in order to measure the customer satisfaction. Additionally, the DevOps team monitors the whole DevOps lifecycle to identify errors, bottlenecks and issues that decrease the performance of the team. All this data is fed back to the organisation and to the development team to take the required actions. As we understand, at this point the DevOps loop starts again which is what makes it a **continuous process**.

In this section, we described the core idea behind DevOps, as a methodology that aids development and operations teams to work better together. The next chapter describes the automation of the roll out process of a MobilityDB cluster on Azure using K8s as well

as the K8s manifests that are used to maintain such a deployment ready and healthy.

Chapter 4

Cloud Database Automation and Management

Automation is a fundamental characteristic of a self-managed database management system. This chapter presents the strategy that was followed in order to introduce automation, by applying different DevOps principles. First, we highlight the use of DevOps in order to automate several deployment processes of this work. After having deployed our cloud database, we present the way that it is automatically managed using Kubernetes tool.

4.1 Cloud Automation

4.1.1 Applying DevOps via Automation

Although this work is not a pure DevOps project, as DevOps presupposes the existence of at least two teams, it adopts several DevOps principals to finally provide an easy-to-use and of high quality product. In this subsection, we are going to present a big part of this work that is related to IaC that was explained in Section 3.6. In the next sections, additional DevOps practices that were used in this work are explained, like Docker containers and cloud services that are mostly related to the deploy stage of the DevOps lifecycle as well as Kubernetes which is related to the operate stage.

As we have already mentioned, MobilityDB is an extension of PostgreSQL, which makes someone think that it has several dependencies on other software. In other words, in order to deploy and run MobilityDB on a system, several other components need to be installed and configured before. In addition, when working on the cloud, the system administrator needs to allocate and configure the desired resources that will host the application. All these tasks require deep knowledge and understanding of the cloud technologies and the low level system configurations, that would be quite difficult for a user to quickly get. Furthermore, even for advanced DevOps engineering teams that are familiar with such tasks, it is convenient to work with IaC, as the creation of multiple operating environments becomes trivial. For these reasons, with this work we provide some alternative ways that enable the user to **automatically** allocate the required resources in MS Azure and install all the components needed to use MobilityDB and Citus on the cloud.

IaC is key when it comes to process automation on the cloud. Most cloud providers offer a way to automatically access and manage the hosted infrastructure. In the remaining of this subsection we explain how the Azure resources are automatically allocated to enable IaC.

1. **Resources Allocation:** MS Azure provides two ways of accessing and managing its

cloud services. The first way is MS Azure Portal[1] which is a graphical environment to build, manage, and monitor everything from simple web apps to complex cloud applications on a single, unified console. The second way is MS Azure command line interface (CLI),[2] which consists of a set of commands to create and manage Azure resources, designed to get the user working quickly with Azure, with an emphasis on automation. For the allocation of the required resources in MS Azure, we mainly use *Linux bash script* commands as well as *MS Azure CLI*. In the following list, they are presented the most important MS Azure CLI commands along with a short explanation:

- `az login`
- `az group create`
- `az vm create`
- `az vm run-command invoke`

az login command is used to login to MS Azure with CLI. This is the only moment where the user needs to interact with the code in order to generate the resources. After providing the credentials, a connections between MS Azure CLI and the user's account will have been established and the script will be able to continue.

az group create command creates a new resource group[3] in MS Azure. A resource group is a container that holds related resources for an Azure solution. All the resources that will be created by the scripts will be placed under the same resource group.

az vm create command is used to create a new VM.[4] A VM is a virtual computer that behaves like an actual computer. A VM is actually a software that runs in a computer, physically being placed on a Microsoft data center, giving the end user the experience that he/she owns a personal computer. However, this is nothing more than an illusion, as multiple VMs machines can run simultaneously on the same physical computer, each of them running its own operating system and functions.

az vm rum-command invoke remotely executes a single command in an already created VM. For instance, such a command can be used to execute a bash script file that has been previously cloned to the VM by a public GitHub repository.

4.1.2 Automating the Deployment of a Management System

One of the primary goals of this work is to provide a Database-as-a-Service solution of MobilityDB on the cloud. By providing such a service to the end user, we assure that MobilityDB, and even more its scalable version, is available to everyone, from a person with very few technical skills in the database and cloud domains, to a more advanced user. More specifically, the configuration of MobilityDB and Citus extension on Kubernetes using cloud infrastructure, requires good understanding of several aspects, like networking, bash scripting and database administration but also deep understanding of the underlying technologies (e.g. Kubernetes) as well as cloud technologies and distributed environments.

[1] MS Azure Portal https://azure.microsoft.com/en-us/features/azure-portal/

[2] MS Azure CLI https://docs.microsoft.com/en-us/cli/azure/

[3] Azure resource group https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/manage-resource-groups-portal

[4] Virtual machine https://azure.microsoft.com/en-us/services/virtual-machines/

Hence, it is evident that even for an experienced person, it may take time to design and implement an efficient architecture with all these components. To tackle this issue, we provide an easy-to-use solution that demands minimum human interaction, with which someone can quickly deploy MobilityDB with Citus extension on a Kubernetes cluster using MS Azure resources. The process of the generation of the cluster is illustrated in Figure 4.1.

As we see from the illustrated process, the user needs to interact with the script only in the beginning and after the end of the cycle. Specifically, before the start of the job, a configuration tab needs to be filled with the desired parameters. More details on how to configure the script can be found in the tutorial in Appendix B:

- ***AzureUsername:*** The email used to login to MS Azure.
- ***ResourceGroupName:*** The name of the resource group that will be deployed on MS Azure.
- ***Location:*** The physical location of the VMs. One location from the complete list of locations[5] should be selected.
- ***VirtualNetwork:*** The name of the virtual network that will be created.
- ***Subscription:*** The name of the user's subscription in MS Azure that will be used to bill the cost of the current deployment.
- ***VMsNumber:*** The number of **worker** VMs that will be created (> 0). By default, the program will generate one coordinator VM and **VMsNumber** workers.
- ***VMsSize:*** The computational power of each VM. One from the complete lists of VM sizes[6] should be selected.
- ***SSHPublicKeyPath and SSHPrivateKeyPath:*** These values specify the location of the ssh private and public keys to access the created VMs. By default, Azure CLI stores the generated key-pairs under *∼/.ssh/* directory and hence, these values **should not** be changed.
- ***Gitrepo:*** The link to the GitHub repository of this work. **The default value should be used**.
- ***Service_app_url:*** The url of the Service Principal to be used to access the Azure resources.
- ***Service_tenant:*** The id of the tenant of the Service Principal.

[5] Azure geographies https://azure.microsoft.com/en-us/global-infrastructure/geographies/#overview

[6] Azure VM sizes https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/

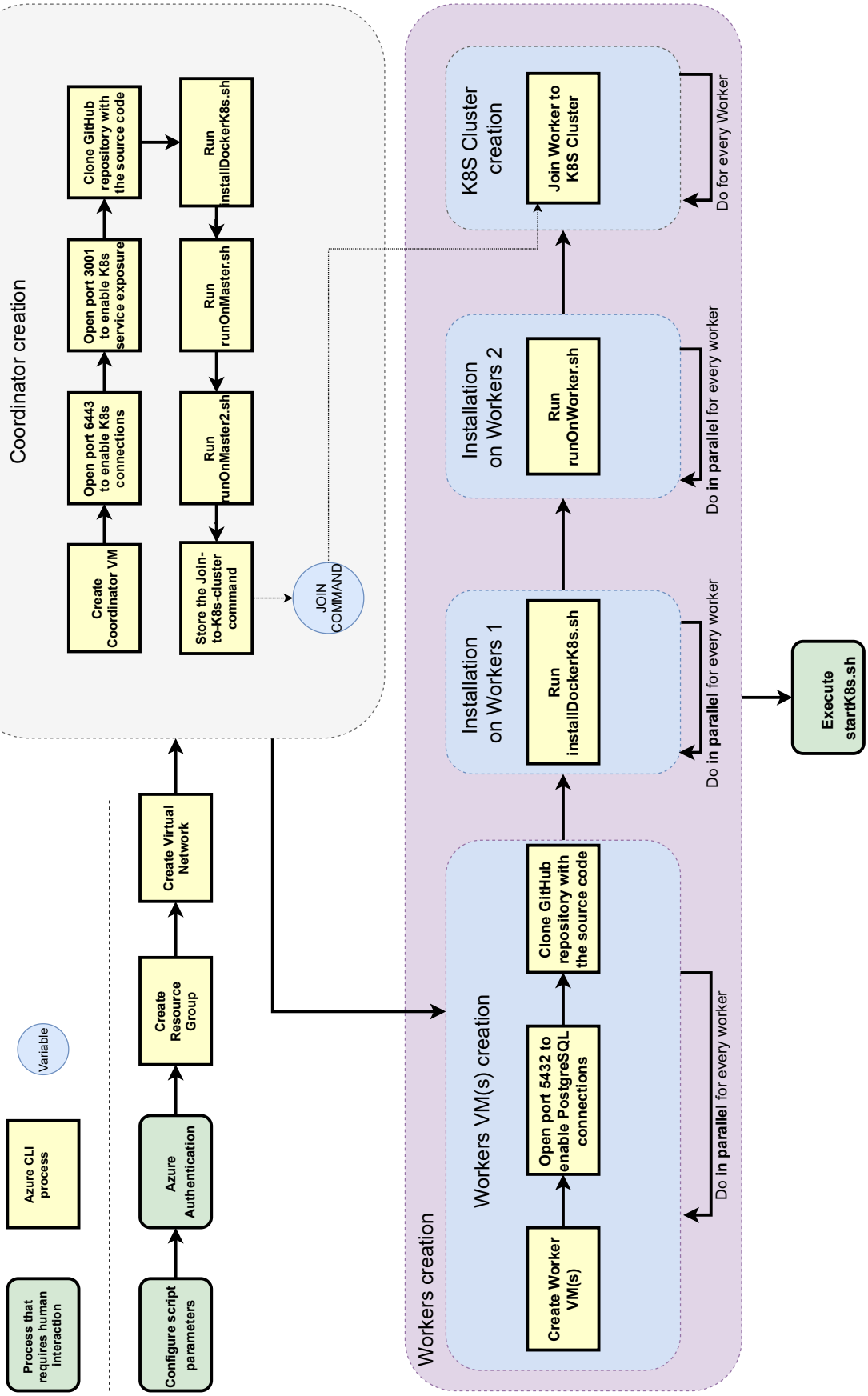


Figure 4.1: Automatic roll out process of a MobilityDB cluster

After all the parameters are given to the script, the user needs to launch the bash script. The user is asked to provide his/her Azure credentials in order to authenticate him/herself to the platform. After the authentication is done, the whole process is under the responsibility of the program. Finally, when the deployment is done, the user needs to establish an SSH connection with the coordinator VM and run the corresponding bash script that deploys the K8S solution. It is noted that the overall process takes approximately 17 minutes to terminate, **regardless of the number of worker nodes**. As the process of creating the worker VMs is done in parallel, the total execution time does not increase linearly when more worker VMs are requested, but it remains almost stable.

In this section, we presented the automated process of generating a MobilityDB cluster on MS Azure using IaC and other DevOps best practices, key features of cloud native solutions. In the next section, we describe the K8s deployment that enables the automatic orchestration of the deployed tool.

4.2 Cloud Database Management

In Section 4.1, we motivated the use of some DevOps practices to facilitate the allocation of the required cloud infrastructure that will host the MobilityDB cluster. In addition, we explained the process that installs the required software, assisted by Docker containers. After the successful termination of the cluster generation process of Figure 4.1, we should have allocated all the resources and a K8s cluster should be running. The last step of the process creates the Kubernetes deployment that is described in this section. More specifically, we explain the chosen K8s architecture and the specific configurations that are listed in the *yaml* manifests that describe the desired state of our K8s deployment.

4.2.1 Architecture Diagram

Figure 4.2 depicts the architecture diagram of a K8s cluster that hosts a Citus cluster, consisting of several MobilityDB instances. Starting from the top of the diagram, the first component that we observe is **Kubernetes Control Plane**. In this diagram, the sub-components of the Control Plane are skipped as they are identical to the components described in Section 3.5.2. K8s Control Plane is the interaction point between the database administrator(s) and the database server. This component is also responsible for monitoring and automatically controlling the different components of the cluster. Whenever a Pod fails, the Control Plane will take care of the autohealing process of the Pod. Whenever the database administrator requests to scale in or out the cluster, the Control Plane will handle the whole process by adding or removing Pod(s) and scheduling them on the right worker node(s).

Right below the K8s Control Plane, we observe the **Citus cluster** component. This component includes the **Citus coordinator(s)** and the **Citus workers**. As described in Section 3.3.1, Citus coordinator is the component that the database clients interact with and it also manages the distribution of the tables and queries over the worker nodes. The coordinator is exposed to the clients via a **service**. It is deployed on K8s using a

Deployment and a **Persistent Volume Claim** along with a **Persistent Volume** are attached to it. At this point, we remind that Docker containers are stateless, which means that once destroyed, no data is kept. As databases are stateful applications, we need the existence of Persistent Volumes in order to save the database data. In this way, even if the Pod that hosts the coordinator is destroyed, the data is still stored on the disk, and the new Pod that will replace the failed one, will be linked with this Persistent Volume. Hence, the database will continue operating with no downtime or any data failure.

Next to the Citus coordinator, we observe N **Citus workers**. These components store locally the fragments of the distributed tables and undertake the delivery of the data to the coordinator. Thanks to the **StatefulSet**, their size can dynamically grow or shrink, depending on the commands of the Control Plane. Citus workers are deployed as **DeamonSet**, which is a way to manipulate the placement of the workers to the cluster nodes. In a previous chapter we highlighted some of the key assets of Citus, one of them being the performance gained by a scalable database. To fully exploit such a capability, we want to ensure that each node of the cluster will host **exactly one** Citus worker container. With such an option, we are sure that all the resources of the nodes (CPU and RAM) are devoted to serve the Citus requests. These constraints are imposed by the DeamonSet. Finally, each Citus worker has attached a Persistent Volume and a Persistent Volume Claim to store the data.

4.2.2 K8s Configuration

Although Kubernetes facilitates the management of many Docker containers deployed on a cluster, its configuration is not trivial, especially when someone wants to create a container that consists of several components. For a beginner user, it is vital to follow the official tutorials of Kubernetes [Kubd] that explain the basics of the framework and familiarize the user with its API. Severalnines presents in another tutorial [Sev] how PostgreSQL can be deployed on Kubernetes. This tutorial is also important for someone who wants to understand how MobilityDB is deployed on Kubernetes.

The aforementioned tutorials provide enough insights to understand how Kubernetes works but they are not enough to deploy MobilityDB with Citus. The main reason is that Kubernetes uses Docker containers, on which all the components are deployed by using pre-generated images, but such images **are not configured**. Docker images introduce the meaning of *components decoupling* by separating the image from its configuration. This design choice makes the Docker containers more flexible, as the images are meant to be consumed by different users for different purposes. Consequently, each user is going to configure the deployment in a different way. In this work, we are using a Docker image of MobilityDB and Citus,[7] the base of which is PostgreSQL.

[7] Docker image `https://hub.docker.com/repository/docker/dimitris007/mobilitydb`

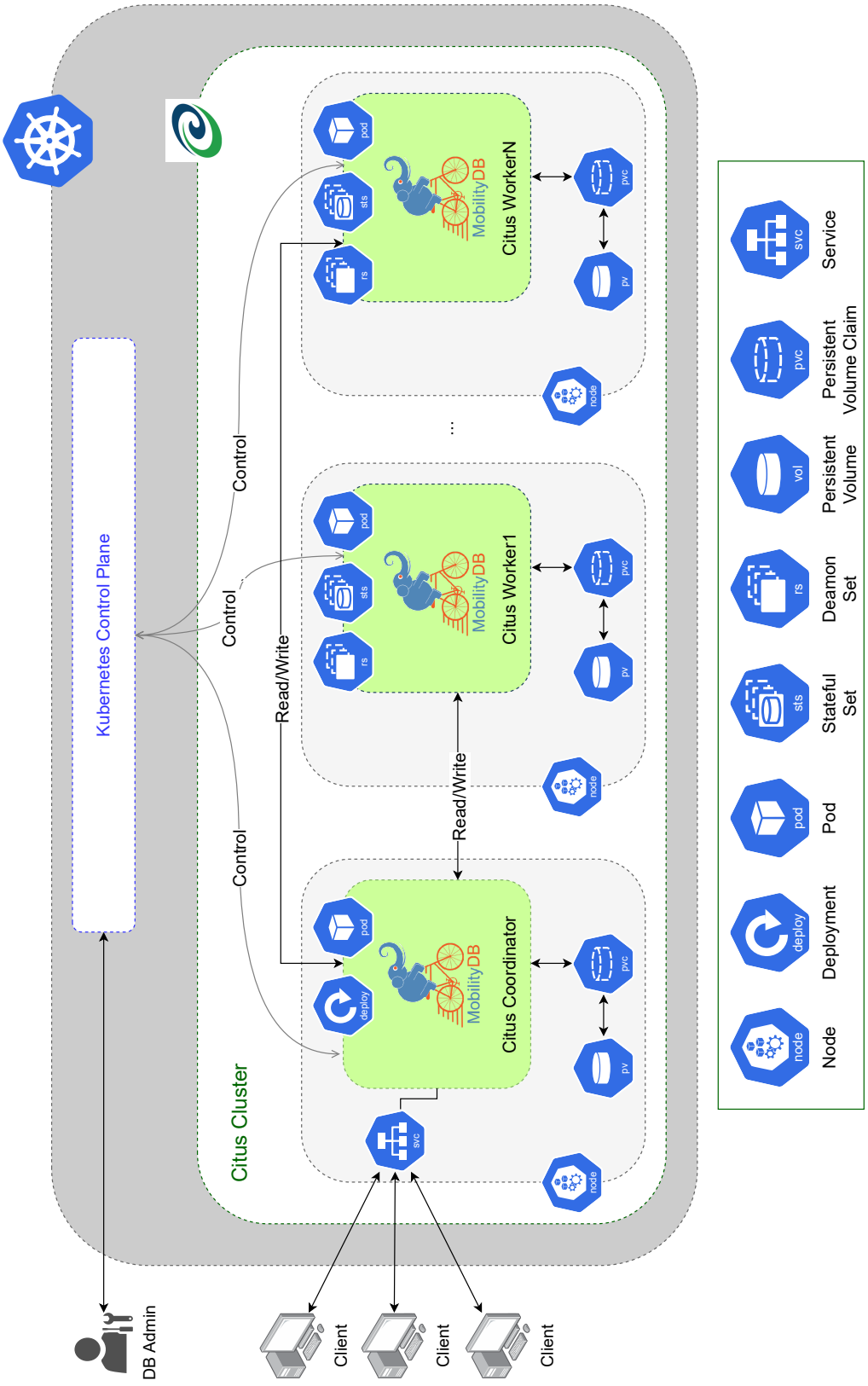


Figure 4.2: Diagram of MobilityDB deployment on K8s

K8s configuration is based on a declarative language consisted of several *yaml* files. The first file that we describe is **postgres-deployment.yaml**. Listings 10 and 11 show the content of the file.

- **spec.template.spec.containers.image** defines the Docker image which will be used to initialize the container. The image is pulled from the defined Docker Hub repository.
- **spec.template.spec.containers.args** is used to pass the desired arguments during the initialization of the container. In our case, we need to pass the configuration parameters for the postgres initialization.
- **spec.template.spec.containers.lifecycle.postStart.exec.command** describes the commands that will be executed right after the initialization of the container. They are used to create the required extensions to the target database, namely Citus and MobilityDB as well as to create `sessions_log` table and a pg_cron job, used for the autoscaler mechanism that is presented at a later chapter.
- **spec.template.spec.containers.env** is used to initialize the postgres environement (database user, password etc.) from the defined secret file.
- **spec.template.spec.securityContext** defines the privileges of the user with who the container is going to be created. 0 corresponds to the root user while 999 to the postgres user.
- **spec.template.spec.volumes** and **spec.template.spec.containers.volumeMounts** are parameters that define the volumes that will be mounted on the containers. In our case, we are mounting one volume that will store the database data and another volume for the secret files. The latter are files used to run Secure Sockets Layer (SSL) protocols on every node in order to enable the proper communication between the Citus nodes and also hide sensitive database information, such as administration username and password.

Another important file that we are using to properly configure the container is **postgres-secrets-params.yaml** (Listing 15). Kubernetes secrets enable the user to store and manage confidential information by using secret objects. These objects stores the desired information as unencrypted base64-encoded strings which means that as soon as the plain-text is retrieved, it can be decoded to the original format. When using secrets, it is highly recommenced to encrypt the encoded data. Once the secret object is defined and mounted on the container, its entries can be accessed by the Pod(s). In our deployment, **postgres-secrets-params.yaml** contains the following entries:

- **data.username** corresponds to the administrator username.
- **data.password** is used to access the database as administrator.
- **data.db** is the name of the default database. It is recommended to leave the default value (`postgres`) as the default database.

The main structure of **postgres-deployment-workers.yaml** file is similar to the one of **postgres-deployment.yaml**. Listing 12, Listing 13 and Listing 14 show the content of the file. One main difference is that the type of the defined object is StatefulSet which introduces the following supplementary entries:

- **spec.replicas** is used to define the initial number of deployed copies. As we already discussed, Citus architecture proposes one master node and several worker nodes, and hence, in this work, the StatefulSet is responsible to host and manage the Citus

worker nodes. There will be created as many worker nodes as we define in this entry.

- **spec.template.spec.containers.lifecycle.postStart.exec.command** describes the commands that will be executed right after the initialization of the container. Similartly to the citus-master deployment, MobilityDB and Citus extensions are created. Moreover, pg_hba.conf is correctly configured in order to allow connections from the master to the worker nodes and the postgres server restarts to apply the configuration changes. Finally, in order to add the worker nodes to the Citus cluster, the corresponding SQL command is executed on the master node via a request, done from the worker node.
- **spec.selector** and **spec.template.spec.affinity** are used to apply the logic behind *DaemonSets*[8] and *Affinity*.[9] In our architecture, we want to ensure that every node of the cluster runs **exactly one** instance of MobilityDB and hence, it uses all its resources to serve the requests of the Citus master node. In Kubernetes, it is possible to enforce the system to run only one Pod per node, by specifying the entry **spec.template.spec.affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution**. In this way, we achieve to have a StatefulSet deployed as DaemonSets. Supposing that with the given configuration, we have a cluster with five nodes and we try to scale the StatefulSet to six replicas. The result will be five **running** replicas on five **different** nodes and one unscheduled replica with **status = pending**.

Two different *Persistent Volumes* and *Persistent Volume Claims* are defined by two different *yaml* files, namely **postgres-storage-coordinator.yaml** and **postgres-storage-worker.yaml**. Both files define exactly the same configuration to mount the desired disk space on the Docker containers, with the only difference that the disks are mounted on different paths. Such a design allows to have one coordinator and one worker node on the same VM, without mixing their postgres directories. If both disks were mounted on the same directory (e.g. /mnt/data) and the filesystem of the VM was used as the actual disk space, then both postgres file systems would be stored under the same path and the intantiation of the databases would not be possible. Below there are listed the entries of **postgres-storage-coordinator.yaml**:

- **spec.capacity.storage** determines the total size of the disk storage that the Pod can use.
- **spec.accessModes** specifies how the storage can be accessed from the several Pods. In our deployment, the option is set to *ReadWriteMany*, as many Pods should be able to simultaneously read and write data.
- **spec.hostPath.path** is used to determine the path of the VM where the disk will be mounted.

Apart from the above entries, the file also defines the corresponding *Persistent Volume Claim*. Listing 16 shows the complete content of the file.

In this deployment, the **Persistent Volumes are local disks**, meaning that the allocated disk is part of the VM disk. In general, it is preferred to use external disks to store the data of an application deployed on K8s, however, this is not possible using Microsoft Azure Files, which is the way that K8s recommends as external storage. The reason is that

[8]K8S DeamonSet https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/
[9]K8S Affinity https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

currently Azure Files do not support the structure of a PostgreSQL server's data directory.

In this section, we described the K8s architecture and the manifests that we use in this work in order to enable the automatic management of a containerized MobilityDB and Citus cluster on Azure. In the following chapter, we introduce the meaning of autoscalers for our database. More specifically, we provide a list of available metrics that can be used to model the state of the system as well as technical implementation details of the developed autoscaler.

Chapter 5

Workload-based Automatic Provisioning

This work targets to provide a Database-as-a-Service of MobilityDB on the cloud, also called Azure Hyperscale MobilityDB, that guarantees **elasticity** and **high availability**. In the next sections, we describe how these two features are achieved with our implementation and we analyze several technical and design choices that were made. First, we enumerate the different available metrics that a user can use to determine the current state of the database server. Second, we explain which of these metrics are used in this work in order to create different autoscalers that transfuse elasticity to our database. Then, we explain how the high availability feature is achieved out of the box when using Citus.

5.1 Metrics Collection

This section provides a detailed list of different metrics that can be used to assess the performance of a PostgreSQL database, deployed on Azure.

Elasticity is the property of a system to adapt itself to the environment changes, by scaling in such a way that its future state will efficiently target the user requests. The decision of when the system should scale is driven by the collection and analysis of metrics that outline its current state. There are plenty of metrics and statistics that PostgreSQL internally collects, measuring how the database server performs. In addition, there are also external projects that contribute to statistics collection and visualisation. PostgreSQL wiki maintains a monitoring section [Cona] with a complete list of PostgreSQL metrics. Pg_stats.dev [Les21] is an interactive visualisation tool that presents the internals, functions and statistics of PostgreSQL as well as relevant information about specific database subsystems. Table 5.1 summarizes the available PostgreSQL metrics related to the current work, a short description of each metric and available sources that can be used to extract them.

In the next section, we describe how some of these metrics are used in order to create a number of different autoscaling mechanisms, using the formal definition of the MAPE loop of Section 2.3.

Metric	Source	Description
Infrastructure Level Metrics		
Percentage CPU	Azure Monitor	The percentage of allocated compute units that are currently in use by the virtual machine(s).
Disk read/write bytes per sec	Azure Monitor	Average bytes read/write from/to disk during monitoring period.
Disk read/write operations per sec	Azure Monitor	Disk read/write IOPS.
Network in/out	Azure Monitor	The number of bytes received/sent on all network interfaces by the virtual machine.
Application Level Metrics		
Database information	PostgreSQL Statistics, pgmetrics	Aggregated information for each database of the server. A sample of the collected statistics: # of transactions in this database that have been committed/rolled back, # of disk blocks read in this database, # of cache hit, # of rows returned/fetched/inserted/updated/deleted by queries in this database etc. This view illustrates the accumulated values since the last reset. Similar information can be extracted using *pgmetrics* tool.
Table general information	PostgreSQL Statistics, pgstattuple, pgmetrics	Aggregated information for each system and user table of the current database. A sample of the collected statistics: # of sequential scans initiated on this table, # of live rows fetched by sequential scans, # of index scans, # of live rows fetched by index scans, # of rows inserted/updated/deleted etc. By using *pgstattuple* tool, more detailed information can be extracted, like # of live/dead tuples, percentage of live/dead tuples etc. *Pgmetrics* tool can also be used to easily extract general table statistics.

Table input/output information	PostgreSQL Statistics	Aggregated I/O information for each system and user table of the current database. A sample of the collected statistics: # of disk blocks read from this table, # of buffer hits in this table, # of disk blocks read from all indexes on this table, # of buffer hits in all indexes on this table etc.
Index general information	PostgreSQL Statistics, pgstattuple, pgmetrics	Aggregated information for each system and user index of the current database. The collected statistics are: # of index scans initiated on this index, # of index entries returned by scans on this index and # of live table rows fetched by simple index scans using this index. By using *pgstattuple* tool, more detailed information can be extracted, depending on the type of the index. For instance, for a B+ Tree index, the # of leaf pages, the average density of leaf pages etc. are available.
Index input/output information	PostgreSQL Statistics	Aggregated I/O information for each system and user index of the current database. The collected statistics are: # of disk blocks read from this index and # of buffer hits in this index.
Function information	PostgreSQL Statistics	Aggregated information for each tracked function, showing statistics about executions of that function. The collected statistics are: # of times this function has been called, total time spent in this function and all other functions called by it and total time spent in this function itself.
User sessions activity	PostgreSQL Statistics	The average number of active users per minute.
Query execution time	pg_stat_statements extension, pgmetrics	Average execution time per query class. Moreover, *pgmetrics* tool provides useful information (# of calls, average time, total time) related to the most expensive queries run on each database

Table 5.1: Metrics collection

5.2 Building an Elastic Database

This section details how the elastic feature is implemented in this work with the assist of autoscalers. Specifically, we describe all the implementations related to the cluster operations, namely scale in and out, and the design choices related to the construction of the autoscalers.

The autoscaling mechanism takes cares of the elastic nature of the database. This module is responsible to *monitor* the system, *analyze* the observed data in order to *plan* for the future operations that need to be *executed* to adapt the system to the current workload. The available operations are *scale out* and *scale in*, where in the first case the system dynamically allocates more VMs to respond to higher workload while in the second case the system dynamically deallocates the unnecessary VMs. The implementation of such a mechanism requires a process that runs in a continuous loop which is also known as *daemon* process. For this purpose, the Daemons Python library is used[1] that provides all the functionality of a regular daemon, namely daemonization, signal handling, process id (pid) management functionality while allowing for any implementation of behaviour and logic. The autoscaler is implemented in Python 3 programming language and deployed with Python virtual environments to enable quick portability.

Python is an object oriented programming language which makes the definitions of classes vital for the developers. Different classes correspond to different conceptual elements of the program. As our autoscaler needs to manage different concepts, namely a daemon, a Kubernetes cluster, a Citus cluster and Azure resources, four different independent classes are defined. Figure 5.1 depicts the class hierarchy. **AutoscalingDaemon** class is mainly used to implement the MAPE loop as it was described in Section 2.3. **K8sCluster**, **CitusCluster**, **Azure** and **Monitor** classes are used to manage the Kubernetes cluster that hosts the deployment of MobilityDB on Azure. The next subsections further analyze the functionalities of those classes.

5.2.1 Cluster Operations

To turn the database server into a scalable cluster, the following operations need to be defined:

- **Scale out operation** adds the specified worker nodes to the cluster. This operation is handled by **K8sCluster.cluster_scale_out** method and the process is the following. Firstly, the names of the deployed VMs are listed by using the Azure SDK library[2] for Python. In our deployment, the names of the VMs follow the naming convention *Worker1, Worker2, Worker3...* which makes necessary to list the current VMs' names whenever we want to add new VMs. After having defined the names of the new VMs, we reuse a similar bash script to the one that we used during the initializing of the cluster (Figure 4.1) to allocate and deploy the new resources. The difference is that we do not need to recreate the coordinator node so we skip the first steps of the process. After the successful creation of the new VMs, we rebalance the

[1] Python Daemons library https://pypi.org/project/daemons/

[2] Azure SDK for Python https://docs.microsoft.com/en-us/azure/developer/python/azure-sdk-overview

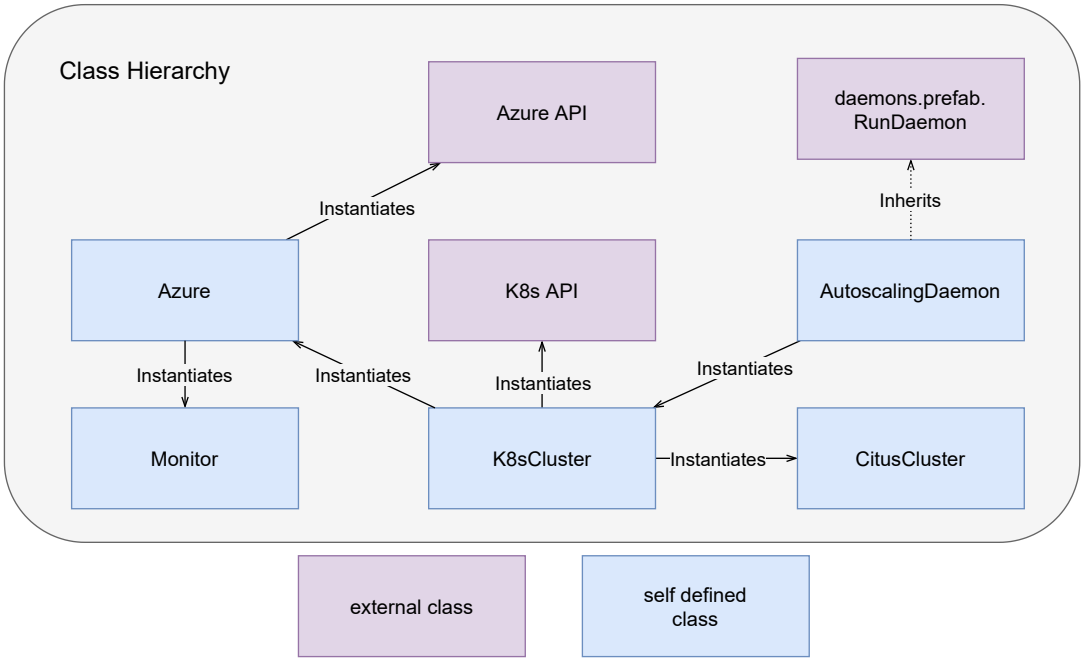


Figure 5.1: Class hierarchy

data across the new nodes by using Citus shards rebalancing mechanism.[3]

- **Scale in operation** removes the specified worker nodes from the cluster. This operation is handled by **K8sCluster.cluster_scale_in** method and the process is the following. The first step is to define the name of the VMs to be deleted, starting from the one with the higher identifier (e.g. for a cluster with nodes *Worker1, Worker2, Worker3*, the scaling in operation of one node will remove *Worker3*). To correctly remove a node from the Kubernetes cluster that hosts MobilityDB with Citus, the following actions need to be performed at different levels:
  1. Firstly, the worker node(s) should be removed from the **Citus cluster**. As previously explained, Citus uses data sharding to horizontally distribute the tables within the cluster. Whenever we want to delete a node, the shards placed on this node need to be transferred to the remaining nodes of the cluster. This process is under the responsibility of Citus and can be triggered with simple SQL queries. Listing 17 illustrates the Citus deleting function.
  2. Furthermore, the next step is to unregister the worker node(s) from the **Kubernetes cluster**. The *StatefulSet* needs to be accordingly scaled and the node should be *drained* and *deleted* from the cluster. Listing 18 depicts the Kubernetes deleting function.
  3. Finally, the unused VMs and the attached resources should be deleted from **Azure**. Listing 19 presents how the resources are deallocated.

[3]Citus rebalancing mechanism http://docs.citusdata.com/en/v10.0/admin_guide/cluster_management.html#shard-rebalancing

5.2.2 Autoscaler Implementation

As we understand by reading the literature of the different autoscaling techniques that have been developed the last years, there are different levels from where metrics can be extracted to measure the state of a scalable system. One major source of information is the infrastructure provider (cloud provider) that provides several metrics for the allocated VMs. Azure provides a wide range of such metrics, like *CPU utilization, disk read/write bytes/operations, network in/out* and many more that are exposed by a REST API[4] which can be queried through Azure Python SDK. The user can define the granularity of the collected data. In our experiments, the data is collected **per VM and per minute**.

Another source of information that generates valuable statistics of the state of the system is the running application by itself. More specifically, PostgreSQL, which is the database on top of which MobilityDB has been developed, provides different metrics that can be used to determine the ideal size of the cluster. Such a metric is the *active user sessions*, as when more users are using the database server, more worker nodes should be added to the cluster to decrease the response time. PostgreSQL stores information related to the current activity of the server in `pg_stat_activity` table. Although the information of this table is quite powerful, it is not what we need for the monitoring purposes of the autoscaler, as the table is continuously updated and it does not store historical data. What we would like to have is a table that stores the **active user sessions per second**, for the last 30 minutes. Such an implementation can be done on the database level by using *pg_cron*,[5] a simple cron-based job scheduler for PostgreSQL that runs inside the database as an extension. As the *pg_cron* works at per minute level, another `do_every_second` PostgreSQL function is implemented to gather the required statistics every second and stores the result in `sessions_log` table. This table will be used during the analysis phase of the MAPE loop. Table 5.2 is a sample of the `sessions_log` table. The table can be queried by using the SQL query of Listing 8 to retrieve the change of active users during the last minutes. It retrieves the average number of active user sessions the last three minutes and the same number between the last three and eight minutes. Having this information (Table 5.3), we can easily compute the change of the active users for the desired time window (in this case a time window of five minutes).

	Time	User_number
1	2021-06-16 07:56:18.314256+00	19
2	2021-06-16 07:56:17.300236+00	19
3	2021-06-16 07:56:16.282292+00	16
4	2021-06-16 07:56:15.267853+00	13
5	2021-06-16 07:56:14.253031+00	10

Table 5.2: Sample output of `sessions_log` table

The next step to be done after the collection of the performance metrics, is to analyze the data and plan the next state of the system. In this work, this phase is performed with

[4] Azure metrics API https://docs.microsoft.com/en-us/rest/api/monitor/metrics/list
[5] pg_cron https://github.com/citusdata/pg_cron

```
SELECT 'NOW' AS period, CEIL(AVG(users_number))
FROM sessions_log
WHERE time > NOW() - interval '3 minutes'
UNION
SELECT 'BEFORE' AS period, CEIL(AVG(users_number))
FROM sessions_log
WHERE time > NOW() - interval '8 minutes' AND time < NOW() - interval '3
    minutes'
```

Listing 8: SQL query to retrieve the change of active users during the last minutes

	Period	Active users
1	BEFORE	12
2	NOW	14

Table 5.3: Sample output of the query in Listing 8

different rule-based techniques, the parameters of which are fine tuned according to the details of each use case (Chapter 6). Finally, the decisions that are planned are actually executed by using the cluster operations, defined in Section 5.2.1.

In this section we presented the implementation details of the autoscalers that are developed within this work. Specifically, we detailed the different metrics that were used to assess the performance of the system and we analyzed the implemented cluster operations, needed during the execute phase of the autoscaler. In the next section, we explain how the deployed solution guarantees high availability to the users.

5.3 Guaranteeing High Availability

This section explains how the deployed MobilityDB cluster is continuously available to the users with almost zero downtime.

As we understand until now, deploying and maintaining a MobilityDB and Citus cluster on the cloud includes several difficulties. One major issue that cloud native solutions should deal with is failure of different components. If a node fails, it will interrupt the normal operation of the application, unless a mechanism takes care of this issue. Our deployment consists of several worker nodes, each one of them storing fragments of the distributed tables. If one or more node fail, Citus coordinator will not be able to answer the SQL queries, as part of the distributed tables will be missing. In this work, we mainly propose two solutions to ensure high availability. First, we leverage K8s autohealing feature which is responsible for the normal operation of the Pods that hosts the Citus workers. Whenever K8s controller detects a failed Pod, it periodically tries to restart it to allow the normal operation of the cluster. However, if for some reason the node that hosts the failed Pod experiences some downtime, K8s controller will not manage to restore the Pod. For such cases, we can leverage Citus replication mechanism by accordingly ad-

justing `citus.shard_replication_factor` variable to a number greater than 1.[6] In this way, we enforce Citus coordinator to **replicate each shard of the distributed tables to more than one worker nodes**. As a result, if some nodes fail, Citus coordinator will still be able to compute the SQL query results, as the shards of the failed workers can be found on different nodes.

This section described how our work deals with node failures and ensures high availability. In the next chapter, we present the experiments that we performed to assess the performance of the proposed autoscalers using a synthetic dataset produced by BerlinMOD benchmark. In addition, Scalar benchmark is used to simulate the behaviour of multiple users that concurrently query the target database server.

[6] Citus replication http://docs.citusdata.com/en/v10.0/admin_guide/cluster_management.html#worker-node-failures

Chapter 6

Experiments and Results

This chapter explores the capabilities of different autoscaling mechanisms on a distributed, self-scalable version of MobilityDB on Azure. First, the used dataset and benchmarks are presented to sketch the contour of our experiments. Second, we explain how the data is distributed using Citus extension, by providing an appropriate example. Subsequently, we present and assess several measurements of the performance of different autoscalers, performed on the aforementioned dataset.

6.1 Dataset and Benchmarking

This section presents the dataset and benchmarks used in our experiments. First, it details BerlinMOD benchmark, a moving object benchmark that simulates the behaviour of people that move around the city of Berlin. Second, K8-Scalar benchmark is presented, which is used to simulate the workload generated by users that query the database.

6.1.1 BerlinMOD Benchmark

BerlinMOD [DBG09] is a benchmark for moving object data (MOD), targeting scalable spatio-temporal DBMS. This benchmark consists a simulation where vehicles are following different trajectories over the network of streets in Berlin. The defined trajectories simulate the actual behavior of drivers going to and from the work during the day as well as leisure trips during the evening and the weekend. The benchmark provides a data generator that uses SECONDO [GBD10] for generating the trips of the vehicles as well as a set of 17 BerlinMOD/R queries as workload. In addition, the benchmark is extensible in such a way that users can generate data for other cities and adapt the data and the workload on other spatio-temporal DBMS. The generator receives as parameter the scale factor of the produced dataset which determines the number of trips that will be generated.

[Moba] is a variation of the BerlinMOD benchmark, applied on the data types of MobilityDB. The produced data simulates the same scenario in the city of Brussels. The generated schema is as follows:

```
Cars (CarId integer, Licence varchar(32), Type varchar(32), Model
varchar(32))
Instants (InstantId integer, Instant timestamptz)
Periods (PeriodId integer, Period period)
Points (PointId integer, Geom Geometry(Point))
Regions (RegionId integer, Geom Geometry(Polygon))
Licences (LicenceId integer, Licence varchar(8), CarId integer)
Trips (CarId integer NOT NULL, TripId integer NOT NULL, TripDate date,
```

```
source integer, target integer, Trip tgeompoint)
```

From the schema above, we see that the database schema consists of seven tables. `Trips` is the table that contains the biggest volume of data. Every other table has relatively small size, which varies from some few hundreds to thousands of rows, even for big scale factors (> 5). Apart from the traditional PostgreSQL data types, there are several types that are defined by either PostGIS extension or MobilityDB. Below, we explain the functionality of these data types.

1. PostGIS provides the `geometry` data type to represent real world objects that exist in space. For instance, by using the type `Geometry(Point)` the user defines a geometry that consists of several points. PostGIS also defines a set of intuitive functions to manipulate these data types. For example, considering that we define two geometries, namely `g1 Geometry(Point)` and `g2 Geometry(Point)`, we can use the function `ST_Intersects(g1, g2)` to determine whether the two geometries intersect.
2. `Period` is a time type that exists in MobilityDB. It is used to express a **continuous** range of instants in time. It is constructed by two `timestamptz` values, where the first one represents the beginning and the other one the end of the `period`. `Period` implies linear interpolation between the consecutive time instants.
3. `Tgeompoint` is one of the temporal types of MobilityDB and it is based on the continuous base type `geometry` of PostGIS. By using this data type, we can express how an object moves in space over time. In our database, this field is used to represent a trip of a vehicle. This means that each line of the `Trips` table represents one complete trip, and not just a single point of it.

The following example helps us better understand the possible operations in MobilityDB. Query14 of BerlinMOD benchmark, presented in Listing 9, answers the following question: *Which vehicles travelled within one of the regions from Regions1 at one of the instants from Instants1?*
`Regions1` and `Instants1` tables are views of `Regions` and `Instants` tables, respectively. To answer the query, we need to perform a join operation between `Trips, Regions1` and `Instants1`, where:

- We first compute a bounding box using `stbox` operator between `R.geom, I.Instant`. Then the query performs a bounding box comparison with the `&&` operator between the computed box and `T.Trip`.
- Regarding the join operation with `Instants1` table, we first compute the `valueAtTimestamp` of `T.Trip, I.Instant`. We remind that `T.Trip` engages both time and space dimensions, hence, we can extract the geometry of the trip at a given moment. The result, which is of type `Geometry`, along with `R.geom`, are passed as arguments to `_ST_Contains` function that returns true if and only if no points of geometry B lie in the exterior of geometry A.
- Finally, the result of the first query is stored into a CTE, which is then joined with `Cars` table to return also the `C.Licence` column to the result set.

```
WITH Temp AS (
   SELECT DISTINCT R.RegionId, I.InstantId, I.Instant, T.CarId
   FROM Trips T, Regions1 R, Instants1 I
   WHERE T.Trip && stbox(R.geom, I.Instant)
   AND _ST_Contains(R.geom,valueAtTimestamp(T.Trip, I.Instant))
)
SELECT DISTINCT T.RegionId, T.InstantId, T.Instant, C.Licence
FROM Temp T JOIN Cars C ON T.CarId = C.CarId
ORDER BY T.RegionId, T.InstantId, C.Licence
```

Listing 9: Query14 of BerlinMod benchmark in MobilityDB

6.1.2 K8-Scalar Benchmark

K8-Scalar [DTR+18] is a workbench that can be used to compare different autoscaling mechanisms for container-orchestrated database clusters. An open source implementation of the benchmark, called Scalar, also exists [Hey14]. K8-Scalar is basically a simulation environment for scalable systems, as it allows to analyze the scalability and quality of service of the system under test. To do so, it simulates the behaviour of many users that try to simultaneously generate requests for the system. The benchmark is fully extensible, as it allows to define different user requests and hence, different user types. It also provides an extensive list of input parameters, where the user can tune the desired total time of the test, the rate of each user type and many other parameters that are related to the workload. After the end of the experiment, the benchmark reports meaningful information about the performance of the system, namely minimum, maximum and mean execution time per user request, number of total requests done as well as number of successful and failed requests.

In this work, we use Scalar benchmark to measure the effectiveness of the scaling mechanisms. Regarding the input parameters, we mainly focus on the number of concurrent users, on the total duration of the test and on the user request. In each of the following experiments, we give more details about the values of these parameters.

In this section we presented the dataset that we use in our experiments and the benchmarks that generate the synthetic workload for the target database. In the next section, we discuss how the produced data is distributed over the Citus cluster.

6.2 Distributing Properly the Data

Selecting the right distribution column is a major issue that determines the performance of fetching the data from the distributed tables when using Citus. The purpose of this section is to illustrate the significance of the distribution column selection by using an example, based on the BerlinMOD benchmark.

In this example, we are using a dataset of scale factor 0.2 which includes around 32000 trips (1.7GB). The tables of the database are the one described in Section 6.1.1. Every relation of the database is replicated to every worker machine, as they are relatively small

in size, except from `Trips` table that is distributed. The used distribution column is `tripId`.

By observing the top left part of Figure 6.1 we can clearly see that the **shards** of the distributed table **are not well balanced**, as machine 2 and 3 store significantly more data than machine 1 and 4. Data misbalance implies that the four machines are not equally used during the query execution as half of them are fully working to fetch data while the others are barely used. The bar chart on the top right of Figure 6.1 better explains the source of the problem. We can see that the dataset includes 16 distinct `tripIds`, where some trips have many thousands of rows while some others have only few decades. We remind that Citus distribution shards the table by using hashing to group all the tuples with the same distribution key to the same machine. Hence, all the tuples of `tripId = 1` (around 29% of the whole dataset) will end up in the same shard that will be quite big in size. Consequently, the chosen distribution column causes a major data misbalance, which results in poor performance and low efficiency.

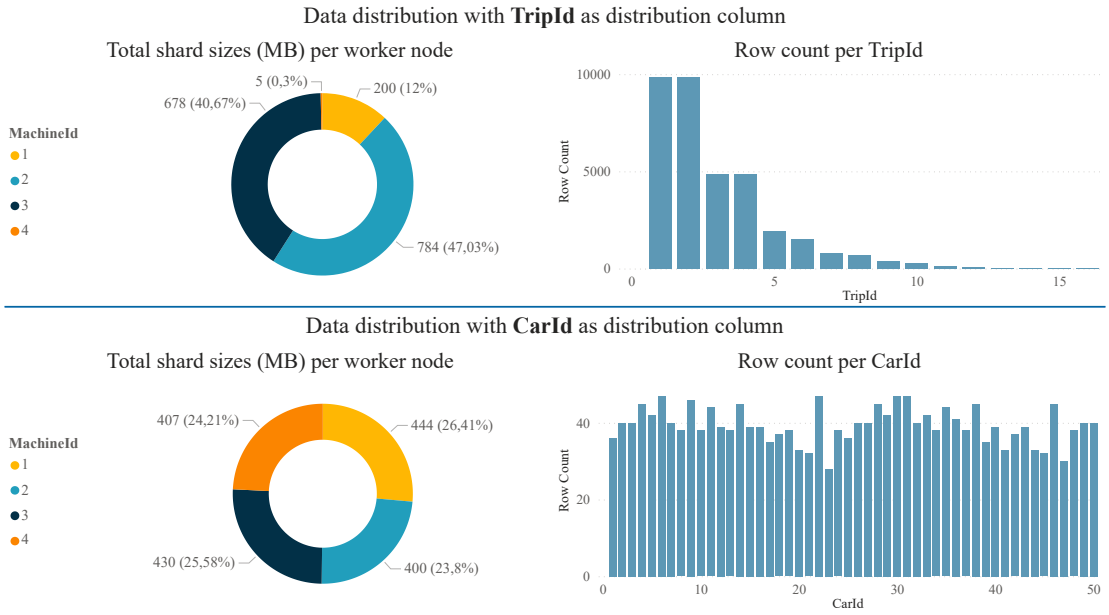


Figure 6.1: Data distribution using different distribution columns[1]

The bottom left part of Figure 6.1 shows the occupied size of disk by the shards in each machine when `carId` is used as distribution column. We clearly see that in this case, **the shards are better balanced** as each machine has approximately the same size of data. The bottom right part of the figure illustrates the distribution of the row count of the first 50 `carIds`. We can easily observe that each car has approximately the same number of rows, which is also valid for the remaining 850 `carIds` that are not included in this chart for space saving. The query performance and the efficiency is much higher with this setup.

Although there are several scenarios where we can use a distribution column that

[1]All the reports of this chapter have been designed using MS Power BI reporting tool

evenly distributes the data over the cluster, there are cases where such columns do not exist. For instance, if we had a `Trips` table containing different amount of trips for each vehicle, choosing `cardId` as distribution column would not solve the aforementioned problem. Luckily for such scenarios Citus provides different **distribution strategies**. By default, Citus uses the simple `by_shard_count` strategy, which distributes the shards by taking into account only the number of the shards. This means that if the cluster has 16 shards and four machines, each machine will host four shards, without considering the size of them. Another available strategy, more suitable for this scenario, is `by_disk_size`. In this case, the shards are distributed according to the total occupied disk space of each machine, meaning that the coordinator will try to evenly distribute the data based on the shards size. Finally, users are able to define their own distribution strategies, according to Citus documentation [Mice].

In this section we discussed the significance of selecting the right distribution column when using Citus. The next section presents the experiments that we performed in this work, by using different autoscaling mechanisms on different scenarios.

6.3 Autoscaling Experiments

This section presents the performance of different autoscalers that use the collected metrics described in Section 5.2.2. Furthermore, it is examined to what extent such autoscalers can be applied on real use cases as well as the existence of different tuning parameters that make them adaptable to various scenarios. Finally, some limitations, that the used tools impose, are discussed.

6.3.1 CPU Utilization Autoscaler

The first experiment consists of a simple **rule-based autoscaling mechanism** that uses the average CPU utilization percentage metric, collected at the infrastructure level. More specifically, the policy depicted in Figure 6.2 is applied. The used benchmark is a combination of the BerlinMod and Scalar benchmarks. The first one is used to generate synthetic workload for the system while the second one is used to simulate the requests produced by users. The parameters of both benchmarks are summarized in Table 6.1.

Parameter Name	**Parameter Value**
BerlinMod Benchmark Parameters	
Scale Factor	0.5
Total Size (GB)	4
Scalar Benchmark Parameters	
Concurrent Users	15
Total Duration (hours)	2
User Request	BerlinMod Query14
VM Parameters	
# of CPU(s)	2
RAM (GB)	4

Disk Size (GB)	60
Azure VM Name	B2s
Disk Type	Premium SSD

Table 6.1: Experiment parameters

Although in the literature, most of the cloud providers use the same performance metric for the decision making of the autoscaler, it seems that **when it comes to database scenarios and especially MobilityDB, CPU utilization is not the ideal metric**. By using only the CPU utilization as the autoscaling metric, we face a major problem that is closely related to the nature of the autoscaler. Figure 6.3 highlights the issue. It basically shows the measured CPU utilization of the four VMs that participate in the experiment. The blue and the red lines correspond to the VMs that are active since the beginning of the test. The dark and the light blue lines plot the CPU utilization of the newly added VMs. As we see, the first two VMs are fully working during the first 30 minutes of the test, while the utilization of the other two progressively increases. We observe that the measured CPU utilization **remains high even after the scale out operation**, which causes a critical problem, as the autoscaler will scale out again. As we have already explained, the autoscaler is basically a continuous loop that monitors the state of the system (performance metrics) and acts accordingly. In this experiment, **the state of the system** after the scaling out operation **will not change**, because the average CPU utilization percentage will still be more than 80%. This observation makes us conclude that **the CPU utilization metric by it-self is not the ideal metric for use cases where the VMs are intensively operating during the whole duration of the test**.

6.3.2 Database Activity Autoscaler

This subsection is devoted to illustrate the use and the performance gain of a more sophisticated **rule-based autoscaling mechanism**, applied on a MobilityDB cluster. To evaluate the first autoscaling mechanism, we run twice the same benchmark, once without applying any scaling and once by using the rules depicted in Figure 6.4. The change of the user sessions is calculated using Table 5.3. The used benchmark is a combination of the BerlinMOD and K8-Scalar benchmarks. The first one is used to generate synthetic workload for the system while the second one is used to simulate the requests produced by users. The parameters of both benchmarks are summarized in Table 6.1.

Fig. 6.5 depicts the measured performance in both experiments. As we see from the bar chart, the mean execution time drops from 57 seconds in the experiment without autoscaling (experiment 1) to 42 seconds when the pre-defined autoscaling mechanism (experiment 2) is applied. Although it is a significant performance gain, it would be rational to wonder why the speedup is not close to 50%, taking into account that the size of the VMs is doubled (from two to four after scaling out). Observing the minimum measured execution time bars, the minimum execution of the first experiment is two times greater than the same time of the second experiment. This measurement is right, as when running the workload of the Scalar benchmark (BerlinMod Query14) alone on the same

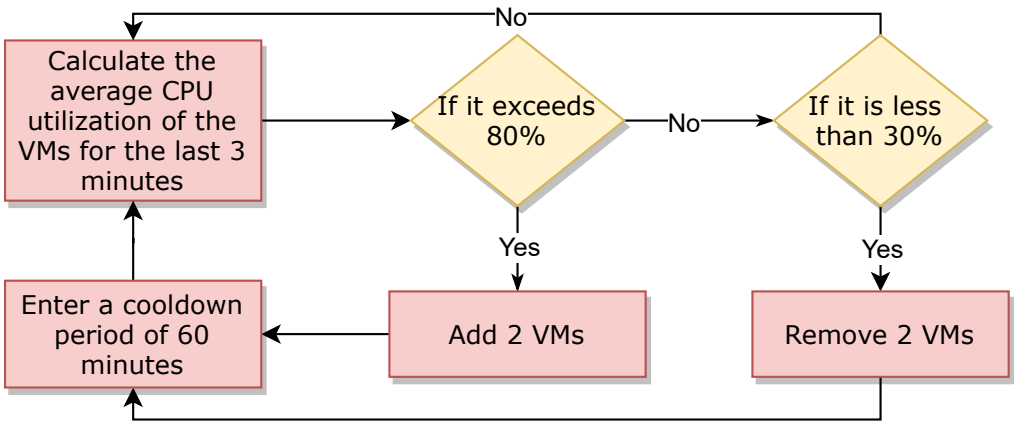


Figure 6.2: Flowchart of CPU utilization autoscaler

(idle) database server, we get exactly the same execution time, namely two and four seconds, respectively. The mean execution time of the experiments is much greater than this number as during the tests, the database is overwhelmed by many concurrent user requests (up to 15) and hence, it is quite overloaded. This observation leads us to conclude that while the performance gain of the minimum execution time is close to the expected one (50%), the same statement is not valid for the whole duration of the experiment, which requires further investigation. Another interesting observation from the rings of Figure 6.5 is that during experiment 1, the database was able to handle all the requests but in experiment 2, around 6% of the total user requests failed. Although some requests fail, the database is able to serve 28% more requests compared to experiment 1, as the mean execution time is less and hence, there is more time to serve user requests during the two hours of the test.

6.3.2.1 Analyzing in Depth the Performance

To address the previous questions, we need to observe the time series in Figure 6.6, illustrating the CPU utilization of the VMs participating in the experiment over the whole duration. The visualization has been exported from Azure portal and it has been edited in such a way that the vertical dashed lines show some of the actions that were taken during the execution (e.g. the time span when the VMs were created).

To better understand the impact of the autoscaling mechanism on the experiment, we need to first explain the activities that take place at each interval. As we see from the graph, the experiment lasts two hours. Starting at 3pm, only two VMs are operating, as none autoscaling mechanism has been applied so far. Around 3:05pm, the user activity increase exceeds the upper threshold (50%) and the system scales out by adding two more VMs. After seven minutes, both VMs have been deployed on Azure and joined the Kubernetes cluster. This means that by this time, the newly deployed VMs start contributing to the query answering of the test. **However, the moment that the new VMs join the cluster, they do not contain any data of the distributed table**. The operation of rebalancing the shards of this table starts around 3:10pm (3rd dashed line) and terminates at 4:20pm. Such a process is quite long, as some few GB of data are

Figure 6.3: CPU utilization of the worker machines during the first minutes of the test

transmitted over the network and at the same time the database serves the concurrent user requests of the Scalar experiment. One nice feature of Citus is that during all this time, the new VMs perform two actions: **continuously receiving table shards** until the shards are equally distributed over the nodes and **partially answering the user requests.** In other words, during this one hour of data exchange, the new machines do not stay idle, but they contribute normally to the query execution. As they progressively accumulate more and more shards, their contribution becomes bigger over the time. The rebalancing mechanism of Citus explains our first observation related to the total speedup of the second experiment. We understand that this speedup cannot be close to the optimal (50%) **as the data is not perfectly distributed over the cluster since the beginning of the experiment**. The Citus cluster needs around one hour to perform this operation, which amounts to half of the total time of the experiment. If we perform the same experiment once more for more that two hours, the speedup will increase, as the query answering is going to fully exploit all the VMs for more time.

Moreover, the fact that on the second experiment, the database could not serve around 6% of the total requests, happens due to the same rebalancing system. More specifically, the Citus coordinator keeps track of the shards placement at each moment and accordingly creates the distributed query execution plan. While rebalancing the shards, there are some gaps between the real shards placement and the one that Citus coordinator believes, as the update of it takes some seconds. During this short period, the incoming queries receive an error as answer, as Citus coordinator tries to locate the shard on the old node, but the

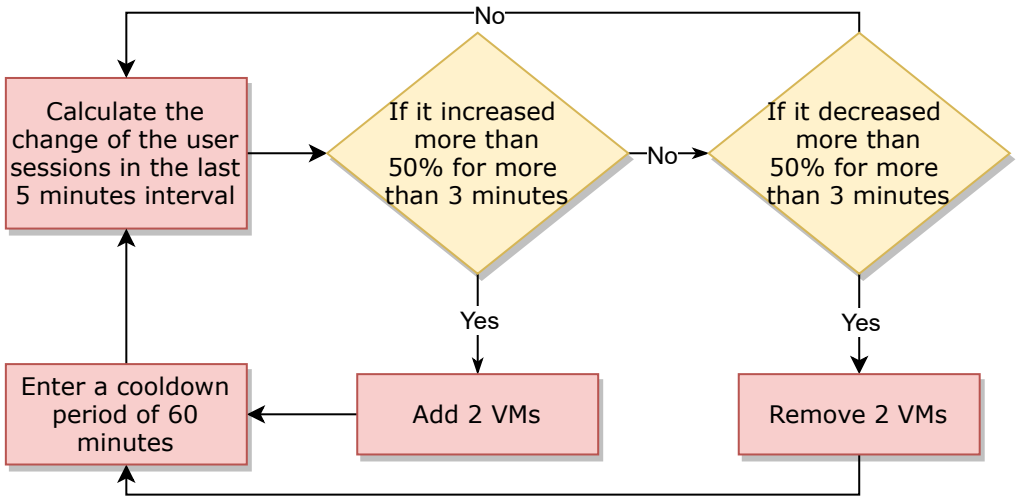


Figure 6.4: Flowchart of database activity autoscaler

shard was just sent to a new node. If we re-execute the same query after some seconds, the coordinator is updated and the query will be normally answered.

The CPU utilization diagram in Fig. 6.6 can also give us more insights about the implementation of the rebalancing mechanism of Citus. More specifically, if we notice the behaviour of worker 3 and 4 during their first minutes of liveness, we can observe a continuous workload transfer between the two machines. This means that for some minutes worker 3 works more intensively than 4, right after the utilization of 3 drops and 4 starts working, then 4 stops again and 3 boosts etcetera. Such behaviour is explained by the **progressive** movement of shards between the old and the new nodes, which is done with only one active thread. According to our research and after close contact with Citus engineering team, Citus rebalancing mechanism has not been optimized for speed, as it is designed to be minimally intrusive since users typically run it when their database is already very busy. **Such a design choice is a major disadvantage for a self-scalable distributed database** because the faster the rebalancing is, the higher performance of the system will be. The process can be further optimized by parallelizing the transmission of shards between nodes, whenever it is possible.

6.3.2.2 Getting more Insights

It is clear from our second experiment that the use of the active database users as a metric for the autoscaling mechanism is more useful compared to the CPU utilization of the first experiment (Section 6.3.1). In addition, we realize that the Citus rebalancing mechanism needs to be further explored with bigger datasets to determine whether the current implementation is suitable for a self-scalable database.

Another interesting observation is related to the parameters of the autoscaler. Fine tuning of these parameters is vital and can significantly increase the performance. One parameter is the threshold that the user defines. In this autoscaler, the threshold defines the minimum increase or decrease of users that the autoscaler will scale out or in, respectively. This parameter needs to be adjusted according to the expected traffic of the

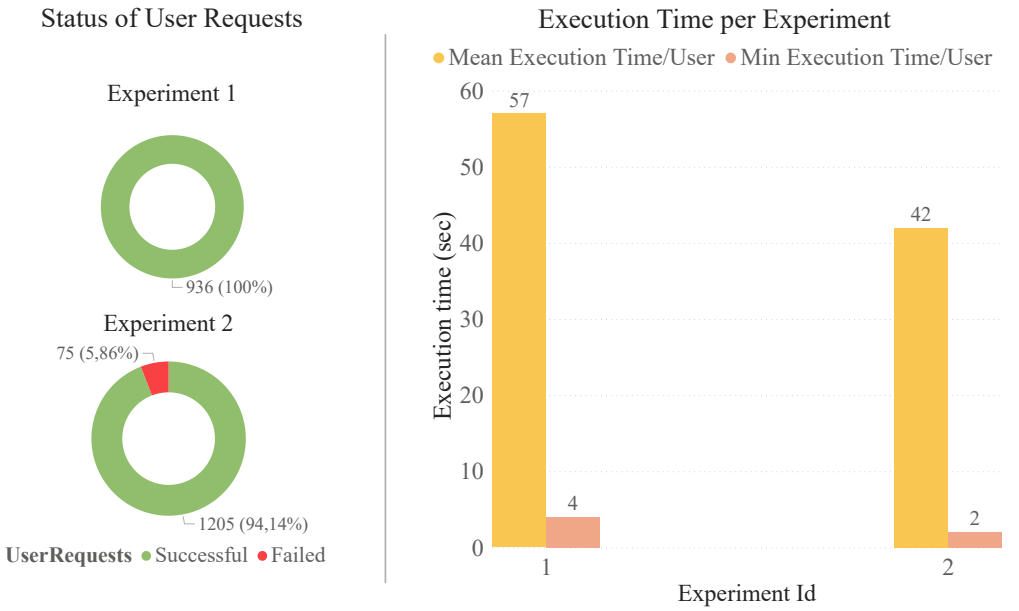


Figure 6.5: Performance measurements of database activity experiment

database. In other words, if the administrator knows that the database typically serves around 100 users, but there are some days where 180 users query the database for some time, then a good threshold would be close to 80%. In another scenario, if the administrator knows that the same database typically serves around 100 users, but there are some days where the workload **progressively** increases from 100 to 180 users, then it would be more rational to set a threshold of 40%. In this way, the autoscaler would be more proactive, as it will scale out early and the database will be completely ready (rebalancing will be done) to serve the user requests during the peak load.

The same logic can be applied on the other parameters of the autoscaler, like the duration of the increase of the active users. We remind that in our experiment, the scaling operation was performed whenever there was a change to the active users, **observed for more than three minutes**. If the administrator knows in advance that there are fluctuations in the user activity that do not last for too long, then the value of this parameter should be higher to avoid wrong decisions and oscillations. The last parameter that needs to be tuned is the number of VMs to be added or removed from the cluster. This parameters is one of the most important, as it directly affects the performance of the system and it also determines the actual cost of the cloud resources.

6.3.3 Algorithm Parameter Tuning

As we already mentioned, an important part of the autoscaling mechanism is the selection of the different algorithm parameters. This section highlights the importance of the number of VMs that the autoscaler decides to add or remove from the cluster during the plan phase of the MAPE loop. To show the effect of the tuning of this parameter, we repeat the same experiments, as described in Section 6.3.2, by **modifying only the number of**

VMs to be added/removed. More specifically, the autoscaler of experiment 3 adds **six VMs** to the cluster, instead of two VMs that were added during experiment 2. Figure 6.7 presents the results of all the experiments.

Regarding the number of user requests that the system was able to handle, we observe that experiment 2 and 3 shows approximately the same statistics, as in experiment 2 around 6% of the requests failed while in experiment 3 the same quantity amounts to 5%. In addition, we can see that the total number of user requests increased from 936 in experiment 1 to 1205 in experiment 2 and from 936 in experiment 1 to 1407 in experiment 3. Comparing the user capacity, we see that by adding two more VMs (experiment 1 and 2), we can serve around 28% more users during the two hours of the test, while by adding six VMs (experiment 1 and 3), we get an increase of 50% of the successful user requests. Hence, **we can safely state that with the current dataset size, the autoscaler improves the user capacity of the system, as it is able to serve more user requests**.

Analyzing the response times per user request, presented by the bar chart of Figure 6.7, we can say that despite the apparently low performance of the Citus rebalancing mechanism, the system seems to partially scale as the mean execution time decreases when more VMs are added to the cluster. Specifically, the addition of two VMs, accelerates the response time by 1.3 times, as the mean execution time drops from 57 to 42 seconds. In addition, the addition of six VMs, shows a speed up of 1.6, as the same number drops from 57 to 35 seconds. Although the database seems to have better performance when more VMs are added to the cluster, the speed up is far from being perfect, as the ideal speed up when adding two and six VMs to the initial cluster of two VMs, would be 2 (v.s. 1.3) and 4 (v.s. 1.6), respectively. We should not forget that cloud providers theoretically provide infinite amount of virtual resources, but everything comes with a cost. This is also valid in our experiments, where in experiment 2, the cost of the cloud resources was doubled compared to experiment 1 and in experiment 3, the cost was multiplied by four. **Hence, we expect the profit (speed up) to be proportional to the cost, which according to our experiments and to the used technologies, seems not to be true**.

6.3.4 A More Realistic Benchmark

In the previous experiments, we explored the capabilities of our autoscaler on a distributed database but with a limited amount of data. The performance of the autoscaler needs to be tested with bigger dataset, especially in the case of Citus rebalancing mechanism that was proved to have low performance. At this point, we remind that the previous experiments were done with BerlinMod benchmark of scale factor 0.5, which amounts to approximately 4GB of data. We measured that the shards rebalancing, which in this case can be reduced to 2GB of data transferring between the VMs, took around one hour, that is more than expected. In this section, we repeat the same experiments, as described in Section 6.3.2, **by using BerlinMod benchmark of scale factor 4, which amounts to 32 GB of data and by changing the size of the VMs**, from B2s (2 CPUs, 4GB of RAM) to B4ms (4 CPUs, 16GB of RAM). As we drastically increase the size of the dataset, we also increase the size of the VMs, taking into consideration that B2s VMs

have quite limited capabilities. Increasing the available computational power, forbids the direct comparison of the results described in this section with the results of Sections 6.3.2 and 6.3.3. Table 6.2 summarizes the parameters of the benchmarks.

Parameter Name	**Parameter Value**
BerlinMod Benchmark Parameters	
Scale Factor	4
Total Size (GB)	32
Scalar Benchmark Parameters	
Concurrent Users	15
Total Duration (hours)	2
User Request	BerlinMod Query14
VM Parameters	
# of CPU(s)	4
RAM (GB)	16
Disk Size (GB)	60
Azure VM Name	B4ms
Disk Type	Premium SSD

Table 6.2: Realistic experiment parameters

Figure 6.8 depicts the performance of the autoscaling mechanism in a more realistic scenario where the database has more data. From the bar chart, we realize that **our intuition regarding Citus rebalancing mechanism is confirmed**. Despite the fact that the system scales out early and adds two more VMs to the database cluster, **the mean response time remains the same**, hence, we doubled the cost of the cloud resources without having any profit. This is also confirmed by the total successful user requests that in experiment 2 were less compared to experiment 1.

Figure 6.9 shows the CPU utilization percentages of the VMs that participate in the experiment. The figure illustrates the same process as in Figure 6.6, but in this case the database has significantly more data (32GB v.s. 4GB). In the beginning of the experiment, only worker 1 and 2 are alive as they are part of the initial cluster. Around 12:15, the cluster scales out, worker 3 and 4 are created and the shards rebalancing begins. However, we clearly observe that in this case the transferring of the shards takes much longer than in the previous experiments. At this point, we remind that the rebalancing of the shards takes place during the first minutes that the new VMs are alive, where it is observed a workload transfer between worker 3 and 4. It is obvious that this time, transferring one shard from one VM to the other takes around ten minutes in the beginning (e.g. the first three shards), but this time increases as the experiment proceeds. For instance, the fourth shard (worker4) takes more than the previous, as the transferring begins at 12:50 and finishes at 1:10. The performance becomes even lower for the sixth shard, that starts around 1:20 and does not terminate even after the end of the experiment (2:00). As a result, after the end of the test that lasts two hours, **the system was not able to complete the shard rebalancing**, as only 6 of the 32 shards were sent from the existing to the new VMs. This observation justifies the poor performance of experiment 2 compared to experiment 1, even after the addition of two new VMs to the cluster.

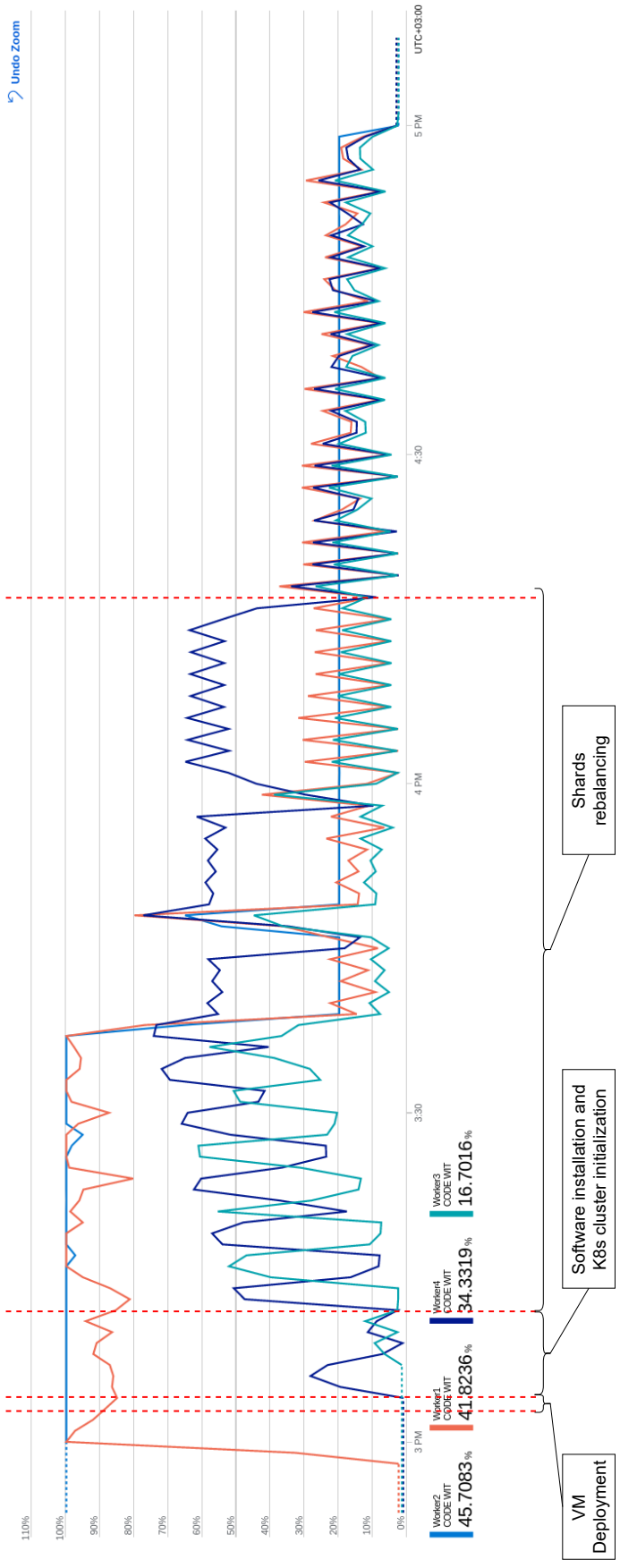


Figure 6.6: CPU utilization of the worker machines during the two hours experiment

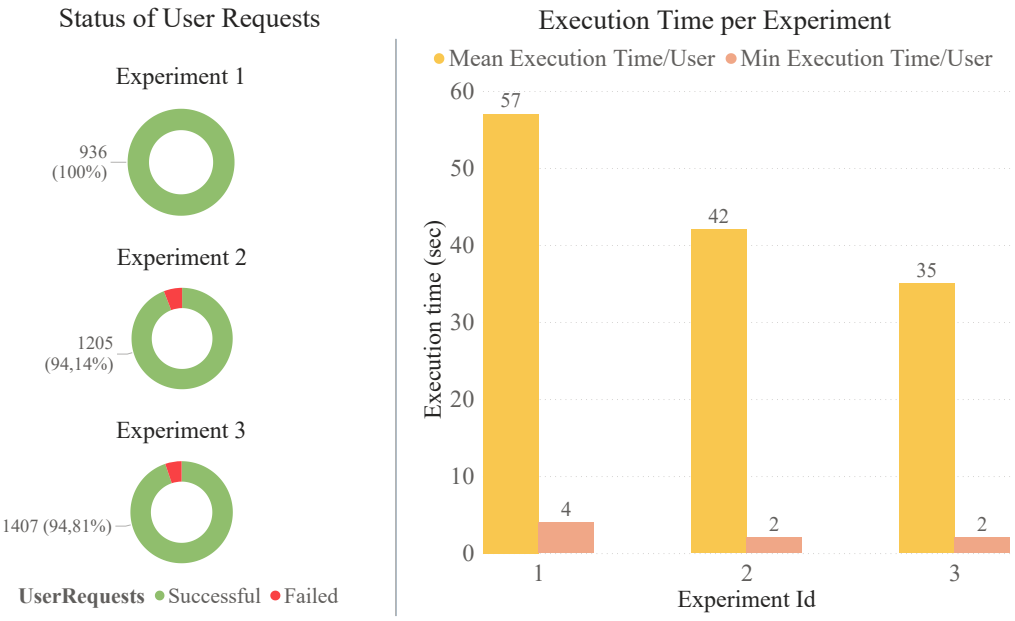


Figure 6.7: Additional performance measurements of database activity experiment

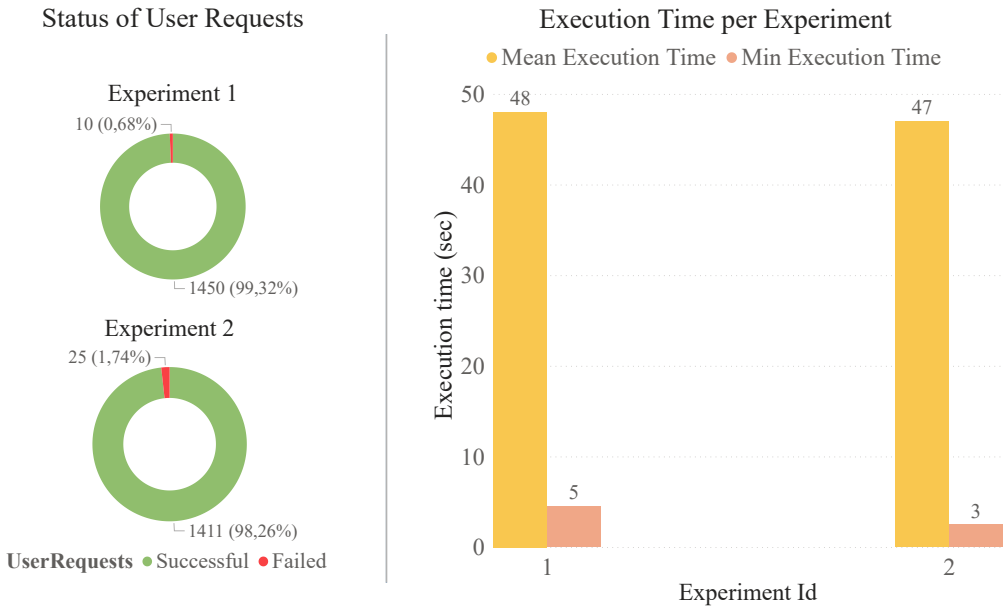


Figure 6.8: Performance measurements of database activity experiment with a bigger dataset

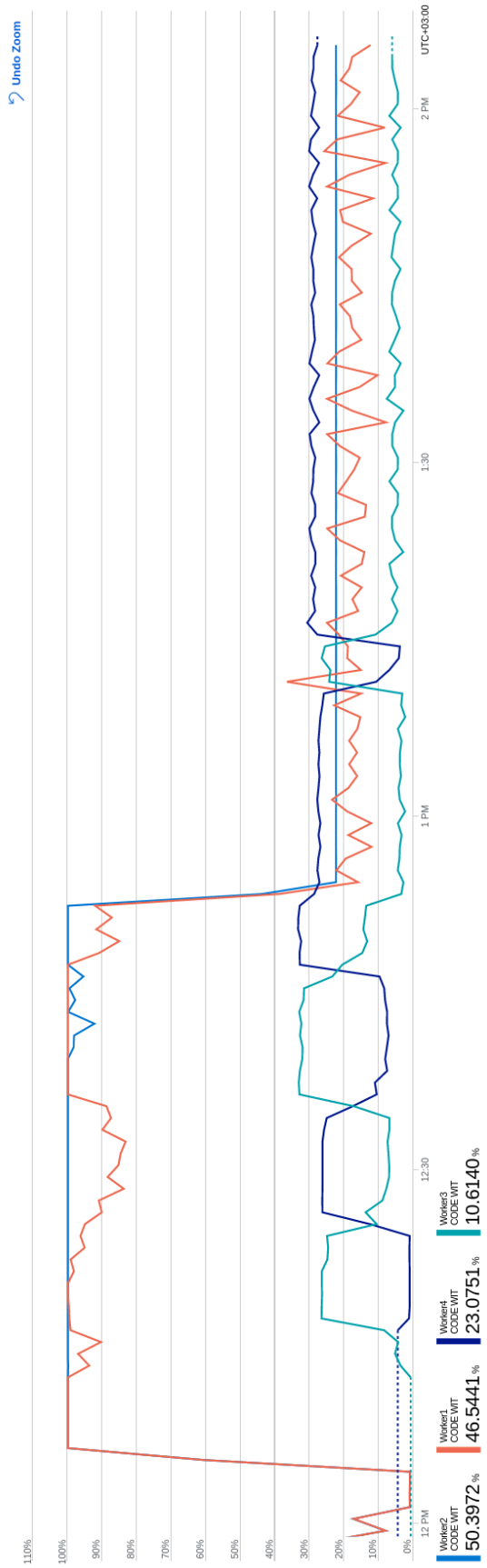


Figure 6.9: CPU utilization of the worker machines during the more realistic experiment

Chapter 7

Conclusion and Perspectives

In this last chapter, we summarize the main contributions of this work in the development of a self-scalable, cloud solution of MobilityDB on Azure. Furthermore, we outline the strongest and the weakest points of our solution as well as some limitations that are imposed by the underneath technologies. Finally, we provide some directions for future research around the deployment of MobilityDB on the cloud.

7.1 Conclusion

In this work, we studied the challenge of deploying and managing MobilityDB, a moving object database that is an extension of PostgreSQL, on the cloud. The abstract goal of this work was to provide a Database-as-a-Service solution on Azure. Designing and implementing such a service, requires automation of several repetitive tasks, such as the rolling out of the initial cluster on the cloud. Specifically, we provide a set of bash scripts that are able to provision the required cloud infrastructure, with minimal user interaction, install and configure all the required software in order to deploy a distributed MobilityDB database. To enable distribution, Citus extension is used. Automating the deployment of a Citus cluster introduces more challenges, as networking and configuration of the nodes are some of the issues that arise.

Second, our deployment targeted to further apply automation, by automatically managing the deployed solution. To respond to this challenge, we designed and proposed a Kubernetes architecture, responsible to monitor and maintain the state of the system, to be as close as possible to the desired state, specified by the database owner.

Third, owning a fully distributed system on the cloud demands human decision making that determines how the system responds to the incoming workload, in terms of scaling in or out. With our work, we provide a fully extensible autoscaler that takes over those repetitive tasks of monitoring the DBMS, analyzing the collected data and accordingly adapting the size of the cluster. Regarding the monitored metrics, we were experimented with the CPU utilization of the deployed infrastructure as well as with the change of the active user sessions. We state that our solution is fully extensible, as other developers can define their own strategies to implement autoscalers based on other metrics. The developed algorithm is versatile, as it is not bound to any specificity of MobilityDB. This means that the database can be used to host any other data, regardless of their nature. The autoscaler will still be capable of executing its job.

Regarding the results, we can safely state that the goals of this work were successfully achieved. Concerning the automatic deployment and management of the cluster, the performance and functionalities of the algorithms are very satisfactory. By using our tool, a user can deploy a MobilityDB or PostgreSQL cluster on Azure in around 15 minutes, by

only configuring and executing a bash script in the beginning of the process. Hence, our tool can be widely used by DevOps engineering teams that extensively use Infrastructure-as-Code to create distinct (e.g. development and production) but identical environments or by non-expert users that want to use an open source Database-as-a-Service PostgreSQL. In addition, our tool reaps the benefits of Kubernetes frameworks that ensures that the deployed cluster is always available to serve the database users. Again, all this maintenance tasks are automatically handled by the system without any administrative intervention.

Regarding the developed autoscaling mechanisms, we mainly proposed two different ruled-based autoscalers that collect and analyze performance metrics from different application levels. As database scenarios are various, there is not one autoscaler ideal for every use case. The provided Python tool can be easily extended by the users that want to apply their own logic to efficiently automate the scaling operations of their clusters.

By measuring the performance of our autoscalers, designed and tuned for BerlinMOD and Scaler benchmarks in MobilityDB, we can conclude that although some performance metrics seem not to be ideal (e.g. CPU utilization), others (e.g. user sessions number) give better results. Moreover, from our experiments we found that a major cost of a database autoscaler is the transfer of the data from the existing to the new cluster nodes. Specifically, we concluded that using Citus shards rebalancing mechanism, we were able to efficiently perform all the required scaling operations in a reasonable amount of time for small datasets (4GB), but the data transfer cost significantly increased for bigger dataset, e.g. 30GB.

In this section, we summarized the contributions of this work. In the next section, we provide ideas for future works that can perform further research around the topic of self-scalable cloud databases.

7.2 Future Work

Multi-cloud is a first aspect. A multi-cloud application leverages multiple cloud computing platforms for different tasks. In addition, another interesting aspect would be the migration of our solution to other cloud providers, namely AWS or GCP in order to explore to what extent the current software is supported as well as investigate other cloud services that can optimize its features.

In Section 4.2.2, we discussed a limitation of MS Azure Files that can be used as **external disks** for the VMs. With the current implementation, Azure Files do not support the storage of a PostgreSQL file system. By using external disks as storage for the database of each worker node, it is highly likely that the performance issues, observed during the shard rebalancing, will be downgraded, especially in the case when only one big external disk will be used across every worker node. One possible service would be AWS S3 Bucket [AWSb], a storage system ideal for storing unstructured data.

PostgreSQL native distributing mechanisms is a hot recent topic that a lot of research is being done on. An interesting new aspect of this work is to experiment with other alternative tools and technologies, different than Citus, that introduce data distribution to PostgreSQL. PostgreSQL Wiki community [Conb] actively discusses, gives guidelines and sets standards around the scale out design of PostgreSQL. For instance,

standard PostgreSQL partitioning schemes can be leveraged to horizontally partition a relational table. Then a distribution query engine is needed to appropriately distribute the data over the nodes, distribute the SQL queries and perform all the other operations that are under the responsibility of Citus coordinator node.

Last but not least, there is a lot of space for designing and building new types of autoscalers. As we already mentioned, one finding of this work is that there is not one perfect autoscaler for every database scenario. An OLTP system has completely different behavior compared to an OLAP system. One potential future work is the creation of a set of autoscalers, each one of them targeting on different types of systems (e.g. an autoscaler optimized for OLTP system). Furthermore, it is a challenge to investigate and experiment with more sophisticated, **proactive autoscalers**, that use machine learning or time series analysis techniques to predict future states of the system in order to make more accurate decisions.

Appendix A

Source Code

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: citus-master
spec:
  selector:
    matchLabels:
      app: citus-master
  replicas: 1
  template:
    metadata:
      labels:
        app: citus-master
    spec:
    #Assign Citus coordinator to master node (needs to have untainted master node first)
      nodeSelector:
        dedicated: master
      containers:
        - name: citus
          image: dimitris007/mobilitydb:citus10_pgcron
          imagePullPolicy: "IfNotPresent"
          args:
            - -c
            - max_locks_per_transaction=128
            - -c
            - shared_preload_libraries=citus,postgis-2.5.so,pg_stat_statements,pg_cron
            - -c
            - cron.database_name=$(POSTGRES_DB)
            - -c
            - ssl=on
            - -c
            - ssl_cert_file=/etc/postgresql-secrets-vol/server.crt
            - -c
            - ssl_key_file=/etc/postgresql-secrets-vol/server.key
          ports:
            - containerPort: 5432
          lifecycle:
            postStart:
              exec:
                #Command to be executed after the initialization of the worker Pod
                command: ["/bin/sh", "-c", "sleep 2; psql -U ..."]
        env:
          - name: POSTGRES_DB
            valueFrom:
              secretKeyRef:
                name: postgres-secrets-params
                key: db
```

Listing 10: postgres-deployment.yaml (part1). Part of the command parameter has been removed as it is too long to be displayed. The complete command can be seen on the GitHub repository.

```
    - name: POSTGRES_USER
      valueFrom:
        secretKeyRef:
          name: postgres-secrets-params
          key: username
    - name: POSTGRES_PASSWORD
      valueFrom:
        secretKeyRef:
          name: postgres-secrets-params
          key: password
    volumeMounts:
      - mountPath: /var/lib/postgresql/data
        name: postgredb
      - mountPath: /etc/postgresql-secrets-vol
        name: secret-vol
      - mountPath: /etc/postgresql-secrets-vol/params
        name: secret-vol-params
securityContext:
  runAsUser: 0
  supplementalGroups: [999,1000]
  fsGroup: 999
volumes:
  - name: postgredb
    persistentVolumeClaim:
      claimName: postgres-pv-claim-coordinator
  - name: secret-vol
    secret:
      secretName: postgresql-secrets
      defaultMode: 0640
  - name: secret-vol-params
    secret:
      secretName: postgres-secrets-params
```

Listing 11: postgres-deployment.yaml (part2)

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: citus-worker
spec:
  selector:
    matchLabels:
      app: citus-workers
  serviceName: citus-workers
  replicas: 3
  selector:
    matchLabels:
      app: citus-workers
  template:
    metadata:
      labels:
        app: citus-workers
    spec:
    #Add Node Affinity to equally distribute the replicas in the Cluster's nodes.
    #Assign explicitly 1 replica per node with requiredDuringSchedulingIgnoredDu...
    #Do not allow to assign Worker nodes to the Control Plane node
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: "app"
                    operator: In
                    values:
                    - citus-workers
                    - citus-master
              topologyKey: "kubernetes.io/hostname"
      containers:
        - name: citus-worker
          image: dimitris007/mobilitydb:citus10
          imagePullPolicy: "IfNotPresent"
          args:
            - -c
            - max_locks_per_transaction=128
            - -c
            - shared_preload_libraries=citus,postgis-2.5.so
            - -c
            - ssl=on
            - -c
            - ssl_cert_file=/etc/postgresql-secrets-vol/server.crt
            - -c
            - ssl_key_file=/etc/postgresql-secrets-vol/server.key
```

Listing 12: postgres-deployment-workers.yaml (part1)

```
        - -c
        - max_connections=500
      ports:
        - containerPort: 5432
      lifecycle:
        postStart:
          exec:
            #Commands to be executed after the initialization of the worker Pod
            command: ["/bin/sh", "-c", "sleep 7;printf \"local all all trust..."]
      env:
      - name: POSTGRES_DB
        valueFrom:
          secretKeyRef:
            name: postgres-secrets-params
            key: db
      - name: POSTGRES_USER
        valueFrom:
          secretKeyRef:
            name: postgres-secrets-params
            key: username
      - name: POSTGRES_PASSWORD
        valueFrom:
          secretKeyRef:
            name: postgres-secrets-params
            key: password
      - name: POD_IP
        valueFrom:
          fieldRef:
            fieldPath: status.podIP
      volumeMounts:
        - mountPath: /var/lib/postgresql/data
          name: postgredb
        - mountPath: /etc/postgresql-secrets-vol
          name: secret-vol
        - mountPath: /etc/postgresql-secrets-vol/params
          name: secret-vol-params
    securityContext:
      runAsUser: 0
```

Listing 13: postgres-deployment-workers.yaml (part2). Part of the command parameter has been removed as it is too long to be displayed. The complete command can be seen on the GitHub repository.

```
        supplementalGroups: [999,1000]
        fsGroup: 999
      volumes:
        - name: postgredb
          persistentVolumeClaim:
            claimName: postgres-pv-claim
        - name: secret-vol
          secret:
            secretName: postgresql-secrets
            defaultMode: 0640
        - name: secret-vol-params
          secret:
            secretName: postgres-secrets-params
```

Listing 14: postgres-deployment-workers.yaml (part3)

```
apiVersion: v1
kind: Secret
metadata:
  name: postgres-secrets-params
data:
  username: cG9zdGdyZXNhZG1pbg==
  password: YWRtaW4xMjM0
  db: cG9zdGdyZXM=
```

Listing 15: postgres-secrets-params.yaml

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: postgres-pv-volume-coordinator
  labels:
    type: local
    app: citus-master
spec:
  storageClassName: manual
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/home/azureuser/coordinatordata"
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: postgres-pv-claim-coordinator
  labels:
    app: citus-master
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
```

Listing 16: postgres-storage-coordinator.yaml

```
def delete_node(self, nodes_ip):
    cur = self.connection.cursor()

    for node_ip in nodes_ip:
        # Mark the worker node with node_ip to be deleted
        cur.execute("SELECT * FROM citus_set_node_property('%s', 5432,
                                'shouldhaveshards', false)" % node_ip)

    # Drain all the marked node at once
    cur.execute("SELECT * FROM rebalance_table_shards(drain_only := true)")

    # Remove the nodes from the Cluster
    for node_ip in nodes_ip:
        cur.execute("SELECT master_remove_node('%s', 5432)" % node_ip)

    # Commit the transaction
    self.connection.commit()

    # Close the cursor
    cur.close()
```

Listing 17: Citus delete node function

```
def delete_cluster_nodes(self, new_cluster_size, worker_numbers):
    # Scale in the Stateful Set
    subprocess.check_call(['sudo', 'kubectl', 'scale', 'statefulsets',
                           'citus-worker', '--replicas=%s' % str(new_cluster_size)] )
    # Wait 10 seconds until the pod is terminated
    time.sleep(10)
    for worker_num in worker_numbers:
        worker_name = "worker"+str(worker_num)
        print(worker_name)
        # Drain the node
        subprocess.check_call(['sudo', 'kubectl', 'drain', worker_name,
                               '--ignore-daemonsets'])
        # Delete the node
        subprocess.check_call(['sudo', 'kubectl', 'delete', 'node', worker_name])
```

Listing 18: Kubernetes delete node function

```
def delete_vms(self, vms_numbers):
    for vm_number in vms_numbers:
        vm_name = "Worker"+str(vm_number)
        nic_name = vm_name+"VMNic"
        nsg_name = vm_name+"NSG"
        ip_name = vm_name+"PublicIP"

        # Delete VM
        try:
            async_vm_delete = self.compute_client.virtual_machines
                              .begin_delete(self.resource_group, vm_name)
            async_vm_delete.wait()
            net_del_poller = self.network_client.network_interfaces
                             .begin_delete(self.resource_group, nic_name)
            net_del_poller.wait()

            # Wait until the Network Interface is deleted to proceed
            while(not net_del_poller.done()):
                sleep(5)

            async_nsg = self.network_client.network_security_groups
                        .begin_delete(self.resource_group, nsg_name)
            async_nsg.wait()
            async_ip = self.network_client.public_ip_addresses
                       .begin_delete(self.resource_group, ip_name)
            async_ip.wait()
            disks_list = self.compute_client.disks
                         .list_by_resource_group(self.resource_group)
            disk_handle_list = []

            # Delete the attached disks
            for disk in disks_list:
                if vm_name in disk.name:
                    async_disk_delete = self.compute_client.disks
                                        .begin_delete(self.resource_group, disk.name)
                    disk_handle_list.append(async_disk_delete)

            for async_disk_delete in disk_handle_list:
                async_disk_delete.wait()
        except CloudError:
            print('A VM delete operation failed: {}'.format(traceback.format_exc()))
```

Listing 19: Azure delete node function

Appendix B

Tutorial

The experiments done in this work are reproducible as the code is freely available to everyone. The goal of this tutorial is to explain step by step all the technicalities and configuration that someone needs to know in order to reuse our tool. **The equivalent tutorial, written in markdown language, can be found in the README file of the GitHub repository**.

B.1 Required Components

This work combines different tools and technologies to create a self-managed database on the cloud. The following list includes the required components along with some links that assist the users to install and configure them.

1. A local computer running **Linux OS** (tested with Ubuntu 20.04).
2. A **Microsoft Azure** account with an active subscription attached to it. The user must have full access to the Azure resources (owner).
3. A **Service Principal**, created and configured for your Azure account. More details on how to create a Service Principal can be found here.

B.2 Cluster Initialization

The first step is to clone the GitHub repository of this work. To initialize the cluster in Azure, the bash script **MobilityDB-Azure/automaticClusterDeployment/KubernetesCluster/deployK8SCluster.sh** should be used. The script executes the process illustrated in Figure 4.1. Before running the script, the parameters placed on the top of the file need to be configured as follows:

- **AzureUsername** parameter is used to login to your Azure account.
- The default **ResourceGroupName**, **Location** and **VirtualNetwork** values can be used.
- **Subscription** defines the name of the active Azure subscription.
- **VMsNumber** determines the number of Worker nodes and **VMsSize** the size of each machine.
- **SSHPublicKeyPath** and **SSHPrivateKeyPath** values specify the location of the ssh private and public keys to access the created VMs. By default, the files will be stored in ~/.ssh/ directory.
- **Gitrepo** specifies the Github repository from which the installation scripts and the rest source files will be found. The default value should be used.
- **Service_app_url** determines the url of the Service Principal and **Service_tenant** the tenant's id. When executing the script, the **Client secret** should

be given by the user to authenticate the application in Azure.

Now, you can execute the script by running `bash MobilityDB-Azure/automaticClusterDeployment/KubernetesCluster/deployK8SCluster.sh` in a terminal. After around 15 minutes, the cluster will be deployed on Azure.

When the cluster is ready, you can access any machine using the ∼/.ssh/id_rsa key. The next step is to establish an ssh connection with the coordinator VM. To connect to the machine, run `ssh -i ∼/.ssh/id_rsa azureuser@{vm_ip_address}`, where vm_ip_address is the public ip address of the coordinator VM that can be found in Azure portal. When connected to the VM, you can confirm that the K8S cluster has been successfully initialized by executing `sudo kubectl get nodes`.

B.3 Deploying a PostgreSQL Cluster

Until now we have created a Kubernetes cluster on Azure. The purpose of this section is to deploy a PostgreSQL cluster with Citus and MobilityDB extensions installed. First we need to modify the provided configuration files:

1. Edit the content of **MobilityDB-Azure/KubernetesDeployment/postgres-secrets-params.yaml** file by changing the values of the **username** and **password**. These credentials will be the default keys to access the PostgreSQL server. The values should be provided as base64-encoded strings. To get such an encoding, you can use the following shell command: `echo -n "postgres" | base64`.
2. Replace the content of the folder **MobilityDB-Azure/KubernetesDeployment/secrets** by creating your own SSL certificate that Citus will use to encrypt the database data. The existing files can be used for guidance.
3. Edit the content of **MobilityDB-Azure/KubernetesDeployment/postgres-deployment-workers.yaml** by setting the **replicas** to be equal to the number of available worker machines that you want to create.
4. Run `bash MobilityDB-Azure/KubernetesDeployment/scripts/startK8s.sh` to create the Kubernetes deployment. After some few minutes, the Pods will be created and ready to serve the database.

Now you are ready to connect to your distributed PostgreSQL cluster. After connecting to the coordinator VM, execute the following shell command to ensure that the Pods are running, as shown in Figure B.1: `sudo kubectl get pods -o wide`. Normally, you should see one Pod hosting the citus-master and a number of citus-worker Pods, equal to the replica number that you defined before.

You can connect to the Citus coordinator by using the **public ip** of the master VM as **host name/address**, **30001 as port**, **postgres as database** and the **username** and **password** that you defined before. The default values are postgresadmin and admin1234, respectively. Try to execute some Citus or MobilityDB queries. For instance, run `select master_get_active_worker_nodes()` to view the available Citus worker nodes.

```
azureuser@Coordinator:~/MobilityDB-in-Azure/KubernetesDeployment$ sudo kubectl get pods -o wide
NAME                              READY   STATUS    RESTARTS   AGE   IP           NODE          NOMINATED NODE   READINESS GATES
citus-master-6b65df8f54-mg5z9     1/1     Running   1          13m   10.32.0.6    coordinator   <none>           <none>
citus-worker-0                    1/1     Running   1          13m   10.44.0.1    worker1       <none>           <none>
```

Figure B.1: Example of deployed Pods with one master and one worker node

B.4 Self-scalable PostgreSQL Cluster

Assuming we have successfully done all the previous step, we are now ready to turn the PostgreSQL database into a self-managed cluster. The process of monitoring and applying an auto-scaling mechanism is managed by a script, implemented as a daemon process and written in Python 3. To execute the script, follow the step below:

1. Replace the parameters on the **top** of the **MobilityDB-Azure/autoscaling/scripts/addNewVms.sh** file with **the same parameters** that you provided in Appendix B.2.
2. Execute `sudo -s` command on the coordinator VM to get root access rights.
3. Create a virtual environment by running `python3 -m venv venv` and activate it `source venv/bin/activate`.
4. Install the required packages by first running `pip install setuptools-rust`, `export CRYPTOGRAPHY_DONT_BUILD_RUST=1` and `pip install -r MobilityDB-Azure/autoscaling/requirements.txt`.
5. Export the following environment variables, by adjusting their values as follows:
   - `export AZURE_SUBSCRIPTION_ID=...`, `export AZURE_TENANT_ID=...`, `export AZURE_CLIENT_ID=...` and `export AZURE_CLIENT_SECRET=...` by specifying the corresponding values from the Azure Service Principal.
   - `export RESOURCE_GROUP=...` with the Azure resource group name.
   - `export POSTGREDB=postgres`, `export POSTGREUSER=...` and `export POSTGREPASSWORD=...` with the corresponding server credentials.
   - `export POSTGREPORT=30001`
   - `export SCRIPTPATH=/home/azureuser/MobilityDB-Azure/autoscaling/scripts`, assuming you have cloned the source code into /home/azureuser path.
6. Copy the content of **~/.ssh/id_rsa.pub** file from your local machine to the same path in the coordinator VM.
7. Finally, execute the following command to launch the auto-scaler: `python3 MobilityDB-Azure/autoscaling/AutoscalingDaemon.py –action start –minvm 2 –maxvm 8 –storage /home/azureuser/autolog –lower_th 70 –upper_th 30 –metric sessions`. You can get more information regarding the available parameters by running `python3 MobilityDB-Azure/autoscaling/AutoscalingDaemon.py –help`. **Note:** the auto-scaler is a daemon process hence, the script can be executed in the background. Information about the state of the auto-scaler can be found in **/var/log/autoscaling.log** and **/var/log/autoscaling_performance.log** log files.

Bibliography

[AWSa] AWS. Amazon relational database service (rds). https://aws.amazon.com/rds/. Last accessed 01 July 2021. (Cited on page 3.)

[AWSb] AWS. Amazon s3. https://aws.amazon.com/s3/. Last accessed 01 July 2021. (Cited on page 78.)

[AWSc] AWS. Aws auto scaling. https://aws.amazon.com/autoscaling/. Last accessed 01 July 2021. (Cited on page 17.)

[BB10] Anton Beloglazov and Rajkumar Buyya. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science, MGC '10, New York, NY, USA, 2010. Association for Computing Machinery. (Cited on page 17.)

[BSZ19] Mohamed Bakli, Mahmoud Sakr, and Esteban Zimanyi. Distributed moving object data management in mobilitydb. BigSpatial '19, New York, NY, USA, 2019. Association for Computing Machinery. (Cited on page 23.)

[BSZ20a] 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. (Cited on page 23.)

[BSZ20b] Mohamed Bakli, Mahmoud Sakr, and Esteban Zimányi. Distributed mobility data management in mobilitydb. In 2020 21st IEEE International Conference on Mobile Data Management (MDM), pages 238–239, Los Alamitos, CA, USA, jul 2020. IEEE Computer Society. (Cited on pages 23, 30 and 31.)

[BWZ15] Len Bass, Ingo Weber, and Liming Zhu. DevOps: A Software Architect's Perspective. Addison-Wesley Professional, Boston, MA, USA, 1st edition, may 2015. (Cited on page 39.)

[Cona] PostgreSQL Contributors. Postgresql wiki metrics. https://wiki.postgresql.org/wiki/Monitoring. Last accessed 01 July 2021. (Cited on page 53.)

[Conb] PostgreSQL Contributors. Scaleout design. https://wiki.postgresql.org/wiki/Scaleout_Design#Introduction. Last accessed 01 July 2021. (Cited on page 78.)

[CP17] Emiliano Casalicchio and Vanessa Perciballi. Auto-scaling of containers: The impact of relative and absolute metrics. In 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W), pages 207–214, Los Alamitos, CA, USA, sep 2017. IEEE Computer Society. (Cited on page 14.)

[Data] Crunchy Data. Crunchy postgresql operator. https://www.crunchydata.com/products/crunchy-postgresql-operator/. Last accessed 01 July 2021. (Cited on page 9.)

[Datb] Crunchy Data. Disaster recovery. https://access.crunchydata.com/documentation/postgres-operator/latest/architecture/disaster-recovery/?_gl=1*1x3y99e*_ga*MTU1ODE5MDE0MS4xNjIzOTIwNjAx*_ga_ZQQNV12TZ5*MTYyNDk1NDg0MS4zLjAuMTYyNDk1NDg0MS4w. Last accessed 01 July 2021. (Cited on page 9.)

[Datc] Crunchy Data. High-availability. https://access.crunchydata.com/documentation/postgres-operator/latest/architecture/high-availability/?_gl=1*1jdxept*_ga*MTU1ODE5MDE0MS4xNjIzOTIwNjAx*_ga_ZQQNV12TZ5*MTYyNDk1NDg0MS4zLjAuMTYyNDk1NDg0MS4w. Last accessed 01 July 2021. (Cited on page 9.)

[DBG09] Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting. Berlinmod: A benchmark for moving object databases. The VLDB Journal, 18(6):1335–1368, dec 2009. (Cited on page 61.)

[DMM+10] Xavier Dutreilh, Aurélien Moreau, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. From data center resource allocation to control theory and back. In 2010 3rd IEEE International Conference on Cloud Computing, pages 410–417, Los Alamitos, CA, USA, jul 2010. IEEE Computer Society. (Cited on pages 16 and 19.)

[DTR+18] Wito Delnat, Eddy Truyen, Ansar Rafique, Dimitri Van Landuyt, and Wouter Joosen. K8-scalar: A workbench to compare autoscalers for container-orchestrated database clusters. In 2018 13th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), SEAMS '18, page 33–39, New York, NY, USA, may 2018. Association for Computing Machinery. (Cited on page 63.)

[Ent21] EnterpriseDB. Cloud native postgresql. https://www.enterprisedb.com/products/postgresql-on-kubernetes-ha-clusters-k8s-containers-scalable, 2021. Last accessed 01 July 2021. (Cited on page 9.)

[GBD10] Ralf Hartmut Güting, Thomas Behr, and Christian Düntgen. SECONDO: A platform for moving objects database research and for publishing and integrating research implementations. IEEE Data Engineering Bulletin, 33(2):56–63, 2010. (Cited on page 61.)

[Goo21] Google. Cloud sql. https://cloud.google.com/sql, 2021. Last accessed 01 July 2021. (Cited on page 3.)

[Groa] The PostgreSQL Global Development Group. Postgresql streaming replication. https://www.postgresql.org/docs/current/warm-standby.html#STREAMING-REPLICATION. Last accessed 01 July 2021. (Cited on page 29.)

[Grob] The PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database. https://www.postgresql.org. Last accessed 01 July 2021. (Cited on page 2.)

[GSLI11] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. Exploring alternative approaches to implement an elasticity policy. In 2011 4th IEEE International Conference on Cloud Computing, pages 716–723, Washington, DC, USA, jul 2011. IEEE Computer Society. (Cited on page 14.)

[Her21] Alvaro Hernandez. Deconstructing postgres into a cloud native platform. https://www.postgresvision.com/agenda, 2021. (Cited on page 10.)

[Hey14] Thomas Heyman. Scalar. https://distrinet.cs.kuleuven.be/software/scalar/, 2014. Last accessed 01 July 2021. (Cited on page 63.)

[Hyp21] Azure Hyperscale. Azure database for postgresql - hyperscale *citus* documentation. https://docs.microsoft.com/en-us/azure/postgresql/hyperscale/, 2021. Last accessed 01 July 2021. (Cited on pages 3 and 28.)

[IT21] Solid IT. Db-engines. https://db-engines.com/en/, 2021. Last accessed 01 July 2021. (Cited on page 1.)

[JPT17] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. Time series management systems: A survey. IEEE Transactions on Knowledge Data Engineering, 29(11):2581–2600, aug 2017. (Cited on page 18.)

[JPT18] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. Modelardb: Modular model-based time series management with spark and cassandra. Proceedings of the VLDB Endowment, 11(11):1688–1701, jul 2018. (Cited on page 19.)

[Kuba] Kubernetes. Cluster autoscaler. https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler. Last accessed 01 July 2021. (Cited on page 17.)

[Kubb] Kubernetes. Horizontal pod autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Last accessed 01 July 2021. (Cited on page 17.)

[Kubc] Kubernetes. Kubernetes components. https://kubernetes.io/docs/concepts/overview/components/. Last accessed 01 July 2021. (Cited on page 35.)

[Kubd] Kubernetes. Learn kubernetes basics. https://kubernetes.io/docs/tutorials/kubernetes-basics/. Last accessed 01 July 2021. (Cited on page 48.)

[Kube] Kubernetes. Production-grade container, orchestration. https://kubernetes.io/. (Cited on page 34.)

[Kubf] Kubernetes. What is kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Last accessed 01 July 2021. (Cited on page 35.)

[LBMAL14] Tania Lorido-Botran, J. Miguel-Alonso, and J. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. Journal of Grid Computing, 12:559–592, oct 2014. (Cited on pages 14 and 16.)

[Les21] Alexey Lesovsky. Postgresql observability. https://pgstats.dev/?version=14, 2021. Last accessed 01 July 2021. (Cited on page 53.)

[Mah] Sakr Mahmoud. Analyzing gps trajectories at scale with postgres, mobilitydb, citus. https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/analyzing-gps-trajectories-at-scale-with-postgres-mobilitydb-amp/ba-p/1859278. Last accessed 01 July 2021. (Cited on page 24.)

[MBEB11] Michael Maurer, Ivan Breskovic, Vincent C. Emeakaroha, and Ivona Brandic. Revealing the mape loop for the autonomic management of cloud infrastructures. In 2011 IEEE Symposium on Computers and Communications (ISCC), pages 147–152, Los Alamitos, CA, USA, 2011. IEEE Computer Society. (Cited on page 14.)

[Mer14] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. Linux journal, 2014(239):2, mar 2014. (Cited on page 33.)

[Mica] Microsoft. Azure autoscale. https://azure.microsoft.com/en-us/features/autoscale/. Last accessed 01 July 2021. (Cited on page 17.)

[Micb] Microsoft. Azure database for postgresql single server. https://docs.microsoft.com/en-us/azure/postgresql/overview-single-server. Last accessed 01 July 2021. (Cited on page 3.)

[Micc] Citus Data Microsoft. Citus data. https://www.citusdata.com/. Last accessed 01 July 2021. (Cited on pages 3 and 27.)

[Micd] Citus Data Microsoft. Citus data github. https://github.com/citusdata/citus. Last accessed 01 July 2021. (Cited on page 28.)

[Mice] Citus Data Microsoft. Rebalancer strategy table. http://docs.citusdata.com/en/v10.0/develop/api_metadata.html#pg-dist-rebalance-strategy. Last accessed 01 July 2021. (Cited on page 65.)

[Moba] MobilityDB. Berlinmod benchmark for mobilitydb. https://github.com/MobilityDB/MobilityDB-BerlinMOD. Last accessed 01 July 2021. (Cited on page 61.)

[Mobb] MobilityDB. Mobilitydb github repository. https://github.com/MobilityDB/MobilityDB. Last accessed 01 July 2021. (Cited on page 23.)

[MSaZ21] Mahmoud Maxime Schoemans andSakr and Esteban Zimányi. Implementing rigid temporal geometries in moving object databases. 2021 37th IEEE International Conference on Data Engineering (ICDE), pages 2547–2558, apr 2021. (Cited on page 23.)

[OnG21] OnGres. Enterprise postgres made easy. on kubernetes. https://stackgres.io/, 2021. Last accessed 01 July 2021. (Cited on page 10.)

[RO04] IBM Redbooks and International Business Machines Corporation. International Technical Support Organization. A Practical Guide to the IBM Autonomic Computing Toolkit. IBM redbooks. IBM, International Support Organization, 2004. (Cited on page 14.)

[RS21] E.G. Radhika and G. Sudha Sadasivam. A review on prediction based autoscaling techniques for heterogeneous applications in cloud environment. Materials Today: Proceedings, 45(2):2793–2800, jan 2021. (Cited on pages 16 and 17.)

[Sev] Severalnines. Using kubernetes to deploy postgresql. https://severalnines.com/database-blog/using-kubernetes-deploy-postgresql. Last accessed 01 July 2021. (Cited on page 48.)

[Sha16] Ashraf A. Shahin. Automatic cloud resource scaling algorithm based on long short-term memory recurrent neural network. International Journal of Advanced Computer Science and Applications, 7(12), dec 2016. (Cited on page 18.)

[TJDB06] G. Tesauro, N.K. Jong, R. Das, and M.N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In 2006 IEEE International Conference on Autonomic Computing, pages 65–73, Los Alamitos, CA, USA, jun 2006. IEEE Computer Society. (Cited on page 19.)

[WB21] Adam Wright and Gabriele Bartolini. Prosper with postgres on kubernetes. https://www.postgresvision.com/agenda, 2021. (Cited on page 9.)

[Zal21] Zalando. Patroni: A template for postgresql ha with zookeeper, etcd or consul. https://github.com/zalando/patroni, 2021. Last accessed 01 July 2021. (Cited on page 10.)

[ZS21a] Esteban Zimányi and Mahmoud Sakr. Managing moving objects data with mobilitydb. https://pgconf.ru/en/2021/291542, 2021. Last accessed 01 July 2021. (Cited on page 23.)

[ZS21b] Esteban Zimányi and Mahmoud Sakr. Mobilitydb: Managing mobility data in postgresql (emea). https://www.postgresvision.com/, 2021. Last accessed 01 July 2021. (Cited on page 23.)

[ZSL20] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. Mobilitydb: A mobility database based on postgresql and postgis. ACM Transactions on Database Systems, 45(4), dec 2020. (Cited on page 23.)

[ZSLB19] Esteban Zimányi, Mahmoud Sakr, Arthur Lesuisse, and Mohamed Bakli. Mobilitydb: A mainstream moving object database system. SSTD '19, page 206–209, New York, NY, USA, aug 2019. Association for Computing Machinery. (Cited on pages 3 and 23.)