UNIVERSIDADE D
COIMBRA

Eurico José Pereira de Sousa

ANALYSIS OF MULTI-TENANCY IN DATA WAREHOUSE DATABASES

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Filipe Araújo and Prof. Pedro Furtado and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Eurico José Pereira de Sousa

Analysis of Multi-tenancy in Data Warehouse Databases

From Single-Tenancy to Multi-Tenancy

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Filipe Araújo and Prof. Pedro Furtado and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2023

Acknowledgements

This work is funded by the project POWER (grant number POCI-01-0247-FEDER-070365), co-financed by the European Regional Development Fund (FEDER), through Portugal 2020 (PT2020), and by the Competitiveness and Internationalization Operational Programme (COMPETE 2020).

This work is funded by the FCT - Foundation for Science and Technology, I.P./MCTES through national funds (PIDDAC), within the scope of CISUC R&D Unit - UIDB/00326/2020 or project code UIDP/00326/2020

POWER

empowering a digital future

Co-financed by:

Partners:

Abstract

Software plays a pivotal role in driving diverse industries and businesses. As organizations strive to streamline operations and reduce costs, many are transitioning from on-premise infrastructures to cloud-based solutions. Software as a Service (SaaS) leverages multi-tenancy, where multiple customers utilize the same infrastructure, enjoying unique configurations and data, thus improving scalability and efficiency while minimizing expenses.

With this internship I implement multi-tenancy in the analytical microservice, Alexa, within the Autonomous Service Operations Platform (ASOP) from Altice Labs (ALB), as part of the POWER project. This involves extending support for various APIs that it contains, and also its data persistence layer. This corresponds to the first stage of this work, which is intended to provide practical knowledge, for the analysis of multi-tenancy presented in the next paragraph, which is the center point of this dissertation.

From the previous stage I move on to the primary focus of this dissertation, which is an in-depth analysis of three distinct approaches to implementing multi-tenancy in data warehouse databases, concentrating on throughput performance, privacy considerations in terms of data isolation, and the overall complexity of the approaches. The study adheres to the guidelines outlined by the TPC-H Benchmark, and concentrates on the persistence layer, setting the groundwork for the development of a porting tool cable of automating the process of porting cloud-based systems, from single-tenant to multi-tenant.

Keywords

Alexa. Analysis. Data-warehouse. Multi-tenancy. Porting.

Resumo

O software desempenha um papel fundamental no progresso de diversas indústrias e empresas. Conforme as organizações procuram otimizar operações e reduzir custos, verifica-se um aumento na migração de infraestruturas locais para soluções baseadas na cloud. O Software como Serviço (SaaS) toma partido de multi-tenancy, onde vários clientes utilizam a mesma infraestrutura, mas com configurações e dados exclusivos, melhorando a escalabilidade e a eficiência, reduzindo os custos.

Com este estágio eu implemento multi-tenancy no microsserviço analítico, Alexa, no ASOP do ALB, como parte do projeto POWER. Isto envolve alargar o suporte de várias APIs e da camada de persistência de dados para tal, o que corresponde à primeira fase deste trabalho, e que pretende fornecer conhecimentos práticos para a análise em multi-tenancy apresentada no parágrafo anterior, que representa o ponto fulcral desta dissertação.

A partir da etapa anterior, segue-se o ponto fulcral desta dissertação que corresponde a uma análise aprofundada de três abordagens distintas para a implementação de multi-tenancy em bases de dados do tipo data warehouse, concentrando-se no desempenho de throughput, na privacidade em termos de isolamento de dados e na complexidade geral das abordagens. O estudo segue as directrizes delineadas pela Benchmark do TPC-H e concentra-se na camada de persistência, estabelecendo as bases para o desenvolvimento de uma ferramenta de porte capaz de automatizar o processo de converter sistemas baseados na cloud de single-tenant para multi-tenant.

Palavras-Chave

Alexa. Análise. Data-Warehouse. Multi-tenancy. Porte.

Contents

Acronyms

ALB Altice Labs.

ASOP Autonomous Service Operations Platform.

CISUC Centre for Informatics and Systems of the University of Coimbra.

DB Database.

DBMS Database management system.

DDL Data Definition Language.

DEI Department of Informatics Engineering.

DML Data Manipulation Language.

EC2 Amazon Elastic Compute Cloud.

FCTUC Faculty of Sciences and Technology at the University of Coimbra.

IaaS Infrastructure as a service.

OLAP Online Analytical Processing.

OS Operating System.

PaaS Platform as a service.

RDS Amazon Relational Database Service.

SaaS Software as a service.

SQL Structured query language.

TPC-H TPC-H Benchmark.

VM Virtual machine.

List of Figures

Chapter 1

Introduction

This is a dissertation in the context of the POWER project that is taking place in the Centre for Informatics and Systems of the University of Coimbra (CISUC) at the Department of Informatics Engineering (DEI) of the University of Coimbra. It is associated with Altice Labs (ALB) and is for a Master's Degree in Software Engineering from the Faculty of Sciences and Technology at the University of Coimbra (FCTUC), in the curricular year of 2022/2023.

1.1 Context and Motivation

Software plays a crucial role in powering various industries and businesses [23]. Such industries use a variety of software to perform all their operations uninterruptedly, creating an increasingly competitive environment that pushes them to replace on-premise infrastructures for cloud-based solutions, to reduce time to market, development, maintenance, and operational costs.

In this context, Software as a service (SaaS) emerges as a pivotal model, offering scalable and cost-effective solutions by delivering software applications over the cloud. It takes advantage of the economies of scale, enabled by multiple customers using the services of a company from the same set of infrastructure and resources. This model aligns with the practice of multi-tenancy, where a shared infrastructure supports individual customer configurations, workflows, and data. This approach not only optimizes resource utilization and operational efficiency but also fosters greater flexibility and customization for each tenant. As a result, businesses can harness the benefits of increased scalability and efficiency while simultaneously reducing costs.

The concept of multi-tenancy is not a recent innovation. It first appeared in the 1960s as explained in [19]. The origins of this idea stem from universities that were looking to provide students access to the growing technology of computers. And to do that more efficiently, a software technique entitled time sharing was created to enable several people to use the same machine at the same time.

In the past, this concept was not widely adopted as it was not deemed essential for most industries. However, today it has become a vital concept, as it carries the benefits of SaaS like reduced costs of maintainability and scalability in the realm of cloud computing, which is now a fundamental aspect of most modern industries.

For this reason and also because this internship is a collaboration between DEI and ALB, this dissertation is divided into two stages. The first focuses on learning how to implement this concept in a data warehouse system based on a star schema, by porting the Alexa microservice from single tenant to multi-tenant. This microservice has three separate APIs, one for ingesting data, another for managing data, and another for query data. The process of porting Alexa requires adding multi-tenant support to these APIs, and also in the database. More details about this process can be found in section 3.

From the previous stage, the main purpose is to learn insights into what a system like a data warehouse needs to be ported to a multi-tenant one, and then proceed to the second and primary stage, where the objective is to conduct an in-depth analysis on how to implement this type of software technic on a data warehouse system based on snowflake schema, to compare aspects like performance, privacy or data isolation, and complexity of implementation of the three different approaches considered, that I explain in detail in section 5. For this process the TPC-H Benchmark (TPC-H) was used to guide the testing phase, hence the analysis being focused on snowflake schema.

1.2 Objectives

The first objective of this internship is to port the analytical microservice from the Autonomous Service Operations Platform (ASOP), called Alexandria, or Alexa in short, in the context of the POWER project, from single-tenant to multi-tenant.

The Alexa functionalities include meta-data management, data loading, and data querying, all done through APIs. The porting process for this microservice involves adding support for multi-tenancy at the data loading API, meta-data management API, querying API, and at the data persistence layer. This process is important to give practical experience and knowledge on how to implement this technique on this type of system, but also because it gives some insights on how to port similar systems, by showing one possible implementation of this technique on all three layers of a cloud-based system.

The second objective is to compare three different ways of implementing multi-tenancy in a data warehouse database, with the intent of analyzing throughput performance, privacy in the sense of data isolation, and complexity of implementation of the solutions used. These different approaches provide different views on multi-tenancy, which in turn give guidelines for, in future work, developing a porting tool capable of automating the process of converting cloud based systems from single-tenant to multi-tenant.

To have valid data sets and results this analysis follows the TPC-H [38] guidelines, that uses the snowflake schema. Moreover, the approaches considered only focus on the persistence layer of data warehouse systems, to perform an in-depth analysis on this layer.

1.3 Results

In the initial phase, the implementation of requirements for the Alexa microservice offered a focused insight into the intricacies of adapting a data warehouse microservice rooted in a star schema. The experience involved transforming the database into a multi-tenant model while aligning the business and logic layers accordingly. The development process executed in the persistence layer was then carried over to the analysis allowing me to perform a more generic study with the aid of the TPG-H Benchmark.

This study, delved into evaluating data isolation aspects and the complexities of three multi-tenancy solutions. Where complexity predictions leaned towards the first solution as being the most complex, because it requires tenants to be separated in individual schemas within the same database, as opposed to the other two solutions where tenants are all in the same schema, with the difference being that in solution 3 there is a grouping of tenants. However, all three approaches exhibited similar implementation intricacies.

Moreover, the study allowed me to conclude that there is a penalty in throughput performance when using the second or third solutions compared to the first, and the outcome revealed what was expected in terms of which solution would perform best, but the results hold some interesting surprises.

1.4 Document Structure

This dissertation is separated into the following Chapters:

Chapter 2 is dedicated to research for the state of the art, that aims to understand in detail, and explain to the reader, what is multi-tenancy, what concepts and technologies it involves, and what challenges it brings.

Chapter 3 details the architectural drivers for the porting process of the Alexa microservice. It shows the current architecture of the microservice and identifies the functional requirements, as well as restrictions for this stage of the work.

Chapter 4 describes the resulting architecture of ASOP, using the C4 model, and the implemented changes after the porting process is complete.

Chapter 5 presents the analysis definition in detail explaining the three solutions to be implemented and tested, as well as the workload definition, all the details about the database design involved, and metrics used to compare the approaches. All of this follows the TPC-H Benchmark guidelines.

Chapter 6 is where the test set-up is presented and explained, going through all the steps on how to create the database, how to generate the data sets and query sets, how to load the databases, and also what tests to run how they are executed.

Chapter 7 presents the results gathered from the executed tests, using a graphical medium, also presenting the results for the calculations from the metrics used, and ending the chapter with an analysis on the aspects analyzed.

Chapter 8 is reserved for the project schedule, where I show the real vs planned work schedules and talk about the risks involved.

Chapter 9 is where I do a final reflection, talk about future work and give some final thoughts about the whole process.

Chapter 2

State of the Art

2.1 Multi-tenancy introduction

A Multi-tenant application serves multiple customers, called tenants, where each tenant can customize some parts of the application, usually the styling of the graphical interface, business rules, and rationale, and their database schema, however, they cannot personalize the application's code.

The architecture behind multi-tenant applications, allows them to have only one instance of software running on a single server while serving multiple tenants. This means the environment is shared between tenants, but their logic is separated, allowing the reuse of a dedicated instance of configurations, and data, among other properties [25].

2.2 Multi-tenancy related concepts

Before going into details about multi-tenancy, it is best to explain what are some related terms, components, and technologies that can be involved. This section is more oriented for the type of reader less familiarised with such concepts and terms.

- **Cloud application**

As defined by Joel Shore "A cloud application, or cloud app, is a software program where cloud-based and local components work together. This model relies on remote servers for processing logic that is accessed through a web browser with a continual internet connection." [34].

Such servers are commonly placed in big data centers manage by third party companies, and their main "tasks may encompass email, file storage and sharing, order entry, inventory management, word processing, customer relationship management, data collection, or financial accounting features." [34].

- **Cloud computing**

  A system that allows for easy and immediate access to configurable computing resources, such as networks, servers, storage, applications, and services, which can be quickly set up and taken down with minimal effort or interaction with service providers. This model can be delivered in three different ways:

  - **Software as a service (SaaS)**

    The general definition of Software as a service (SaaS) is "The acronym SaaS stands for software as a service and is defined as a software licensing and delivery method in which software is accessed online, rather than installed in a device. In SaaS models all the software and applicable data are hosted on the provider's servers, meaning the provider is responsible for managing the security, availability, updates and performance of the applications." [21].

  - **Infrastructure as a service (IaaS)**

    The general definition of Infrastructure as a service (IaaS) is "Infrastructure as a Service (IaaS) is a cloud computing service that lets organizations rent resources like servers, network security features, and data centers. Instead of investing in building and maintaining your own IT infrastructure, you simply pay for the resources you need and focus on your software application." [11].

  - **Platform as a service (PaaS)**

    The general definition of Platform as a service (PaaS) is "Platform as a service (PaaS) is a cloud computing model where a third-party provider delivers hardware and software tools to users over the internet. Usually, these tools are needed for application development. A PaaS provider hosts the hardware and software on its own infrastructure. As a result, PaaS frees developers from having to install in-house hardware and software to develop or run a new application." [8].

- **Database**

As defined in the ORACLE official website, "A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (Database management system (DBMS)). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just database.

Data within the most common types of databases in operation today is typically modeled in rows and columns in a series of tables to make processing and data querying efficient. The data can then be easily accessed, managed, modified, updated, controlled, and organized. Most databases use structured query language (Structured query language (SQL)) for writing and querying data." [30].

Databases can be divided in two main types: relational databases and non relational databases. In the first, data is organized in through relationships where it is stored in one or more also called tables, (also called relations), of columns and rows. The latter does not use the tabular schema of rows and columns, instead uses a storage model tailored for the requirements of the type of data being stored, for example, a simple key and value pair model, as Json documents.

- **Virtual machine**

As defined in the vmware official website, "A Virtual Machine (Virtual machine (VM)) is a compute resource that uses software instead of a physical computer to run programs and deploy apps. One or more virtual “guest” machines run on a physical “host” machine. Each virtual machine runs its own operating system and functions separately from the other VMs, even when they are all running on the same host. This means that, for example, a virtual MacOS virtual machine can run on a physical PC." [20].

- **Container**

A software container is a way of packing up an application or service and everything required for it to run (the program to execute and all its dependencies, such as the code, system libraries, etc), regardless of environment, in a single place, making it easy to sharing a fully developed application or service. A container can be inside the nodes of a clusters or not, therefore a container is the lowest level of isolation.

- **Cluster**

In cloud computing a cluster is a group of inter-connected computers, or virtual machines, or hosts, connected within a virtual private cloud, that work together to support applications and middleware (e.g. databases). In a cluster, each computer is referred to as a “node”. A cluster can have many nodes, and each node can have many containers inside.

- **Microservice**

  As defined in [32]: "Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are:

  - Highly maintainable and testable
  - Loosely coupled
  - Independently deployable
  - Organized around business capabilities
  - Owned by a small team

The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack."

2.3 Types of Multi-tenancy

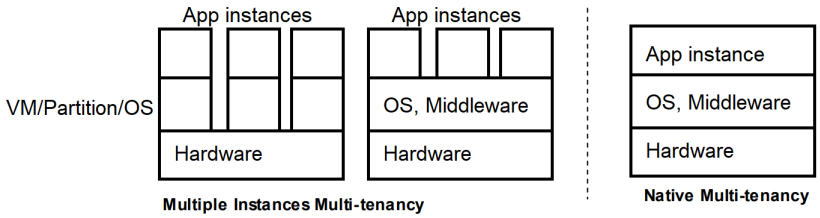

Figure 2.1: Multi-tenancy Patterns

As illustrated in Figure 2.1 multi-tenancy can be separated into two kinds of patterns: multiple instances and native multi-tenancy. In the native pattern everything is shared between tenants, the application instance, hardware, and Operating System (OS) or a middleware server in a hosting environment. On the other hand, the latter supports all tenants using the same hardware, but each tenant has its instance of the app and can share the middleware or not [15]. However, this is a broader view of the separation. Going into database separation, multi-tenant applications can be divided into three main models:

2.3.1 Multi-tenant app with one database per tenant

This model uses a single instance of a multi-tenant application with many databases, one for each tenant, where the database can be in a separate server or just a separate database within the same machine. This allows for complete tenant isolation, and complete customizability of each database schema, which is easy, due to the isolation level. Additionally, the noisy neighbor problem is avoided. The noisy neighbor problem refers to a scenario where one tenant's performance is affected by the actions of another tenant sharing the same resources. [24]. Since each tenant has its database, scaling can be done individually tenant by tenant [25] [31].

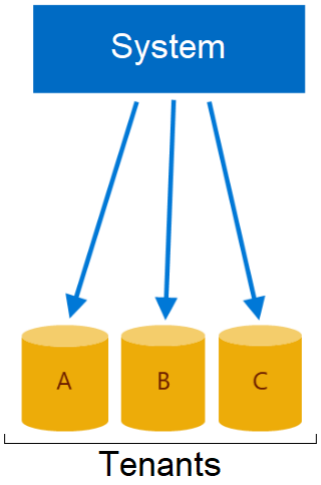

Figure 2.2: One database per tenant model

2.3.2 Multi-tenant app with multi-tenant databases

This model can be split into two sub models: one single database for every tenant; several databases for all the tenants, where each database houses data from various tenants.

In this model, tenant isolation is sacrificed, because data of multiple tenants are stored together in one database. Moreover, the computing and storage resources are shared between all tenants, and therefore the risk of having noisy neighbors is not negligible. However, multi-tenant databases have the lowest per-tenant cost because resources are shared. [25].

- **Multi-tenant app with a single multi-tenant database**

In this model, a single database is used to host the data of all tenants, and the separation between tenants can be made at table level, where tenant logic is handled in the application layer, which poses a security concern due to how easy it is to make mistakes and leak tenant data. An alternative is to have separation at the schema level (a schema is a blueprint for organizing data within a relational database, including table names, fields, data types, and relationships between entities, and it is used for data modeling and creating roles, such as database users, administrators, and programmers [10]), where the schema routing logic stays on the application side and can be encapsulated in a separate router [31]. To scale this model, more storage and compute resources are added scaling up, which can quickly become very difficult to manage [25].

- **Multi-tenant app with several multi-tenant databases**

This model uses several databases, also referred to as shards (sharding is a way to split a database into multiple smaller pieces called partitions, self-contained

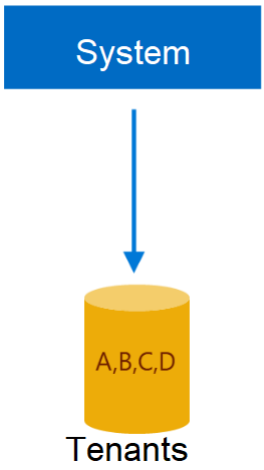

Figure 2.3: One database for all tenants model - separation at table level

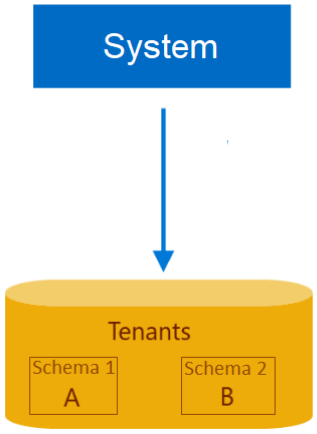

Figure 2.4: One database for all tenants model - separation at schema level

with the same schema and columns, but each with its own set of rows, allowing more efficient data management and improved performance [9]), to hold tenant data, where all the data for each tenant is contained in one shard. Sharding adds complexity and a catalog is required to maintain the mapping between tenants and databases, but the advantage is in resource efficiency, which is visible in scaling because it allows adding a new shard and populating it with new tenants, or in management by splitting densely populated shards into less-densely ones, or merging shards together that have few tenants and discontinued tenants, and even to balance workloads tenants can be moved around [25].

This combined with the previous model gives more scaling capabilities. By distributing tenants across multiple databases the result is smaller databases making management easier. One example is the process of restoring a tenant to a past state only involves restoring a single smaller database from a backup, that holds few tenants, rather than a larger database with all of them [25].

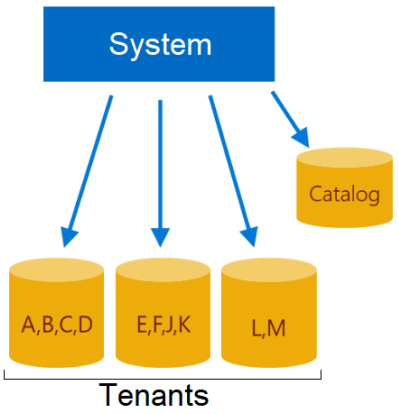

Figure 2.5: Several multi-tenant databases model

2.4 Multi-tenancy challenges

Multi-tenancy comes with a few challenges that should be addressed to better benefit from it. In this section, some of these challenges are presented, and the focus of this report is defined.

- **Application upgrades**

Starting with application upgrades, if an upgrade is something beneficial for all tenants, like a security upgrade, then all tenants should get it. But if the upgrade is not necessary for every tenant, then it is important to allow customization at run time without affecting other tenants. Normally, customization implies changing the code and a re-deployment of the application. However, in a multi-tenancy scenario, this is impractical because the same instance of the application is shared, therefore the customizations made by one tenant would be present for all tenants. Additionally, constantly redeploying can lead to frequent interruptions, which can become more common with a larger number of tenants, greatly reducing availability [33] [40].

- **Data isolation**

Another challenge is data isolation. Being able to keep tenant data isolated and secure inside a shared environment is crucial, and as mentioned before, there are three approaches to manage this in the Cloud: storing tenant data on separate Database (DB)s; using the same DB for multiple tenants, where each tenant has his own set of tables grouped into a schema; using the same DB, and the same set of tables for multiple tenants data [33] [27].

- **Performance**

Moving forward, performance and scalability pose another critical challenge, since resources are shared and tenants use them according to their needs, there must be a way to balance these resources so that if one tenant obstructs a significant amount of resources, the performance of all the others is not affected. This differs from the single-tenant scenario, in which the workload of a tenant only affects himself. In virtualized instances this problem is solved by assigning an equal amount of resources to each instance (or tenant), which can be very inefficient since not every tenant will need the same amount of resources, leading to wasted resources. This is regarding performance [33] [6].

- **Scalability**

In regards to scalability in a multi-tenancy scenario, another challenge rises, because there needs to be resource provisioning and allocation to satisfy all tenant's needs, the problem is those resources are shared which makes it more difficult to provision and allocate. Moreover, tenants from different countries impact scalability requirements, in the way that each country may have its legislation on data placement or routing. Finally, there may be more constraints related to specific requirements, like the need to place all data of one tenant on the same server to speed up regularly used DB queries. Such constraints influence how an application and its datastore can be scaled [33] [6].

- **Security**

If in single-tenant applications security is already an important part of it, in multi-tenant applications the resource sharing aspect, adds security, accessibility, and privacy concerns. For example, successful SQL attacks on a shared storage medium will expose and or corrupt the data of all tenants, hence the need for good security measures.

- **Zero-Downtime**

The need for constant growth and evolution is always present in multi-tenant systems because it is necessary for the process of adding new tenants or adapting the system to changing business requirements of existing tenants. But this process should not have any impact on the services provided to the other existing tenants. Therefore the requirement of zero downtime for multi-tenant applications arises, as downtime is usually very expensive in cloud applications [6].

- **Maintenance**

Has it was already explained, multi-tenancy brings a lot of benefits over single-tenancy, specifically in terms of deployment where the number of application and database instances that need to be updated is a lot smaller the in its counterpart. But in terms of maintenance, this clear and cut improvement is not so clear. In particular, introducing multi-tenancy into a software system adds complexity, which will likely affect the maintenance process and its cost. The amount of maintenance needed is probably correlated to the type of services provided by the software, but to have any kind of conclusion further research would be needed [6].

2.5 Advantages and risks of multi-tenancy

In the previous section, we discussed the various challenges that come with multi-tenancy, however, multi-tenancy also presents a number of advantages that can make it a valuable option for an organization or software solution. This section will delve deeper into these advantages and also identify potential risks, to give a balanced and holistic view of the implications of using multi-tenancy.

Advantages

- Fewer costs due to the sharing of resources. Since it is usually a cloud-based service, tenants only pay for what they need, and a lot of aspects of management like labor and staff, onboarding new tenants, maintenance, development, and updates, are handled by the cloud host. [35] [23].
- Because of the previous point, scaling is easy. Low cost means that tenants can add or remove resources as needed, and support for the microservice architecture is provided, allowing an entire batch of tenants to be provided with updates in one go [35] [23].
- It has good security, even though single tenancy can be more secure, multi-tenancy can also be very secure in tenant data separation [35]. Banking industries are one example of this, where multi-tenant architecture is used to maintain customer accounts and databases. [23].
- It is maintenance-free for the tenants. The host takes care of monitoring and administration costs, as well as upgrade and update costs. This is beneficial for both tenants, and hosts as they can make changes in a central application to share with all the tenants [35] [23].

Risks

- Third-party accessibility. Even though it can be very secure, while being a cloud service, there is always the risk of data being accessible to third parties, either through bugs or errors in the software or through malicious activity, like hacking [35].
- Response times can be affected. If one or more tenants are executing heavy tasks loading the hardware used by the cloud provider too much, response times for every tenant will slow down [35].
- The tenants have no control over server downtime, which is reserved for the host only [35].
- Multi-tenancy is more complex than single tenancy to implement [35].
- Introducing multi-tenancy at a later stage may be problematic. Processes like data migrations and changes in running services and data entities may cause problems, as well as backward incompatibility with older clients can occur [23].

2.6 Technologies

This section presents and explains some of the technologies and support tools that aid in the implementation of multi-tenancy. It focuses on container and cluster management tools, databases, and data warehouses specifically the star schema.

The reason why these tools are highlighted is that they possess specific characteristics that make them well-suited for the task of implementing multi-tenancy. Their features such as containerization, cluster management capabilities, and the ability to support multiple tenants make them a viable option to implement the multi-tenancy feature in a microservices architecture.

The goal of this section is to provide an overview of these tools and to demonstrate how they can be effectively used to implement multi-tenancy in a microservice-based application.

2.6.1 Container & cluster management support tools

Container and or cluster management tools are software programs that provide a graphical interface or command line window to manage clusters and containers. They allow monitoring of nodes and containers inside the clusters, configure services and administer the entire cluster server [1].

- **Docker**

Docker is an open platform for developing, shipping, and running applications in containers. Containers are isolated and secure environments allowing a single host to run many instances at once, making Docker well-suited for multi-tenancy implementation. However, these instances do not rely on host-installed software because they already have everything needed to run an application, plus they can be easily shared to other hosts [16].

Docker also provides a tool for managing clusters called Docker Swarm. It enables the creation of a cluster of one or more Docker Engines called a swarm. A swarm consists of one or more nodes: physical or virtual machines in swarm mode. There are two types of nodes: managers and workers [17], and the swarm can utilize them to scale up to 50,000 containers and 1,000 nodes with no effect on performance as new containers are added to the cluster [1].

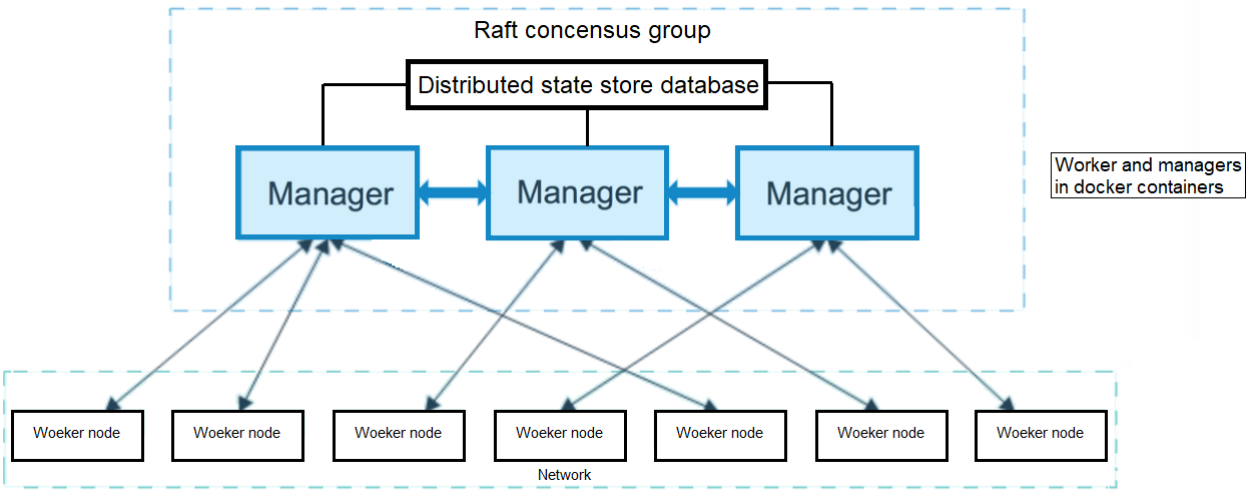

Figure 2.6: Docker swarm - relation between worker and manager nodes [17]

A brief description of worker and manager nodes is presented next. Visit [17] for more information.

Manager node description

This type of node handles cluster management tasks:

- maintaining cluster state
- serving swarm mode HTTP API endpoints
- scheduling services (containers)

Using a Raft implementation ("raft is a consensus algorithm for managing a replicated log. Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members" [28]), the managers maintain a consistent internal state of the entire swarm and all the services running on it [17].

Worker node description

Worker nodes are also instances of Docker Engine that execute containers. They don't participate in the Raft distributed state, make scheduling decisions, or serve the swarm mode HTTP API. Every worker node requires at least one manager node, and all managers are also workers by default [17].

Docker Swarm is a native solution, meaning, it is possible to implement Docker networking, plugins, and volumes using Docker commands. The Swarm manager creates several masters and specific rules for leader election, implemented in the event of a primary master failure. Additionally, the Swarm scheduler features a variety of filters including affinity and node tags, where filters can attach containers to underlying nodes for better resource utilization and enhanced performance, which allows easily deploy and management multi-tenancy in a microservices architecture.[1].

- **Kubernetes**

Developed by Google, "Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery."[5].

Figure 2.2 shows a control plane with components that make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when a deployment's replicas field is unsatisfied). Control plane components can be run on any machine in the cluster. However, for simplicity, set-up scripts typically start all control plane components on the same machine, and do not run user containers on this machine [4].

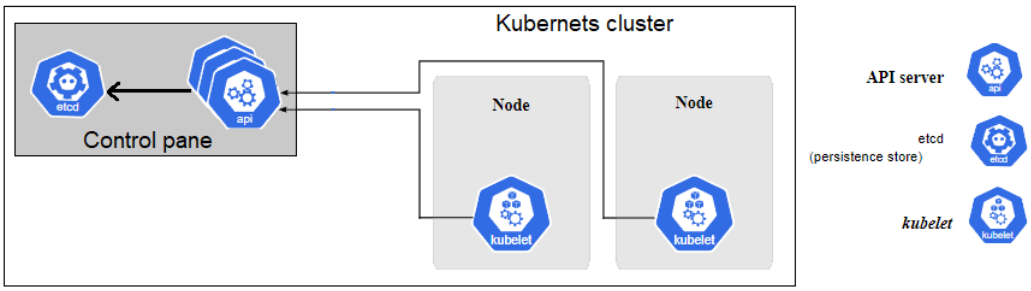

Figure 2.7: The components of a Kubernetes cluster

Cluster components descriptions

- kube-apiserver exposes the Kubernetes API.
- etcd is a key value store for cluster data. (See [18] to learn about etcd).
- kube-scheduler assigns nodes to newly created Pods.
- kube-controller-manager runs controller processes.
- cloud-controller-manager links the cluster into the cloud provider's API.

Node components descriptions

- kubelet runs on each node in the cluster and makes sure that containers are running in a Pod.
- kube-proxy is a network proxy that runs on each node of a cluster and maintains network rules, that allow network communication.

For more datailed information on this components visit [4].

As shown in Figure 2.2 Kubernetes is based on Clusters and Pods. A Pod is the smallest deployable unit that holds one or several containers that make

up a service, this is how containers are scheduled and deployed at the same time, which makes pods the basic configuration unit for scheduling. Additionally, pods are built and eliminated in real time as demand and requirements change [1]. Moreover, groups of pods, related or unrelated, run on a cluster grouped under logical borders called namespaces (namespaces are used to divide a cluster into smaller sections, similar to folders, and provide a way to manage access control, limits, and quotas for resources. [26]), in these groups, a pod is a unit of replication in the cluster.

Moving onto clusters, they provide all the logic for the nodes to execute and communicate. Regarding nodes, they stay between the pod and cluster and are essentially the machines, physical or virtual. Managed by the control plane they contain the services necessary to run Pods. [7].

Kubernetes architecture based on clusters and pods, makes it an efficient solution for multi-tenancy, it can handle changes in demand, and resources can be allocated between tenants with ease using the namespaces feature.

- **Fedora CoreOS - Fleet**

Fedora CoreOS is an automatically updating, minimal, monolithic, container-focused operating system, designed for clusters but also operable standalone. Rather than installing a package through apt or yum, CoreOS leverages Linux containers to handle services at a higher abstraction level, providing advantages similar to virtual machines, but with a concentration on applications rather than complete virtualized hosts [12] [1].

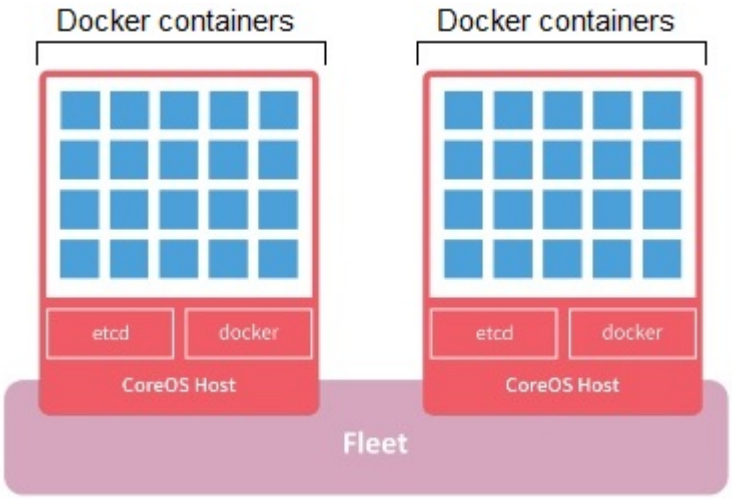

Figure 2.8: Fleet CoreOS with Docker

Moving onto Fleet the tool for cluster and container management. With Fleet, every machine has an agent and an engine, and the entire community of engines is active at all times in a cluster. Fleet utilizes etcd, and can also handle socket activation, meaning, containers can be activated to take care of a connection on a specific port. This allows the system to create processes when needed as opposed to waiting for demand. In the event of a machine failure, the containers are automatically moved to healthy machines [1].

CoreOS and Fleet provide an efficient solution for multi-tenancy by automatically handling the allocation of resources and responding to changes in demand.

- **Apache - Mesos**

As defined in the official documentation, Apache Mesos is a distributed system that abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively [13].

Apache Mesos can also be used as a cluster manager because it focuses on the effective isolation of resources and sharing of applications across distributed networks or frameworks. It is open source and gives managers the ability to share resources [1].

Apache Mesos is built using the same principles as the Linux kernel but with a different level of abstraction. It runs on every machine with one machine designated as the master running all the others and provides applications with APIs for resource management and scheduling across the entire data center and cloud environments. Moreover, any Linux program can run on Mesos and provides an extra layer of safeguards against failure [1].

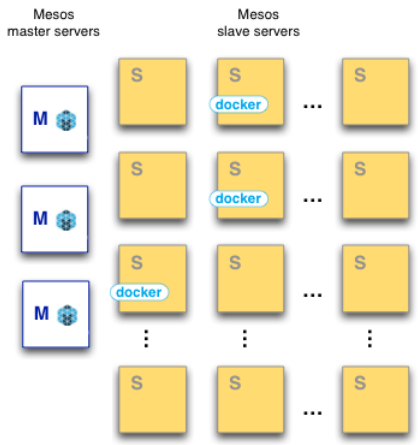

Figure 2.9: Apache Mesos architecture implementation with docker

Mesos uses a system of agent nodes to run tasks, where nodes can be divided into masters and slaves. The slave agents send a list of available resources to a master, and the master nodes distribute tasks to the slave agents. At any given time, there can be hundreds up to thousands of agent nodes in operation [1].

Apache Mesos is a good solution for multi-tenancy as it allows for the sharing and isolation of resources, the tasks are distributed across a large number of nodes, and provide an extra layer of safeguards against failure.

- **Decision**

These are some of the tools that are helpful for cluster and container management in multi-tenancy scenarios, however, since the technologies used in this project are already defined, a support tool will not be used.

2.6.2 Databases

This section compares the performance of relational and non-relational databases in multi-tenancy scenarios. PostgreSQL will serve as an example of a relational database because it is used by Alexa. The goal of this comparison is to highlight the importance of selecting the appropriate database type when implementing multi-tenancy and how it can affect the performance of the system.

- **Relational databases - PostgreSQL**

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2. It is open-source and supports a large part of the SQL standard [14] giving it a lot of community support. It runs on all significant Operating System (OS), it is compatible with a lot of different programming languages [22], and it supports Json (non-relational) queries, making this database technology very flexible.

PostgreSQL has many features but for this section, only the most relevant ones for multi-tenancy are mentioned. These include support for a customizable storage interface for tables, support for column and row-level security, and support for schema.

These features allow tenant separation and customization, in the same database instance. The first two are more oriented for tenant separation at table level, which is common in environments that do not require strong separation of tenant data, where every table that contains per-tenant data has a special column indicating which tenant the row belongs to. The latter is for tenant separation at the schema level, whose design helps organize data into separate entities, and allows for more organization and better communication among internal stakeholders because it provides a common source of truth ensuring data validity.

Additionally, some advantages that help in multi-tenancy are flexible data management; support for ACID properties (ACID properties in PostgreSQL ensure that transactions are atomic (completed in an all-or-nothing manner), consistent (valid according to predefined rules), isolated (determines the level of visibility by other transactions), and durable (stored permanently [41]); data integrity and security; reliable data storage with easy backup and recovery; compliance with data protection standards and regulations (such as GDPR); rapid integration with commercial software, and a streamlined development cycle.

Some drawbacks include hardware and software costs; scalability can be expensive; effective and efficient work with RDBMS is not an easy and fast skill to learn, it requires experience.

Two examples of PostgreSQL databases used by Alexa are TimescaleDB and CitusDB. TimescaleDB is an extension of PostgreSQL that is optimized for time-series data and analytics. It provides the scalability, performance, and usability that applications require when handling large streams of time-series data. Similarly, CitusDB is a PostgreSQL-based database that is specifically designed for

horizontally scaling out (horizontal scaling means adding more machines to handle more data or traffic, as opposed to adding more resources to a single machine - vertical scaling) data warehouses. It distributes data across a cluster of machines, allowing for efficient querying and processing of large amounts of data.

- **Non-relational databases**

Non-relational databases can be divided into five types: document data store, which saves data in a document entity stored in Json format, and does not require all documents to have identical data structures; columnar data store, which organizes data into columns allowing sparse data to be structured in a denormalized way (denormalization consist in introducing redundant data to improve read performance); key-value stores, it is a collection of key-value pairs contained within an object; document stores, it does not have a document structure specified with a schema, it stores documents with their original format and structure; graph, it is designed to efficiently store relations between entities when data is greatly interconnected, it is the most complex type.

Some benefits that non-relational databases have in multi-tenant applications are: faster development iterations due to dynamic schema, has seen in the previous paragraph the type of data structure in these databases is very flexible; horizontal elastic scalability for peak loads, for example, sharding; high availability, reliable performance, and better end-user experience, since they are built to serve low-latency requests; cheap implementation, they only require low-cost servers for effective operation; the use of Object-relational-mapping, giving non-relational databases good flexibility in terms of the query language to use.

Some disadvantages of NoSQL databases can be scalability issues in certain scenarios when the size of the data grows, lack standardization for data models, query languages, and APIs, require complex data modeling that make it difficult to understan and work with the data, and lack transactional consistency meaning that multiple updates to the data may not be executed atomically.

- **Final remarks**

As it is clear, both types of databases have pros and cons, and the selection of one over the other in any project depends on the requirements and objectives of that specific project. For the current project, the first type of database will be used, due to the reason aforementioned in the introduction of this section.

2.6.3 Data warehouses - star schema

In the context of this project, it is important to understand the concept of a data warehouse as Alexa is one. The examples given use the star schema because it is what Alexa uses.

A data warehouse is a type of data management system with the purpose of supporting business activities like the analytical analysis. Built from the operational data collected from transactional databases and other operational systems, they store large collections of historical data and are just intended to perform query operations to this data [29]. They also support advanced querying and analysis, such as Online Analytical Processing (OLAP) and data mining, that are beyond the capabilities of traditional transactional databases.

The key characteristics of a data warehouse are:

- Temporal dependency - data is collected over time, and it does not represent a specific moment, it represents history. This requires that a temporal reference be associated with all data in the database.
- Nonvolatile - the data in the data warehouse is never updated.
- Target-oriented - the data stored must be relevant for decision support, discarding all other types of data.
- Data integration and consistency - data must be integrated and made consistent before being loaded in the data warehouse. This requires a data format specification.
- Designed for queries - the data format specification, to store data, must be designed with query performance in mind, this is why multidimensional view and partial denormalization are common in data warehouses.

OLAP databases store data in a multi-dimensional schema, where data can be visualized as a data cube because it allows data visualization through multiple dimensions. This model has two main components, the Facts and Dimensions, and a central theme that is represented by a Fact table.

Facts are the metrics or measurements from a business process. A Fact table is normalized and contains the metrics (also referred to as facts), the foreign keys to one or more dimensions, and a timestamp for time reference.

A dimension provides the attributes that contextualize the process events of the business, in other words, it represents the who, the what, and the where of a fact. Dimensions are denormalized, and there is no limit to the number of dimensions a fact table can reference.

To help visualize the connection between facts and dimensions the star model is used as it is visible in Figure 2.10.

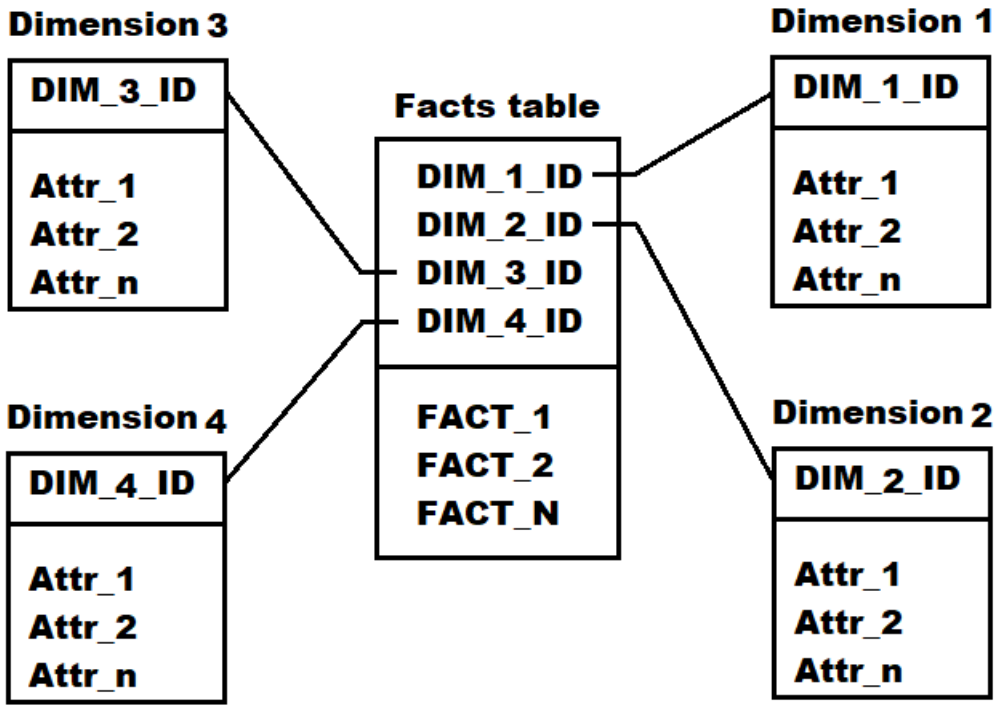

Figure 2.10: Star model visualization

To help visualize the Multidimensional data model Figure 2.11 shows a sales example, where facts can be units sold, sale value, etc, and a dimension can contain the brand of a product, the type, the name, etc.

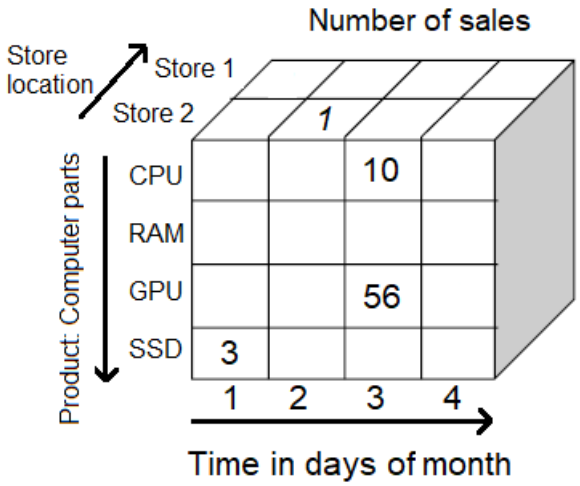

Figure 2.11: Sales Multidimensional model

Figure 2.12 shows one possible star model for the Sales example.

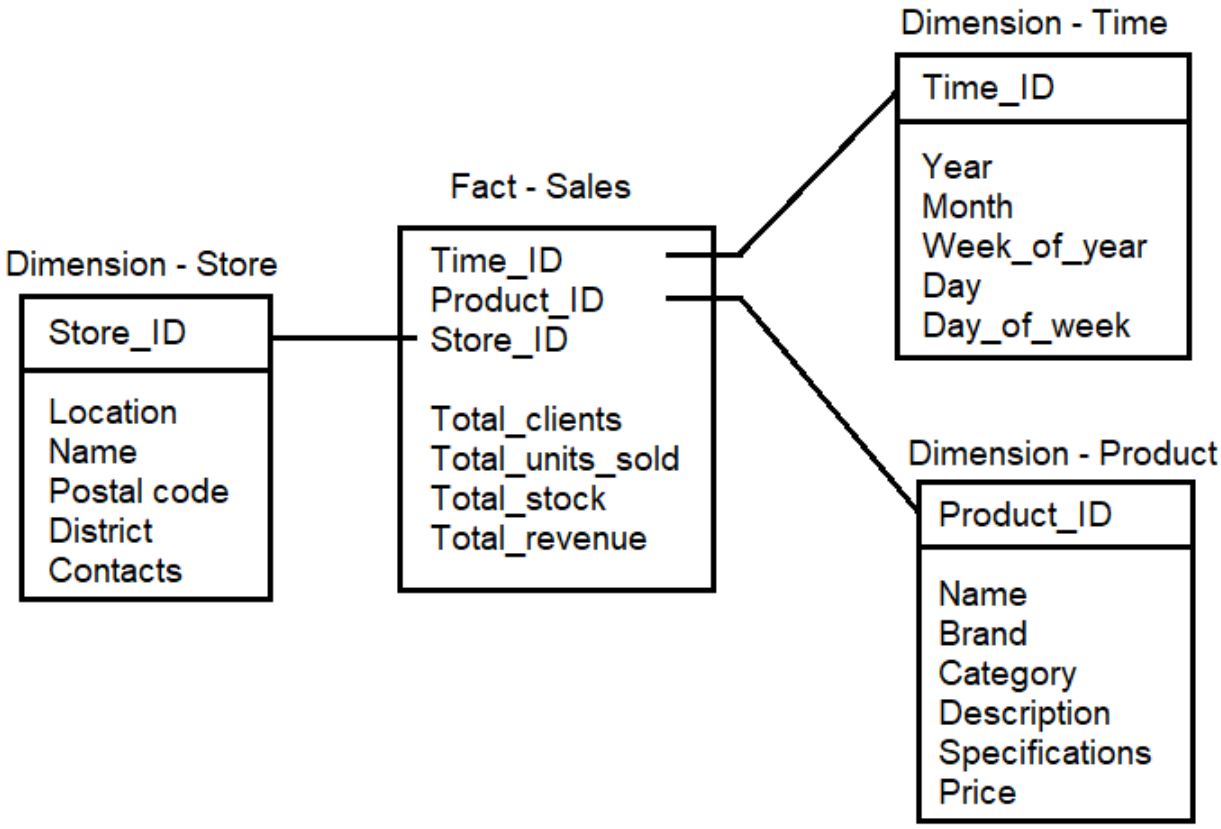

Figure 2.12: Sales star model example

Chapter 3

Architectural Drivers for Alexa Microservice

This Chapter describes the way the Autonomous Service Operations Platform (ASOP) analytical microservice functions, and also the architectural drivers for the porting process, which comprise functional requirements, and technical.

3.1 ASOP-Alexa current state

3.1.1 Architecture

In the ASOP the Alexa component is dedicated to storing and retrieving analytical data. Its functionalities include metadata management, data loading, and data querying. The main modeling concepts are

- Dimension - a business domain entity and its attributes, e.g., a CPE equipment with a Unique Identifier, Serial Number, Vendor, Model, Version, etc.
- Fact - a set of measurements, with a timestamp and the identifiers of the related dimensions.
- Timeseries - a flat view over one or more Fact tables, and selected dimensions.

The Alexa microservice provides a virtual flat table view over timeseries data stored in a physical star schema in the database. This means that the virtual flat table abstracts the business model from the clients. The only mandatory field for each timeseries is the timestamp, the other fields are chosen and defined by the microservice's clients and may include dimensional data and facts, see Figure 3.1 below.

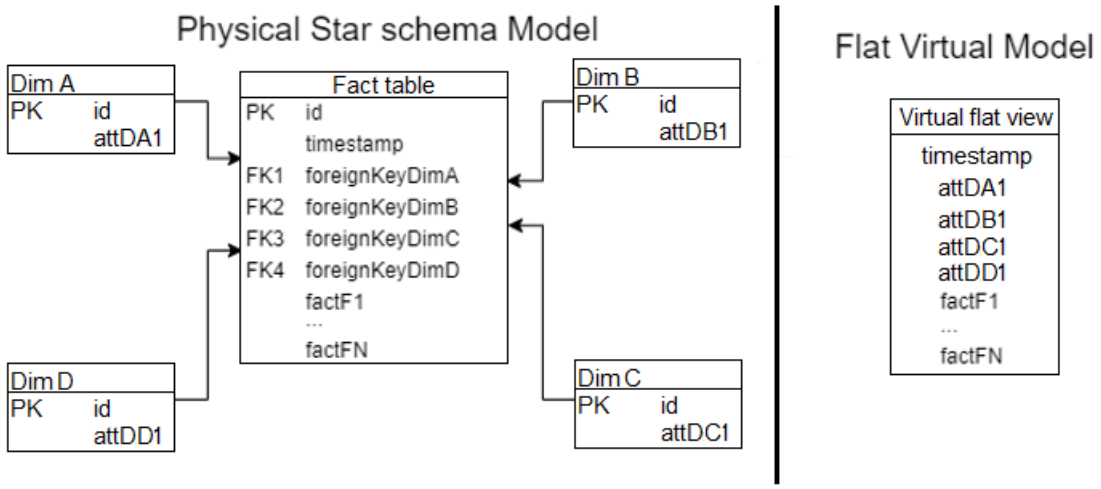

Figure 3.1: Alexa Virtual Flat vs Physical Star schema model

These different perspectives provide an abstraction layer between the business model (the flat view) and the persistence details, which may be chosen (and altered) according to functional, performance, or other non-functional requirements, without impacting the clients or their usage of the APIs.

The service uses docker containers to store all the data, and there are two types of data: metadata, always stored in the same Postgres database container (labeled "metadata" in Figure 3.2), and timeseries data which can be stored in one or many different database containers (represented in Figure 3.2 as "Database").

The metadata data contains the descriptions for the models of timeseries data. It details in what instance of database a timeseries is located, what facts and dimensions it holds, and also what attributes are included.

The timeseries data is organized in a star schema model, each instance of such database can have multiple timeseries, each timeseries can belong to many different clients, and each client can have multiple different timeseries. This means that there is a many-to-many relationship between clients and timeseries. Additionally, different stars (timeseries) can share fact and dimension tables, as long as they are in the same database instance.

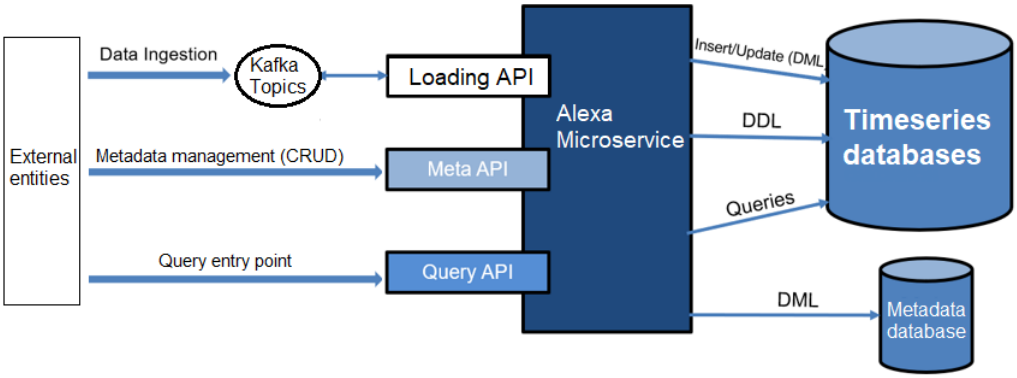

Figure 3.2: Alexa Logical Architecture

3.1.2 Metadata management

Alexa provides a REST API metadata management, it is a CRUD REST API (MetadataRestApi.java) with endpoints for timeseries, Facts, and Dimensions, to list, create, update, and delete. The API endpoints follow the OpenAPI/Swagger specification format [36].

When this API executes Data Manipulation Language (DML) operations in the metadata database, either to create or update a model of a timeseries, it also executes Data Definition Language (DDL) operations in the corresponding instance of timeseries database, in order to maintain consistency between the metadata and the timeseries data.

3.1.3 Data loading

Loading of data is done via Kafka, a distributed event streaming platform. Alexa provides a REST API (LoaderRestApi.java) for clients to write into the Alexa topic in Kafka, and the Alexa service is listening on this topic and writing to the database via DML operations, specifically Insert and Update, as seen in Figure 3.2. A timeseries event contains the Fact table attributes to write, and also the relevant Dimension attributes. A Dimension event contains only the Dimension's attributes.

3.1.4 Data querying

Data querying in Alexa is done via another REST API (ServiceRestApi.java). It serves all the clients and uses plugins to query data. There are two types of plugins: metadata plugins and query plugins. Query plugins are universal to all the metadata plugins, and are basically "SELECT" queries that fetch data from a certain instance of timeseries database. It can be TimescaleDB, CitusDB, ORACLE, etc.

When a client connects to the query API to get data, the request has a Json body with the required values to create the select query, like which timeseries to query and the attributes from the fact and dimension tables. This Json payload also includes the name of the database instance to query, specifying what timeseries the client wants. Alexa will then go to the metadata database to get the metadata plugin for that timeseries to know in what database it is located, in turn, the metadata plugin will call the correct query plugin to execute the query in the database with the parameters passed in the Json payload.

3.2 Functional Requirements

This section presents the requirements to port the Alexa analytical microservice from single tenant to multi-tenant, detailing what will be implemented, what conditions are involved, and the scenarios of usage. The requirements are aimed at the three APIs mentioned above as well as some data persistence details.

3.2.1 Data persistence requirements

The first requirement is making tenants explicit, which implies two changes *having a meta-model to represent the tenant entity* and adding a column called *tenant_id* to the fact and dimension tables to allow for data separation at row level.

This is done by adding a table to the metadata database called Tenant, and also to the timeseries databases that need to be multi-tenant. The metadata version of this table details which instances of the timeseries databases have a Tenant table, and when a timeseries database includes this table, it has all the information about the tenants, such as tenant ID and name.

Adding this column to the dimension tables will cause data replication which could affect performance. However, this will not be a problem since it is not expected to have many tenants requiring the same attributes.

3.2.2 API requirements

It is important to mention that for the metadata database illustrated in figure 3.2, the TimescaleDB instance is used for metadata and timeseries data storage.

The requirement list aimed at these APIs is the following:

- There must be tenant support in the Load API (LoaderRestApi.java), this means adding the tenant_id using the tag HEADERS of the Json Body of the HTTP request.
- When creating a dimension or fact table using the metadata management API (MetadataRestApi.java) it must be possible to specify if that table must be multi-tenant or not. For this, the boolean value represented by the tag "multitenancy" inside the payload of the Json body of the request is added. If this value is true the column *tenant_id* is added to the table in question.
- It must be possible to manage tenants inside each timeseries database instance. For this, endpoints to for CRUD operations must be created in the metadata management API (MetadataRestApi.java), to allow the management of the Tenants in the timeseries database.

- Invalid relations between different tenant data must be avoided. This means that rows from a fact table that belong to the tenant with ID 1, must not reference rows of a dimension that belong to the tenant with ID 2. To avoid this, the tenant_id value must be associated with the dimension table key.
- It must be possible to filter search queries for the SubQuery API (ServiceRestApi.java), by tenant_id. For this, the tenant_id value is passed as pathParam of the endpoint request. With the previous requirement fulfilled, the current one corresponds to adding to the "WHERE" statement of the query the clause *"where tenant_id = x"* where x is some tenant_id value.

3.3 Restrictions

In this section, the technical restrictions are presented. Such restrictions limit the flexibility of the project regarding the final result and overall architecture.

3.3.1 Technical Restrictions

The technical restrictions identified come from the fact that this project is based on another ongoing project from Altice Labs (ALB). The restrictions are:

- Docker must be used for running the system;
- Java must be used as a programing language;
- For the metadata database TimesacleDB must be used, for the timeseries databases some flexibility is allowed;
- The solution must be in compliance with the Altice Labs (ALB) collaborator goals, some flexibility is allowed;

Chapter 4

Architecture of Alexa Microservice

This Chapter describes Alexa's intended architecture, using the C4 model, after the porting process is complete.

4.1 C4 Diagram

The C4 model is comprised of 4 diagrams that go to different levels of detail of the architecture.

4.1.1 System Context Diagram

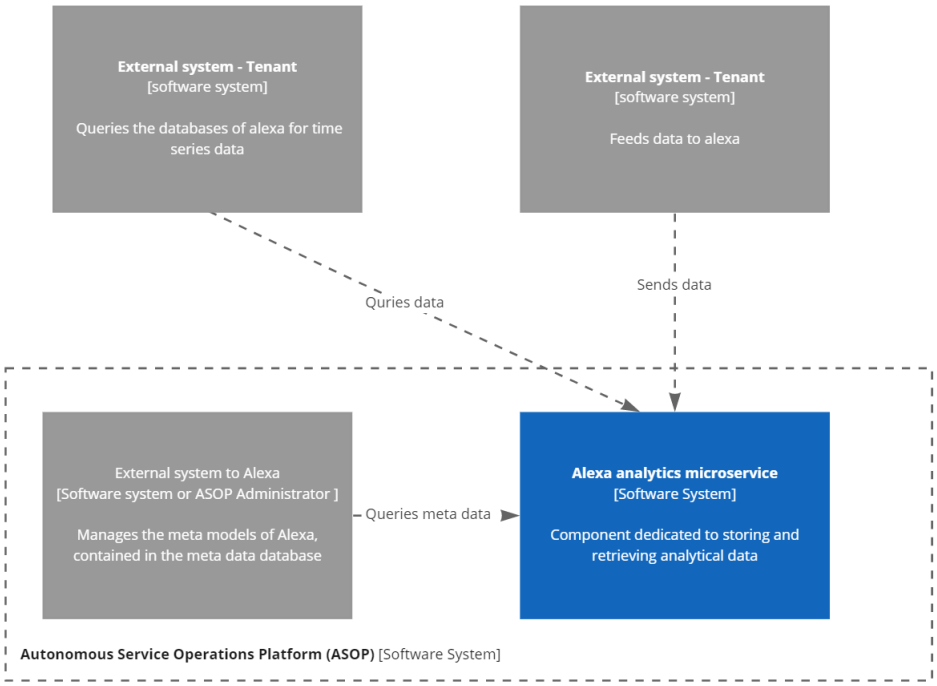

Figure 4.1: Context Diagram

This diagram shows three external actors, two are external to the Autonomous Service Operations Platform (ASOP) and represent tenants and other systems, and a third actor that is external to Alexa but internal to ASOP that can represent another service of the platform or an actual person like an administrator. This third actor executes Data Manipulation Language (DML) and Data Definition Language (DDL) operations in the metadata database, and the two external systems, query and send data respectively, to the timeseries databases.

4.1.2 Container Diagram

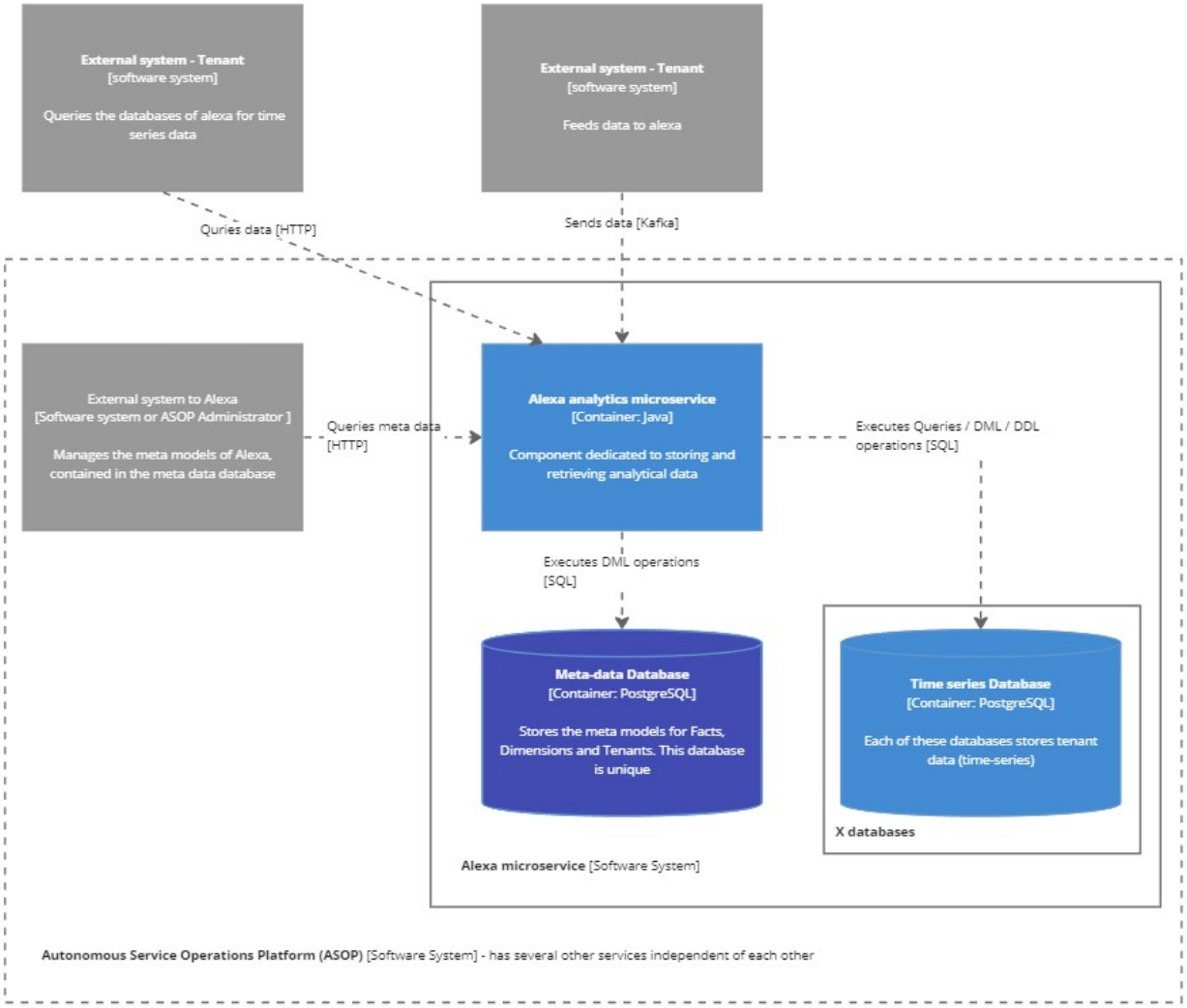

Figure 4.2: Container Diagram

This diagram goes into detail on the Alexa microservice, it shows what type of communications are done and where the metadata and timeseries data are stored.

The communication between external systems and Alexa is done in two ways: via API REST endpoints and Kafka topics, as demonstrated in the di-

agram. Between the microservice and the databases, Data Manipulation Language (DML) operations (like insert, update, delete, joins, etc.), Data Definition Language (DDL) operations (like create table, alter table, etc.), and queries, are executed via SQL.

For storage, this microservice uses several databases. One of these databases holds all the metadata needed for the Fact entity, Dimension entity, and Tenant entity, and then there are as many timeseries databases as needed to store time-series data represented by the (X databases in the container diagram).

4.1.3 Component Diagram

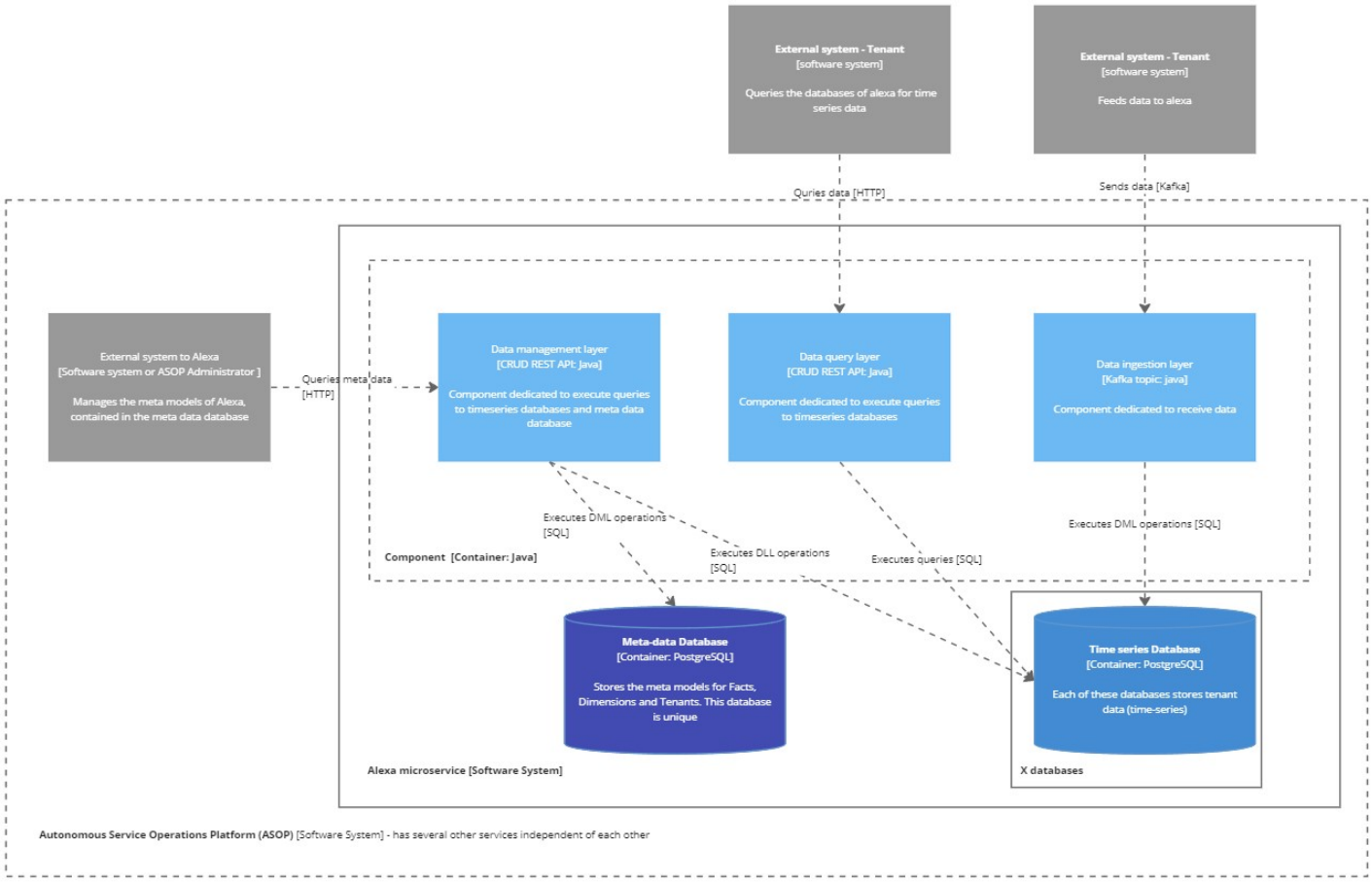

Figure 4.3: Component Diagram

This diagram details the architecture of Alexa at the level of its components. It is composed of three components or layers, one for data ingestion via Kafka topics, one for data management via REST API, and one for querying data also via REST API.

4.1.4 Code Diagram

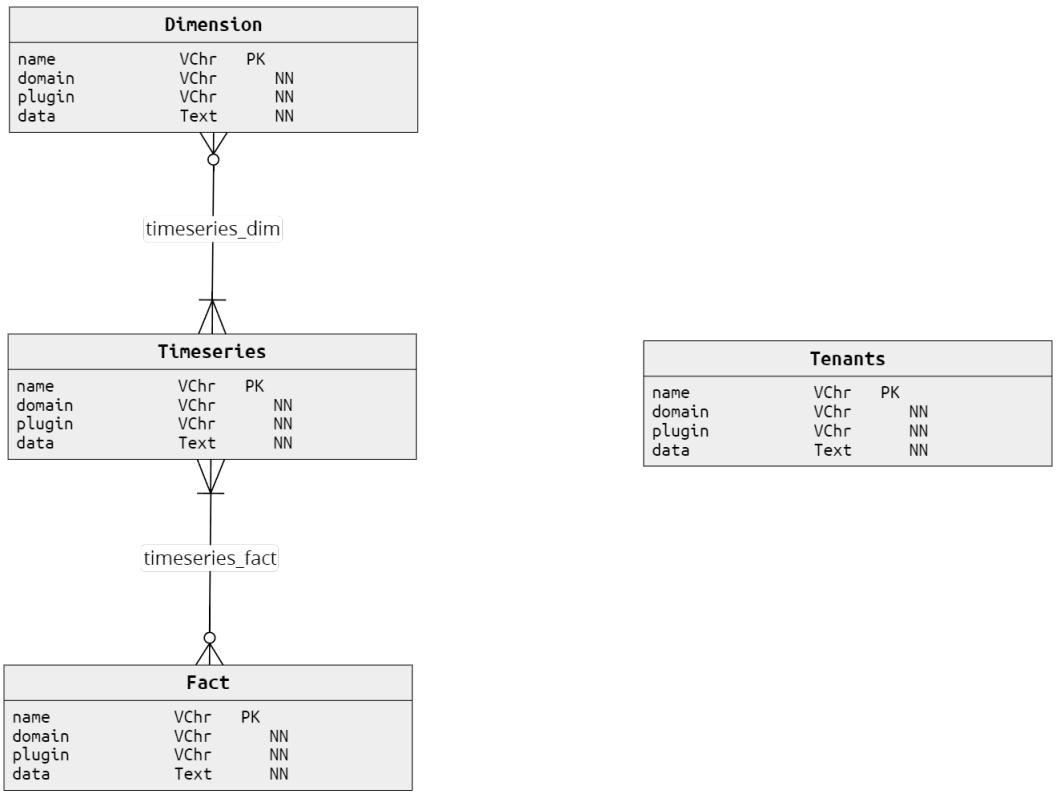

Figure 4.4: Code Diagram

The code diagram shows in detail the architecture of the metadata database component, there exists a many-to-many relationship between all the entities excluding the tenant entity. In all the entities, the *name* is the primary Key that uniquely identifies each entry of an entity in the metadata database, the *domain* is a tag used for characterizing the tables, the *plugin* indicates in what instance of timeseries database the entity is located, and the *data* attribute holds all the actual metadata in Json format.

This will be all the columns, the variable type of the values of each column (int, varchar, boolean, etc.), to what other tables is the table in question connected, and any other variables required. In the case of the tenant entities the *data* attribute holds among other data the tenant_id and the name of an individual tenant that uniquely identifies it inside a certain timeseries database instance.

Up until the C4 component diagram, this architecture is identical to the one explained in Chapter 4, Section 4.1, this is because the changes to implement on the systems are aimed at the code level.

4.2 Alexa Microservice final state

In this section I list the changes implemented in the resulting state of the Alexa microservice. The list is as follows:

- The first change implemented was in the metadata database, where a tenant model was created using the existing table template to create the Tenant table and make Tenants explicit.
- Also in this database I added the *tenant_id* column to the fact and dimension tables for the data separation at the row level.
- Tenant support was added to the Load API (LoaderRestApi.java) using the tag HEADERS of the Json Body from the HTTP request.
- In the metadata management API (MetadataRestApi.java), the boolean value represented by the tag "multitenancy", inside the payload of the Json body of the request was added, to specify during the creation of a dimension or fact table, if such table must be multi-tenant or not. When the value is true the column *tenant_id* is added to the created table.
- In the metadata management API (MetadataRestApi.java) four endpoints were added to manipulate Tenants, allowing the create, delete, update, and read operations on tenants.
- In order to prevent invalid relations between tenant-specific data in dimension tables and fact tables, the "tenant_id" value is added to the dimension key during the loading process of dimensions. This way when a row with data for tenant 2 is added to a dimension, and the dimension table already has that data but it is owned by tenant 1, the row will be inserted in the table but the value in the *tenant_id* column will be 2. This means that there will be replication of data, but when a new entry for tenant 2 is added to the fact table, that requires this row, it will connect to the dimension row that belongs to tenant 2, instead of connecting to the row owned by tenant 1.
- To filter the queries executed through the SubQuery API (ServiceRestApi.java) the clause *"where tenant_id = x"* was added to the query, where x corresponds to the tenant_id value.

Chapter 5

Multi-tenancy analysis

This section is dedicated to the analysis on multi tenancy. The objective is to analyze different approaches of implementing multi-tenancy in data warehouse databases comparing some of the critical aspects presented in the state of the art of this document, and again in more detail in the next section bellow.

5.1 Analysis introduction

After the research phase of the first semester, it became apparent that developing a porting tool within the initial project parameters would be extremely complex and difficult with the available time, due to the amount of technical aspects needed to take into account.

As a result, after a few months of working on porting the Alexa system from single-tenant to multi-tenant, which was also part of the work plan of the project, I decided to downsize the original objective by reducing the span of systems that the porting tool is aimed at. This meant reducing the number of parameters involved in developing such a tool.

With this in mind, I wanted to analyze different approaches to implementing multi-tenancy in data warehouse systems, to provide a better understanding of the options for developing a porting tool in future work. I set on analyzing three different approaches to multi-tenancy in a single data warehouse database.

These approaches serve the purpose of comparing throughput performance, and complexity of implementation of different levels of isolation, which are intended to help define the type of porting tool to develop and guide its development process. The ultimate goal is to have a set of guidelines that help develop this tool in a way that makes it easy to make it a restrictive tool aimed only at a small group of systems or make it a more generic tool capable of porting multiple different systems from single-tenant to multi-tenant.

The aspects I decided to analyze are related to performance, data isolation, and the complexity of different solutions of how to implement multi-tenancy in a

data warehouse database. For the first aspect mentioned, different execution scenarios trying to simulate real usage scenarios, are used to measure performance and see how much difference is there between the three approaches. It is expected that the first approach is the best performing in this field, but by how much? Will there be a big difference in performance compared to the other two approaches that make it worth considering in the porting tool development process, or not? And the other two approaches, will they have similar performance or not? Will there be a big difference between them? These are some of the questions that this analysis intends to answer. More information about test specification is provided in detail in section 6.

In regards to the other two aspects, I am focused on implementing different solutions of multi-tenancy with different levels of tenant isolation, to see how complicated the implementation process is, with the intent of gathering insights that help decide what types of systems should the porting tool be able to produce when porting single-tenant systems to multi-tenant systems. Figures 5.1 and 5.2, provide a visual representation of how these aspects are reflected in the approaches considered.

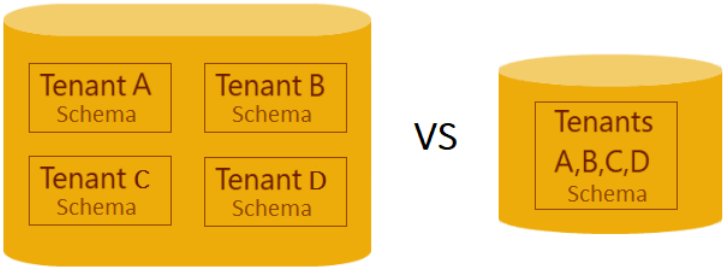

Figure 5.1: Number of schemas to use: one schema per tenant vs one schema for all tenants in the database

Fact or Dimension table

Attr1	Attr2	Attr3	tenant_id
x1	x2	x3	1
y1	y2	y3	2
z1	z3	z3	3

+

Fact or Dimension table

Attr1	Attr2	Attr3
x1	x2	x3
y1	y2	y3
z1	z3	z3

Figure 5.2: Data isolation at the table level: row level isolation mixed with no isolation

Note that in the context of this analysis, the term *isolation* does not refer to transactional isolation of queries, but rather to data isolation as demonstrated in figure 5.2.

5.2 Approach Definition

Moving on, the three approaches to multi-tenancy considered are:

- *First approach:* consists in using a separate schema for each tenant allowing for multiple tenants in the same database, while having total tenant separation, as figure 5.3 shows. This method also allows for easier tenant customization and manipulation, making it less complex to manage each tenant individually but may prove more complex to apply the same changes to all or several tenants because it does not allow scenarios where data is shared between tenants.

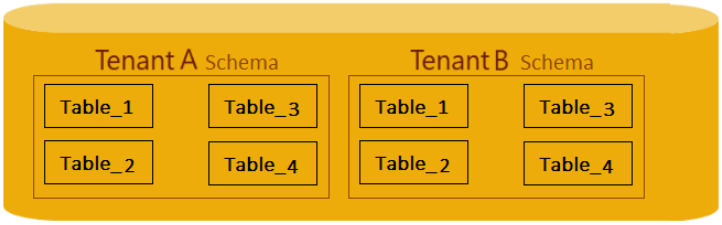

Figure 5.3: Approach 1 Diagram

- *Second approach:* consists in using a single schema for all tenants, where they are uniquely identified by a tenant_id stored in a separate table called TENANTS. With this, it is possible to separate data at row level, like in figure 5.2, where dimension and fact tables may or may not be multi-tenant. If a table is multi-tenant then it includes an extra column called "tenant_id".

  If this column is included it means that each row of the table will belong to a specific tenant identified by the corresponding id value in this column, and only this tenant will be able to query the specific row. If the table does not include this column, it means that there is no separation of data in the table, and every tenant can query every row. This approach provides a flexible framework that enables both unique and shared data scenarios, however, it is still a bit limited because it needs a separate table for each option, and it does not allow both in the same table. See figure 5.4 for a visual representation of this approach.

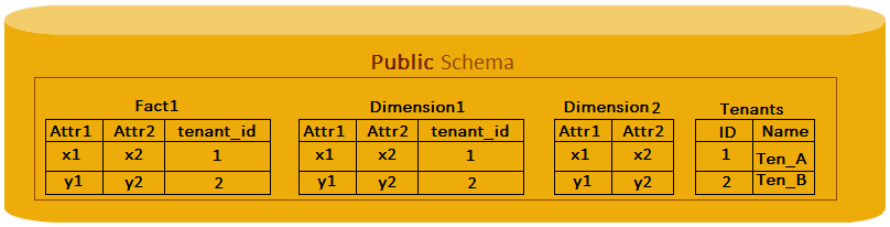

Figure 5.4: Approach 2 Diagram

- *Third approach:* consists in using a single schema for all tenants, where tenants are uniquely identified by a tenant_id stored in a separate table called TENANTS, like in the second approach, but one extra table called GROUPS is added here. This table allows the creation of tenant groups, where a relationship of many to many exists between tables TENANTS and GROUPS.

In this relationship, a group consists of one or more tenants, and a tenant can belong to one or several groups. These groups are also represented by the same unique identifier as the tenants, the tenant_id. This means that in the dimension and fact tables, the column "tenant_id" contains values that can represent a single tenant or a group of tenants.

By utilizing the GROUPS table, different levels of data sharing can be defined, allowing tenants to share specific data while maintaining isolation for others, based on individual business requirements. Moreover, it is possible to dynamically group tenants which facilitates seamless changes in tenant relationships, allowing situations like the creation of temporary joint ventures or specialized projects. See figure 5.5 for a visual representation of this approach.

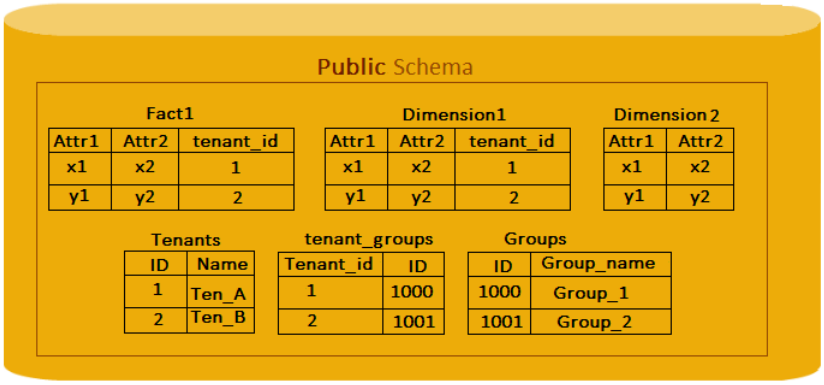

Figure 5.5: Approach 3 Diagram

A pivotal aspect to consider while developing the porting tool is the flexibility offered by different multi-tenancy approaches. The single schema approaches, whether it incorporates tenant groups or not, provide distinct advantages over the approach of using a separate schema for each tenant. Because they enable unique and shared data scenarios, they can better represent and fulfill the needs of complex business scenarios involving many different tenants.

Understanding these differences will enable the development of a porting tool that can accommodate a wider range of multi-tenant system requirements, maintaining the applicability in different business contexts intended with the original project idea.

To help visualize this analysis the figure 5.6 below depicts the three approaches considered, using a binary tree diagram.

5.3 Workload definition

In order to have a business representative data set I used the TPC Benchmark™ H [38], which is a decision support benchmark consisting of a group of business-oriented ad-hoc queries and data, that represent broad industry-wide relevance while maintaining a sufficient degree of ease of implementation. The

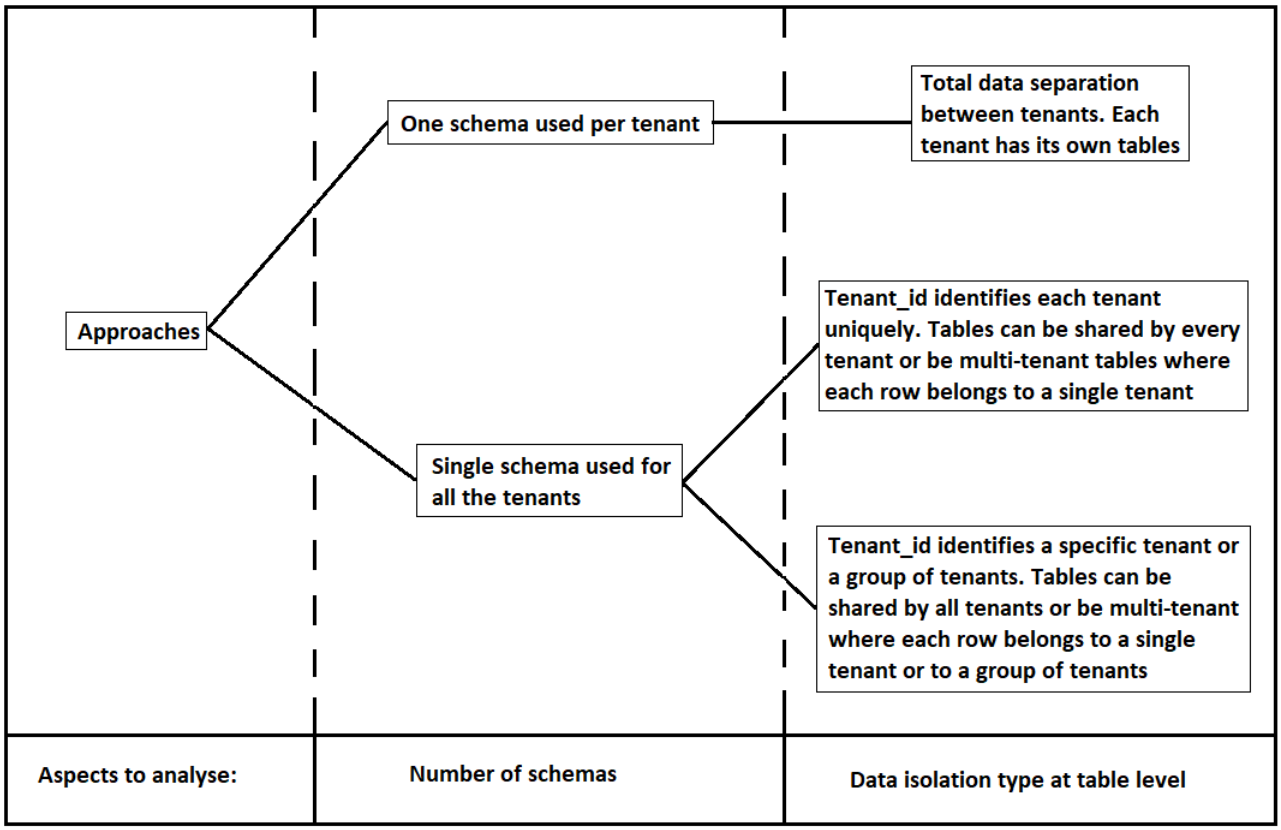

Figure 5.6: Analysis tree diagram

example used by the benchmark is the Wholesale business scenario, which represents any industry that must manage sell, or distribute a product worldwide.

This benchmark is relevant to the analysis because it is aimed at systems handling substantial data volumes and executing complex queries, both crucial aspects of the porting tool target systems. Also ensuring relevance for real industry scenarios, it enhances the decision-making process, steering future porting tool development.

The benchmark tool uses data generation, loading, and querying to gauge system performance. But for comparing the stated approaches, I use only the throughput tests from TPC-H Benchmark (TPC-H), using the inbuilt data generator (dbgen) for data sets and (qgen) for query generation. To ensure a fair comparison, the database size and data volume per tenant remain constant across all approaches.

The guidelines provided by this benchmark are used for the performance test.

5.4 Logical database desing

Next I will explain the database Entities, relationships, and characteristics, starting by the table creation process. To create the eight tables required for this

benchmark I used the provided structure for each table from the TPC-H documentation [39]. It contains the columns, primary and foreign key definitions for each table.

The Wholesale scenario contains eight tables organized in a snowflake schema. The fact table is called LINEITEM, and the dimensions are PART, PARTSUPP, ORDERS, CUSTOMER, NATION, REGION, and SUPPLIER. This schema is a multi-dimensional data model that extends the star schema model where dimension tables are normalized. Meaning that data in these tables are broken down into additional tables to eliminate redundancy and dependency, leading to more tables and more complexity, raising the applicability of the benchmark results.

With this structure in mind, the TPC-H benchmark integration into the snowflake schema is as follows:

- The fact Table is the LINEITEM table that contains keys to the dimension tables.
- Dimension tables linked to the LINEITEM table are:
  - ORDERS: Connected to the LINEITEM table via the orderkey.
  - CUSTOMER: Connected to the ORDERS table via the custkey.
  - SUPPLIER: Connected to the LINEITEM table via the suppkey.
  - PART: Connected to the LINEITEM table via the partkey.
  - PARTSUPP: This is a bridge table between the PART and SUPPLIER tables.
- The NATION and REGION tables connect to the CUSTOMER and SUPPLIER tables.

Each of these tables comprises multiple attributes whose details, with respect to their purpose in the TPC-H context, are as follows:

1. LINEITEM:
   l_quantity: The quantity of the product ordered.
   l_nextendedprice: The extended price of the product ordered (quantity times base price).
   l_ndiscount: The discount on the item ordered.
   l_ntax: The tax on the item ordered.
   l_nreturnflag: Flag to indicate if the item was returned.
   l_nlinestatus: The status of the item ordered.
   l_nshipdate: The shipping date for the item.
   l_ncommitdate: The date the order was committed.
   l_nreceiptdate: The date the item was received.
   l_nshipinstruct: The shipping instructions.
   l_nshipmode: The shipping mode.
   l_ncomment: The comment about the line item.

2. ORDERS:
   o_norderpriority: The priority of the order.
   o_nclerk: The clerk who processed the order.
   o_nshippriority: The shipping priority.
   o_ncomment: The comment about the order.
   o_norderdate: The date of the order.
   o_norderstatus: The status of the order.

3. CUSTOMER:
   c_nname: The customer's name.
   c_naddress: The customer's address.
   c_nnationkey: The key to reference the nation the customer belongs to.
   c_nphone: The customer's phone number.
   c_nacctbal: The customer's account balance.
   c_nmktsegment: The market segment the customer belongs to.
   c_ncomment: The comment about the customer.

4. SUPPLIER:
   s_nname: The supplier's name.
   s_naddress: The supplier's address.
   s_nnationkey: The key to reference the nation the supplier belongs to.
   s_nphone: The supplier's phone number.
   s_nacctbal: The supplier's account balance.
   s_ncomment: The comment about the supplier.

5. PART:
   p_nname: The part's name.
   p_nmfgr: The manufacturer of the part.
   p_nbrand: The brand of the part.
   p_ntype: The type of the part.
   p_nsize: The size of the part.
   p_ncontainer: The container in which the part is shipped.
   p_nretailprice: The retail price of the part.
   p_ncomment: The comment about the part.

6. PARTSUPP:
   ps_nsupplycost: The cost of the supply.
   ps_navailqty: The quantity of parts available.
   ps_ncomment: The comment about the part supply.

7. NATION:
   n_nname: The name of the nation.
   n_ncomment: The comment about the nation.

8. REGION:
   r_nname: The name of the region.
   r_ncomment: The comment about the region.

All comment columns in these tables serve as random text strings to add realism to the database size and to test text manipulation performance. Figure 5.7 below shows the conceptual diagram of the first approach.

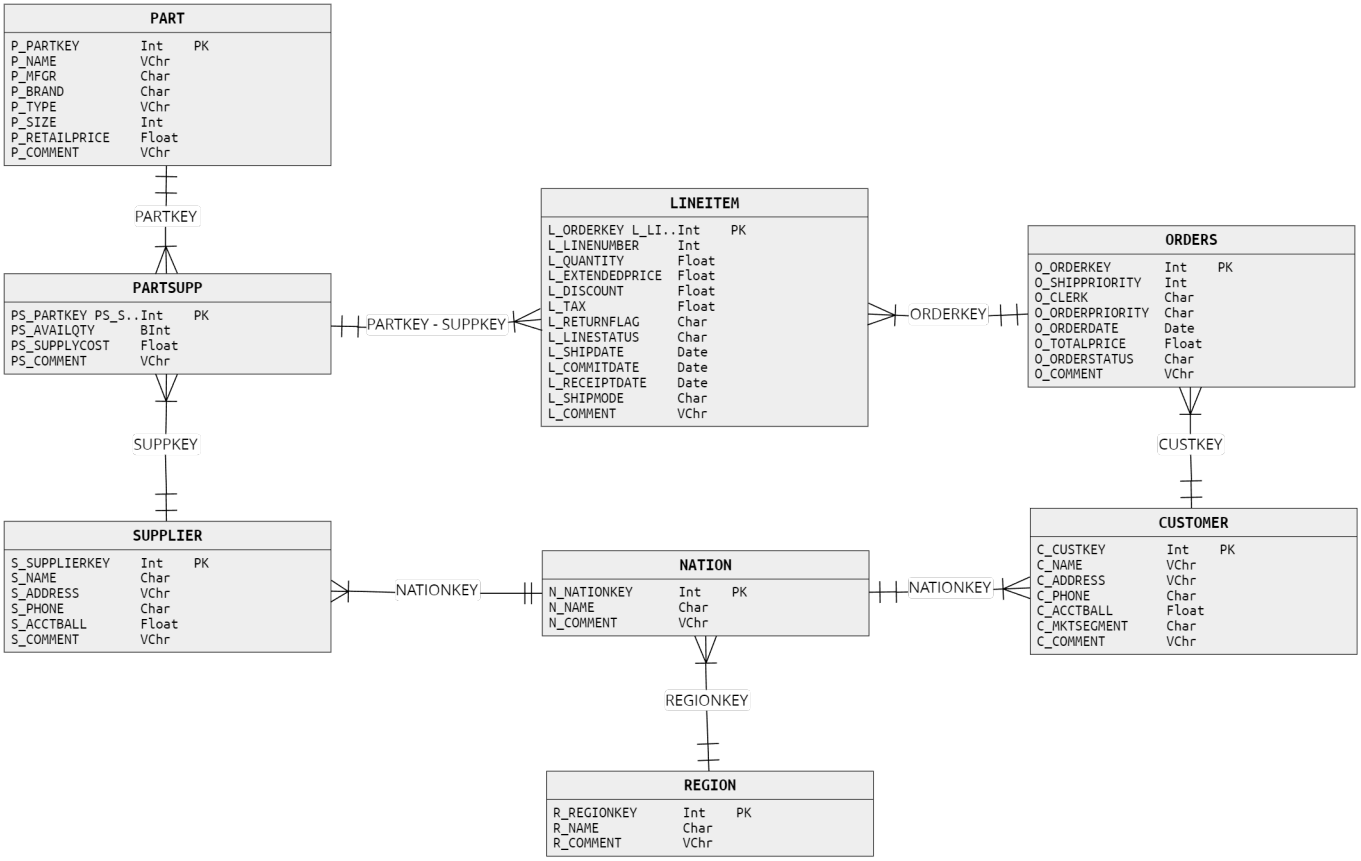

Figure 5.7: Approach 1 Conceptual Diagram

The relationships between tables is the following:

- REGION and NATION tables, one region can have multiple nations, and many nations belong to one region. For instance, the 'Europe' region can include nations such as France, Germany, Italy, etc;
- SUPPLIER and NATION tables, one supplier belongs to one nation, and a nation can have many suppliers;
- NATION and CUSTOMER tables, one nation can have multiple customers, but a customer belogs to one nation only. For instance, France can have many customers, but a customer can only live in one nation.
- ORDERS and CUSTOMER tables, each order is associated with one customer, and one customer can have multiple orders. For example, consider

three orders (Order1, Order2, Order3) all three orders could be associated with one customer (CustomerA), but no order can be associated with multiple customers.

- SUPPLIER and PARTSUPP tables, one supplier can supply many parts, but a part can only be supplied by one supplier. Consider the situation where a supplier, say "ACME Supplies", provides various parts (e.g., widgets, gadgets, sprockets). Each of these parts will have an entry in the PARTSUPP table that refers back to "ACME Supplies" in the SUPPLIER table.

- PART and PARTSUPP tables, one part can be supplied by multiple suppliers, hence there can be multiple entries in the PARTSUPP table for one part, on the other hand, many entries in the PARTSUPP table can refer to the same part in the PART table.

  For example, consider a part with partkey 'P1'. This part might be supplied by multiple suppliers (Supplier S1, Supplier S2, Supplier S3). So, in the PARTSUPP table, there will be multiple entries like: partkey 'P1', suppkey 'S1'; partkey 'P1', suppkey 'S2'; partkey 'P1', suppkey 'S3'.

  In this case, there is one part 'P1' that corresponds to many entries in the PARTSUPP table, which illustrates the One-to-Many relationship from PART to PARTSUPP. Conversely, multiple entries in the PARTSUPP table refer to the same part in the PART table, which illustrates the Many-to-One relationship from PARTSUPP to PART.

- PARTSUPP and LINEITEM tables, one record in the PARTSUPP table can be associated with multiple records in the LINEITEM table, and each record in the LINEITEM table can only be associated with one record in the PARTSUPP table.

  For example, a specific combination of a part and supplier (identified by a unique partkey and suppkey in PARTSUPP table) may supply several line items (LINEITEM table) in multiple orders.

  Note that there isn't a direct foreign key relationship from the LINEITEM to the PARTSUPP in the TPC-H schema. The relationship is established through the combination of the partkey and suppkey in both the LINEITEM and PARTSUPP tables.

- ORDERS and LINEITEM tables, one order can have multiple line items associated with it, and each line item can only belong to one order.

  For example, if an order is placed for multiple different items (e.g., a book, a pen, and a notepad), each of these items would represent a different line item in that order, as in the other way, each of these line items (the book, the pen, and the notepad) would all link back to the same single order.

From the previous conceptual diagram the resulting physical diagram is presented in figure B.1.

The figures 5.8 and 5.9 show the corresponding conceptual diagram for approaches 2 and 3 respectively.

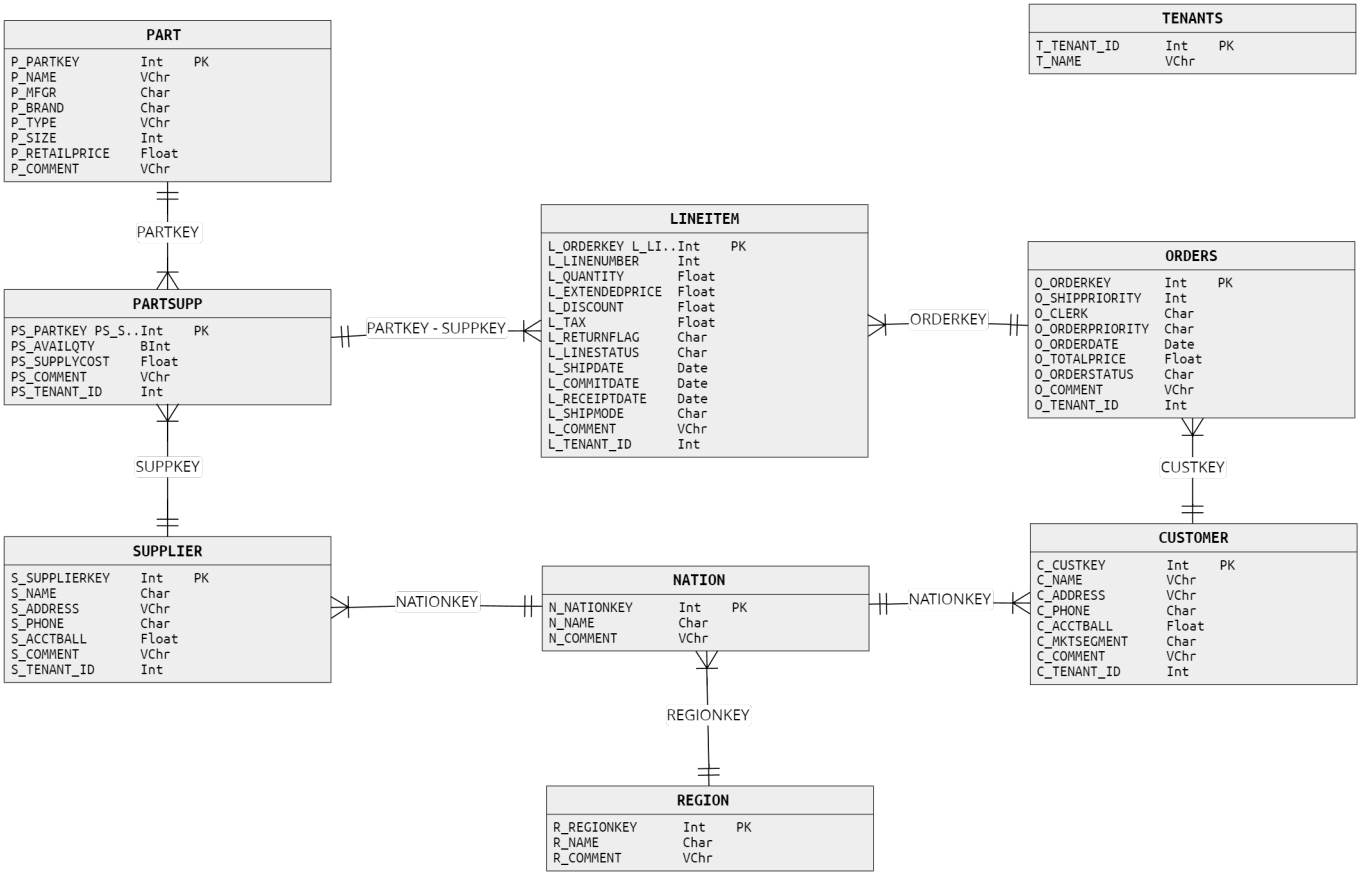

Figure 5.8: Approach 2 Conceptual Diagram

This wholesale model is applicable to approach 1, but for approaches 2 and 3, the model had to change to allow multi-tenant support. This change added the column "tenant_id" to the tables that directly relate to business operations like ORDERS, LINEITEM, PARTSUPP, CUSTOMER, SUPPLIER, and PART tables. These tables represent entities that differ from tenant to tenant in most scenarios.

The NATION and REGION tables contain geographical data common across multiple tenants and therefore do not include this column. For example, America or Europe are usually the same for all tenants, so it does not make sense to have a tenant_id attached to these entries.

After adding the column, two extra tables, TENANTS and GROUPS, were added. Table TENANTS was added in approaches 2 and 3, and table GROUPS only in approach 3. In the latter, there is a many-to-many relationship between tables TENANTS and GROUPS, where a single tenant can belong to many groups, and each group must have at least one or more tenants, as described in section 5.1.

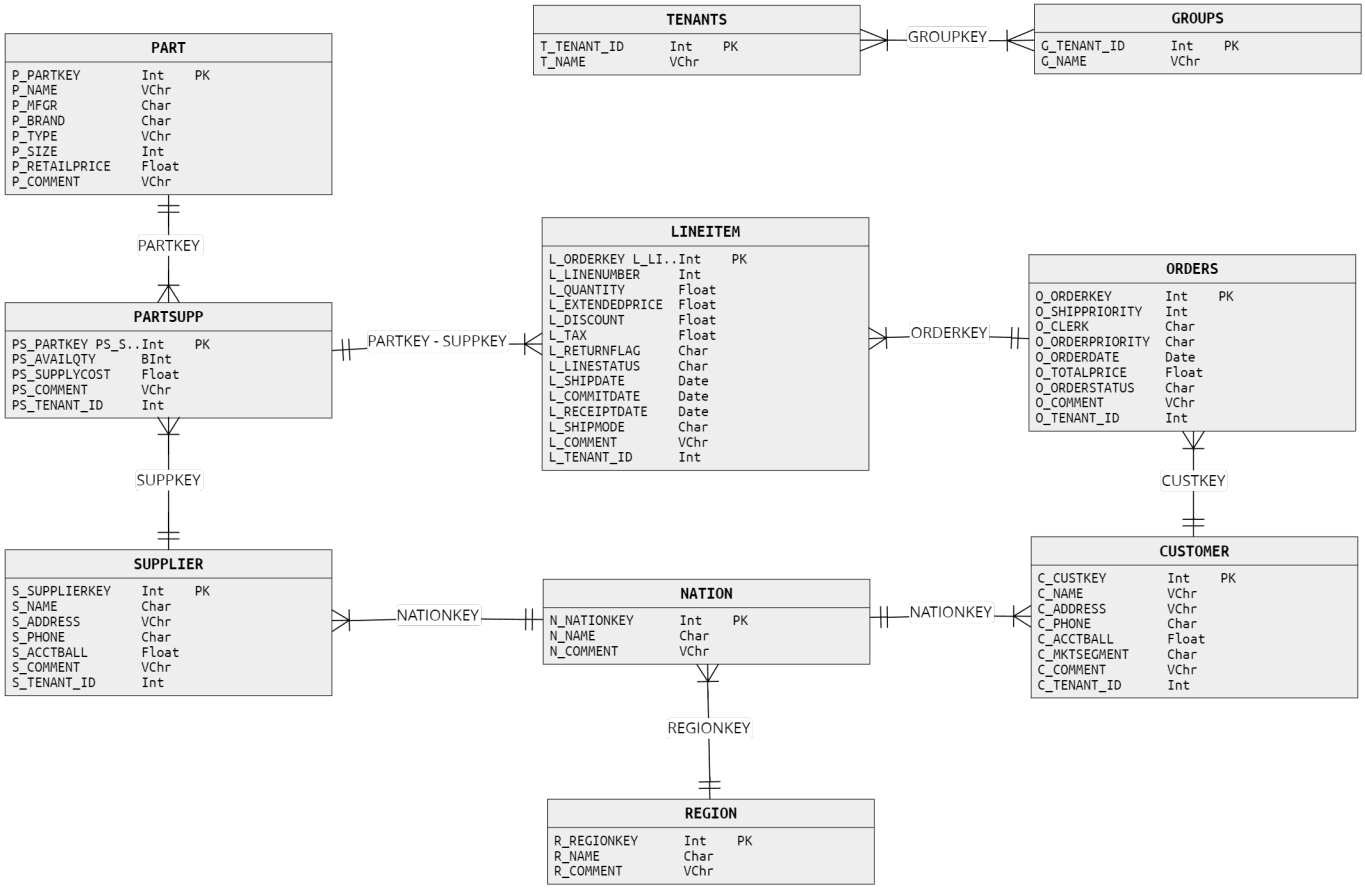

Figure 5.9: Approach 3 Conceptual Diagram

5.5 Query definitions

The TPC-H Benchmark provides a set of twenty-two queries used for the throughput tests in all three approaches. To avoid database caching during the tests, query variants generated with the qgen file introduce diversity into the query set. The qgen file uses randomly selected values from a uniform distribution, over the range of list of values specified in the dist.dss file, to generate variants of the twenty-two queries. Both qgen and dist.dss files are provided in the TPC-H Benchmark official website [38].

The list below shows a brief description of all the queries of the query set, but the full set is presented in detail in appendix C, and the complete query set is available in my public GitHub repository [37].

- Q1 - Pricing Summary Report Query reports the amount of business that was billed, shipped, and returned.
- Q2 - Minimum Cost Supplier Query finds which supplier should be selected to place an order for a given part in a given region.
- Q3 - Shipping Priority Query retrieves the 10 unshipped orders with the highest value.

- Q4 - Order Priority Checking Query determines how well the order priority system is working and gives an assessment of customer satisfaction.
- Q5 - Local Supplier Volume Query lists the revenue volume done through local suppliers.
- Q6 - Forecasting Revenue Change Query quantifies the amount of revenue increase that would have resulted from eliminating certain company-wide discounts in a given percentage range in a given year. Asking this type of "what if" query can be used to look for ways to increase revenues.
- Q7 - Volume Shipping Query determines the value of goods shipped between certain nations to help in the re-negotiation of shipping contracts.
- Q8 - National Market Share Query determines how the market share of a given nation within a given region has changed over two years for a given part type.
- Q9 - Product Type Profit Measure Query determines how much profit is made on a given line of parts, broken out by supplier nation and year.
- Q10 - Returned Item Reporting Query identifies customers who might be having problems with the parts that are shipped to them.
- Q11 - Important Stock Identification Query finds the most important subset of suppliers' stock in a given nation.
- Q12 - Shipping Modes and Order Priority Query determines whether selecting less expensive modes of shipping is negatively affecting the critical priority orders by causing more parts to be received by customers after the committed date.
- Q13 - Customer Distribution Query seeks relationships between customers and the size of their orders.
- Q14 - Promotion Effect Query monitors the market response to a promotion such as TV advertisements or a special campaign.
- Q15 - Top Supplier Query determines the top supplier so it can be rewarded, given more business, or identified for special recognition.
- Q16 - Parts/Supplier Relationship Query finds out how many suppliers can supply parts with given attributes. It might be used, for example, to determine whether there is sufficient number of suppliers for heavily ordered parts.
- Q17 - Small-Quantity-Order Revenue Query determines how much average yearly revenue would be lost if orders were no longer filled for small quantities of certain parts. This may reduce overhead expenses by concentrating sales on larger shipments.
- Q18 - Large Volume Customer Query ranks customers based on their having placed a large quantity order. Large quantity orders are defined as those orders whose total quantity is above a certain level.

- Q19 - Discounted Revenue Query reports the gross discounted revenue attributed to the sale of selected parts handled in a particular manner. This query is an example of code as might be produced programmatically by a data mining tool.
- Q20 - Potential Part Promotion Query identifies suppliers in a particular nation having selected parts that may be candidates for a promotional offer.
- Q21 - Suppliers Who Kept Orders Waiting Query identifies certain suppliers who were not able to ship required parts in a timely manner.
- Q22 - Global Sales Opportunity Query identifies geographies where there are customers who may be likely to make a purchase.

5.6 Performance test

The performance test executed in this analysis follows the guidelines proposed by the TPC-H benchmark. And according to the guidelines, the performance test is composed of two runs, where a run consists of one execution of the Power test, followed by one execution of the Throughput test.

The power test is intended to measure the raw query execution capabilities of the system when interacting with a single active user to understand how efficiently a single query can be processed when no other user is querying the database. The throughput test is intended to measure the capacity of the system to process the maximum number of queries in the shortest possible time.

A run is composed of one or more queries from the introduced default and variant queries. The sequential execution of a given set of queries is called a query set, and the sequential execution of query sets is called a query stream. A session is defined as the process context capable of supporting the execution of a query stream.

Moreover, there is also the use of refresh functions which are operations that simulate updates to the database, one for inserting new data entries, and another to delete data entries. The purpose is to simulate a workload that includes both querying and updating the database.

To automate the bench marking process by submitting queries to the database, recording the time taken for each query to complete, and measuring performance metrics the JMeter application is used.

At this stage it is mandatory to mention that in the tests executed query number 17 and query number 20 were left out of the final query sets, because of the extremely long execution times and the expensive hardware needed to execute them, resulting in high costs of service usage in the Amazon cloud platform, which conflicted with the budget allowed by the University. This decision was taken after the first round of tests on approach 1. The power test was executed in about 4 hours which was relatively fine, however, the four throughput tests started and did not complete, because the execution time was over 12 hours and

none of the tests was close to the halfway point, even after lowering the number of threads from 25 to 10.

Changing the number of queries does not affect the validity of the final results, as these queries are intended to represent real queries executed in systems of several different business scenarios, taking two queries out will simply reduce the applicability of the results, which seemed the best solution for this problem, taking into account the remaining time and budget available. This restriction implies editing the formulas used to calculate the performance and the next section describes how.

5.7 Metrics

To evaluate the performance the composite Query-per-Hour metric (QphH@Size) from TPC-H is used. To calculate this metric, two parameters are needed, one comes from the power test and the other from the throughput test. And to calculate these parameters the time measurements in milliseconds taken with JMeter during the tests are required, which means that the time is also a metric.

The first step is to calculate the TPC-H power parameter with a scale factor corresponding to the database size in GB, and the individual query execution times of the 22 queries logged by JMeter. Recall that these 22 queries are the 20 queries from the default set, excluding queries 17 and 20 due to the aforementioned, plus two queries representing the refresh functions. The formula used to calculate comes from the TPC-H documentation and the changes made to it are in the value of i which goes from 1 to 20, and the square root that changes to 22.

$$\text{Power@Size} = \frac{3600 \cdot SF}{\sqrt[22]{\prod_{i=1}^{20} QI(i,0) \cdot \prod_{j=1}^{2} RI(j,0)}} \tag{5.1}$$

Where:

- QI(i,0) is the timing interval, in seconds, of query Qi within the single query stream of the power test.
- RI(j,0) is the timing interval, in seconds, of refresh function RFj within the single query stream of the power test.
- Size is the database size chosen for the measurement and SF the corresponding scale factor.

With this, it is possible to compare the three approaches in terms of the raw performance of the database, or in other words how quickly it can execute a set of queries from a single session.

The other parameter needed is the TPC-H throughput numerical quantity which measures the total amount of work performed during the throughput test.

It also requires the time measurements taken in milliseconds by JMeter during the throughput test, and it is computed as the number of queries executed per hour during the throughput test (Qph) multiplied by the scale factor (SF) used during the test. It is reported as "Qth = SF * Qph" and its unit is queries. To calculate this the following formula is used, and it was also changed, where the scale factor (SF) multiplies by 20 instead of 22.

$$\text{Throughput@Size} = \frac{S \cdot 20 \cdot 3600}{Ts} \cdot SF \tag{5.2}$$

Where:

- Ts e the measurement interval, in seconds, and it starts either when the first character of the executable query text of the first query of the first query stream is submitted to the database by the driver, and it ends when the last character of output data from the last query of the last query stream is received by the driver from the database.
- S is the number of query streams used in the throughput test, in this case 4.
- SF is the scale factor, which is 30.

With these values, it is possible to calculate the TPC-H composite Query-Per-Hour performance metric, which is the primary metric of the TPC-H benchmark and represents a balance between the raw querying power of the database (Power Test) and its ability to handle multiple concurrent query streams (Throughput Test). To calculate it the following formula is used:

$$\text{QphH@Size} = \sqrt{Power@Size \cdot Throughput@Size} \tag{5.3}$$

Chapter 6

Analysis Experimental Setup

In this section I present in detail the the steps taken in the correct order, to perform this analysis, such steps are the database creation, database loading, data set generation, query set generation, and also the test plans in JMeter. The objective with this is to facilitate the implementation of the approaches considered, but also the replication of the analysis for further investigation.

6.1 Database creation

Starting with step one, database creation, for the databases I used the Amazon Relational Database Service (RDS) with a PostgreSQL instance, and for the client side I used Amazon Elastic Compute Cloud (EC2) machines. The database creation process itself, for all three approaches, was executed locally on my personal machine using a docker container with the latest version of Postgres, instead of directly in the RDS instances. The reason was to avoid possible loss of data due to internet connection failures, and also avoid unnecessary service costs. After the creation phase, came the upload phase, where I had to import the resulting databases of each approach to the RDS instances. For this, snapshots of such versions were taken from the docker console. See appendix D to view the commands used.

Here I want to mention that initially, the size of the database and the number of tenants generated were 100GB and 50 tenants respectively, however, after trying to execute the first test from the 18 prepared, (3 Power tests and 15 Throughput tests divided by 3 approaches), it was clear that such a large amount of data and number of tenants would result in very high costs in cloud services, and it would take considerable time to complete, as I realized after unfinished tests that took over 12 hours until crashing. Therefore the size of the database was reduced to 30GB and the number of tenants to 25. The database size was determined by the scale factor values determined by the TPC-H Benchmark guidelines, see appendix F.

To initialize the required database structure I used the script *create_and_load_DB.sql* that builds the 25 tenant schemas, with the eight tables

of the wholesale model. The SQL commands required for the tables and key constraints were obtained from the TPC-H Benchmark documentation [39], see appendix A to view the commands.

After creating the tables, the script also loads the data generated for the first approach. And due to limited disk space, the data was stored on an external hard drive, to then copy it into the database using the necessary SQL code in the script mentioned. Details about the data generation process come in the next subsection. Both the *docker-compose.yaml* file and the *create_and_load_DB.sql* script are available in my public GitHub repository [37].

6.2 Data Generation

In this section I explain how the 30GB data set used in the tests was obtained. The generation process consisted of using two auxiliary scripts built to manage and execute the provided *dbgen* and *dist.dss* files from the TPC-H Benchmark. The first script is called *tenant_folder_creator.py*, and generates 25 folders copying the files into each folder. The second script is called *table_data_gen.sh*, and it iterates over each folder executing the *dbgen* file inside it.

During the execution of *dbgen* a random data size between 512MB and 2048MB is selected, using a uniform distribution, to determine the amount of data to generate for each tenant. With each iteration, the size of data generated for the current tenant is subtracted from the total of 30720MB. All the scripts used for this step are available in my public Github repository [37].

6.3 Query generation

Up next is the query generation step, where the 22 default queries provided in the TPC-H Benchmark documentation [39] were used. However, these queries are not completed in the form available in the documentation, they contain template fields where the values generated by the provided *qgen* file, are inserted. To produce the query sets for the tests I used the script *query_set_generator.sh*, which iterates through the 22 default queries, using the iterator value as seed to generate random values for the template fields of each query, and creating 35 variants of each query. From this process, 35 different query sets were generated, one for each tenant and one for each group of tenants, for a total of 770 queries. After this, the sets still required some modifications to allow filtering data by tenant_id, such modifications are:

- In approach 1, the tenant schema has to be specified for a query to execute correctly, this means editing the *from* clause of the queries like this: " ... from tenant_19.<table_name> ... ".
- In approaches 2 and 3 to allow filtering queries by tenant_id another statement was added to the WHERE clause like this: " ... WHERE X_tenant_id =

<id> AND ... " where "X" represents the table initial, "l" for LINEITEM, "o" for ORDERS, "s" for SUPPLIER, "ps" for PARTSUPP, "p" for PART and "c" for CUSTOMER.

As for the refresh functions, two were generated, one for inserting a new entry in table REGION, and another to delete that entry. These functions were tested in the power test like the other queries, but in the throughput test, they were only used in the first query set corresponding to tenant 1. This was to avoid duplicate key constraint problems if multiple simulated users were to try and execute these queries because they are not tenant-specific. All the scripts used for this step are available in my public Github repository [37].

6.4 Data Loading

This next step is where I explain how the data set was loaded and then converted to be reused in approaches 2 and 3, to maintain data consistency. The generated data set for approach 1 was loaded during the database creation process. The *create_and_load_DB.sql* script iterates over the 25 tenant folders containing the generated table data, and in each iteration creates the respective schema and the tables, and proceeds to copy the data from the generated files into the database inside the docker container.

For approaches 2 and 3 this process was different because it consisted of adapting the data set used in approach 1. For this, I used the script called *create_sql_for_approach_2.sql*, which produces an SQL script called *adapt_DB_for_approach_2.sql* containing all the commands needed to migrate each tenant data from the corresponding tenant schema to the public database schema, and to create the table TENANTS.

The Python script can be explained with the following steps:

1. Creates a *line_counts_lineitem* dictionary where the keys are the tenants and the values the total number of rows in the corresponding LINEITEM table, obtained with the tbl files generated by dbgen. Example {tenant_1:3000000, tenant_2:7281267, ..., tenant_50:20938475}.

2. Initializes an *increment_var* with the row count of tenant_1. This variable is used to increment the primary and foreign keys of all tables.

3. Starts writing the *adapt_DB_for_approach_2.sql* script by adding the column *tenant_id* to tenant_1 populating it with the value 1.

4. Disables key constraints of tenant_1.

5. Iterates over the *line_counts_lineitem* dictionary writing for each tenant the commands to disable key constraints, incrementing primary and foreign

key values with the *increment_var*, adding the column *tenant_id* and populating it with the respective tenant id. After each iteration adds to the *increment_var* the row count of the LINEITEM table of the current tenant plus 100000.

This 100000 is used to account for the fact that for some tables like LINEITEM, dbgen does not linearly generate primary key values, there are "jumps" in the generation process, like so: 1, 2, 3, 4, 7, 8, 13, 20, ..., adding this extra value makes sure there are no overlaps in primary or foreign keys.

6. Writes the commands to create the eight tables of the wholesale model and the TENANTS table in the public schema.

7. Creates five dictionaries like the *line_counts_lineitem* dictionary, one for each of the following tables PART, SUPPLIER, CUSTOMER, PARTSUPP, and ORDERS. Tables REGION and NATION do not require this because they only have 5 and 25 rows respectively in every tenant.

8. Iterates over each dictionary, writing the commands to copy and then delete the rows of every tenant schema into the public schema.

9. Writes the commands to drop all the tenant schemas.

10. Writes the commands to create the table TENANTS with the columns *tenant_id* and *Name*, where the first is the primary key.

For approach number 3, I used the script called *create_sql_for_approach_3.sql*, that builds the SQL script called *adapt_DB_for_approach_3.sql*, and this python script can be described with the following steps:

1. Writes the commands to create the table GROUPS with the columns *g_group_id* and *g_Name*, where the first is the primary key.

2. Writes the commands to create the tenants_groups table to establish a many-to-many relationship between the tables TENANTS and GROUPS.

3. Creates a *tenant_groups* dictionary where the key is the group_id and the values are the list of tenants in that group.

4. Creates 10 groups by looping 10 times, while randomly selecting several tenants between 2 and 10, and randomly selecting tenants to insert in each group.

5. Writes the commands to populate the tables GROUPS and tenants_groups.

6. Creates the *lines_to_update_for_each_group* dictionary, where the keys are the group names and the values are the number of lines to update in the LINEITEM table.

7. Loops through the groups selecting a random number of lines between 1 line and 3% of the total lines in the LINEITEM table to update.

8. For each group, for each of the tables: ORDERS, PARTSUPP, SUPPLIER, CUSTOMER, and PART, selects 3% of the total lines in the respective table to update.

9. For each group divides the number of lines to update in each table by the total tenants included in the group.

10. Writes the commands to update the tables in the order: LINEITEM - ORDERS - PARTSUPP - CUSTOMER - PART - SUPPLIER, to avoid data inconsistencies, the updates are only made to the rows that are referenced by the foreign keys, and that have the tenant_id from the current tenant in the *tenant_id* column.

    This method of selecting group data was used because initially just 3% of the LINEITEM rows were allocated to each group, and the rows updated from the other tables connected to these LINEITEM updated rows, had no limit. This resulted in the majority of lines in the other tables to be updated, causing almost no tenant-specific data to be retained. With 3% of the total lines of each table updated for each group, a better balance between tenant-specific data and tenant-group data is achieved allowing me to analyse the impact of grouping tenants.

These scripts are all available in my public GitHub repository [37].

6.5 Performance test specifications

In this section I explain in detail the JMeter plans produced for the tests. As mentioned before, the client side consisted of EC2 machines, where JMeter was installed and from where the test plans were executed. A total of 5 machines were used all with the t3.micro CPU with 2 cores and 2 GB of RAM, and to connect to these instances I used SHH in VSCode. The full specification for these machines is available here [2]. For the database instances, I used 5 RDS instances with a t3.2xlarge CPU with 8 cores and 32GB of RAM, each instance contained the latest PostgreSQL image. The full specification and pricing are available here [3]. Next, I explain the test plans created in JMeter, step by step.

The power test corresponds to a single query stream with 22 queries, where 20 are query variants generated with *qgen* and the other two are refresh functions. Here, tenant number 19 was selected to be queried in all three approaches because it is the tenant with the most amount of data, and using the same tenant ensures test consistency. The timing intervals for each individual query are collected by JMeter.

1. The power test follows these steps below:

    A. Create a Thread Group with 1 thread to represent a single user.

    B. Set the thread group loop count to 1, so that the single user executes the 22 queries only once.

C. Add a JDBC request sampler for each query.

D. Add a JDBC connection configuration sampler to set the connection to the RDS database instance, providing the Database URL, server IP, port number, DB name, username, password, and the JDBC driver class name.

E. Add two listeners inside the thread group, a View Results Tree and a Summary Report to view the results and collect metrics.

The throughput test must be driven by queries submitted by a driver through sessions. The number of sessions depends on the scale factor of the database. The database total size is 30GB, therefore the scale factor is 30, which means that at least 4 sessions must be executed according to the benchmark guidelines. See appendix F.

Sessions serve to mimic the behavior of one or multiple users interacting with the system in a way that represents a specific usage scenario. With 25 tenants considered in the database, initially, the intent was to use 25 threads in each scenario, simulating 25 separate users querying each tenant's data. However, due to cost and time restrictions, the hardware used in the database instances did not support that amount of queries at the same time leading to only memory swaps taking place during execution, resulting in unending and expensive sessions. Therefore the number of threads was reduced to 10.

The scenarios are:

2. Steady Request Rate: This scenario represents a typical steady state of system usage, where tenants send queries at regular intervals. This is the most basic case of a multi-tenant system during regular business hours. To set it up I did the following steps:

   A. Create 25 Thread Groups with 1 thread each (representing 25 users) and disable thread groups from 11 to 25.

   B. Add a constant timer to the test plan with a two-second delay between each request to ensure that each thread sends requests every two seconds.

   C. Set the thread groups loop count to 1.

   D. Add a JDBC connection configuration sampler inside the test plan to set the connection to RDS database instance, providing the Database URL, server IP, port number, DB name, username, password, and the JDBC driver class name.

   E. Inside each thread group, add a Random Order Controller to randomize the order in which threads send requests with queries.

   F. Inside the Random Order Controller add one JDBC request sampler for each query variant of the respective query set, from the 25 query sets generated.

G. Add two listeners inside the test plan, a View Results Tree, and a Summary Report to view the results and collect metrics, specifying an output file to print the results.

3. Gradual Increase: This scenario models a situation where the tenant base within the system expands over time, leading to a gradual increase in the number of queries, with the threads representing tenants sending requests as soon as they are initiated.

   A. Create 25 Thread Groups with 1 thread each (representing 25 users) and disable thread groups from 2 to 16.

   B. Repeat the steps C. to G. from the scenario 2.

   C. In each thread group specify the start-up delay, dividing 300 seconds by 25 threads, which means that each thread starts within 12 seconds from the previous thread.

4. Spike Traffic: In this scenario, the system experiences a sudden influx of queries.

   A. Create 25 Thread Groups with 1 thread each (representing 25 users) and disable thread groups from 2 to 8 and from 18 to 25.

   B. Repeat the steps C. to G. from the scenario 2.

5. Random Traffic: This scenario represents a situation where the system receives a random number of queries at random times.

   A. Create 25 Thread Groups with 1 thread each (representing 25 users) and disable thread groups from 11 to 25.

   B. Repeat the steps C. to G. from the scenario 2.

   C. Add a Gaussian Random Timer inside the test plan to introduce a random delay between requests. Set the constant delay offset to 300 milliseconds, and the deviation to 150 milliseconds. This introduces delays to each user request that center around 300 milliseconds but will vary within the range of approximately 150 to 450 milliseconds of random delay with each user request. Mimicking a more realistic scenario of unevenly distributed requests.

In these tests thread 1 queries tenant 1 in the database, thread 2 queries tenant 2, and so on, this is why it is specified which threads are disabled in each test scenario. As mentioned before in this section, the number of threads active in each test was reduced from 25 to 10, and different sets of threads were selected in each test scenario to allow querying every tenant in the database.

For approach 3, the thread selection was done differently. Here the same thread-tenant relation exists for groups, in the sense that thread G1 queries group 1, and so on. With this in mind, the selected threads were, in scenario 1 threads

1 to 5 for groups and threads 1 to 5 for tenants. In scenario 2, threads 1 to 5 for groups and threads 1, 22, 23, 24, and 25 for tenants. In scenario 3, threads 6 to 10 for groups and threads 1, 14, 15, 16, and 17 for tenants. Finally, in scenario 4 threads 6 to 10 for groups and for tenants 1 to 5. The test plans created to execute in JMeter are accessible in my public GitHub repository [37].

Chapter 7

Experimental Results

In this chapter I present the graphical results of the tests, the calculations from the metrics used for the throughput performance, along with some conclusions and observations.

7.1 Graphical Overview of Test Results

This section shows the graphical test session results. The results of the refresh functions are displayed in separate graphics in each session, for better visualization of the throughput results. In the Power Test graphics and the refresh function graphics, the queries correspond to the execution of a single thread, but in the throughput graphics, the queries represent the average result from the execution of the 10 threads considered in each test session. The average is used to simplify the graphics and make them easier to analyze.

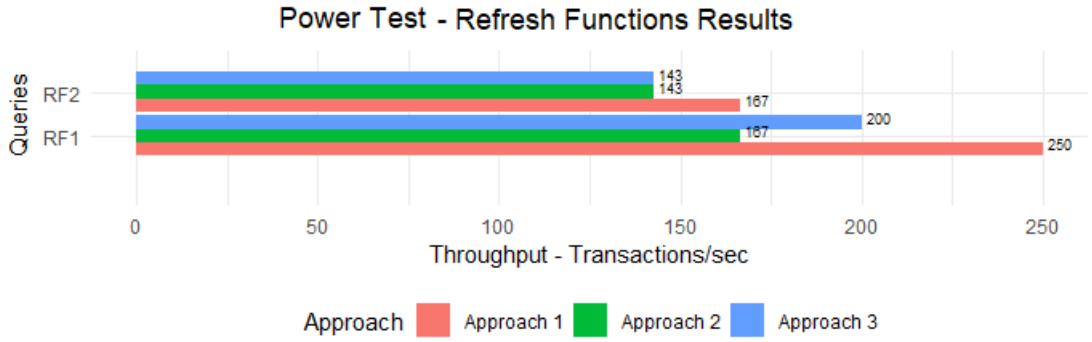

Figure 7.1: Power Test results for refresh functions.

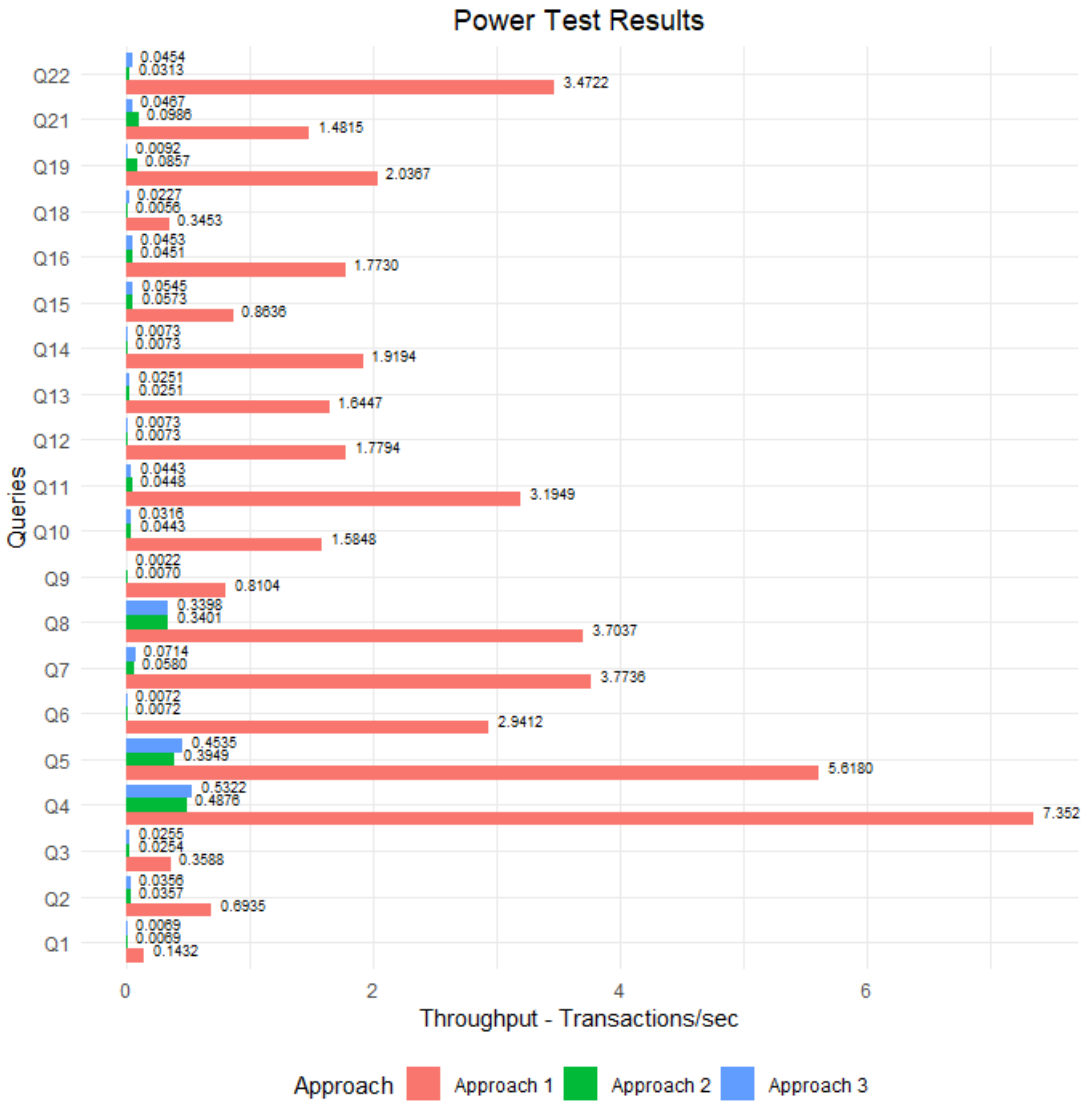

Figure 7.2: Power Test results for queries Q1 to Q22 excluding Q17 and Q20.

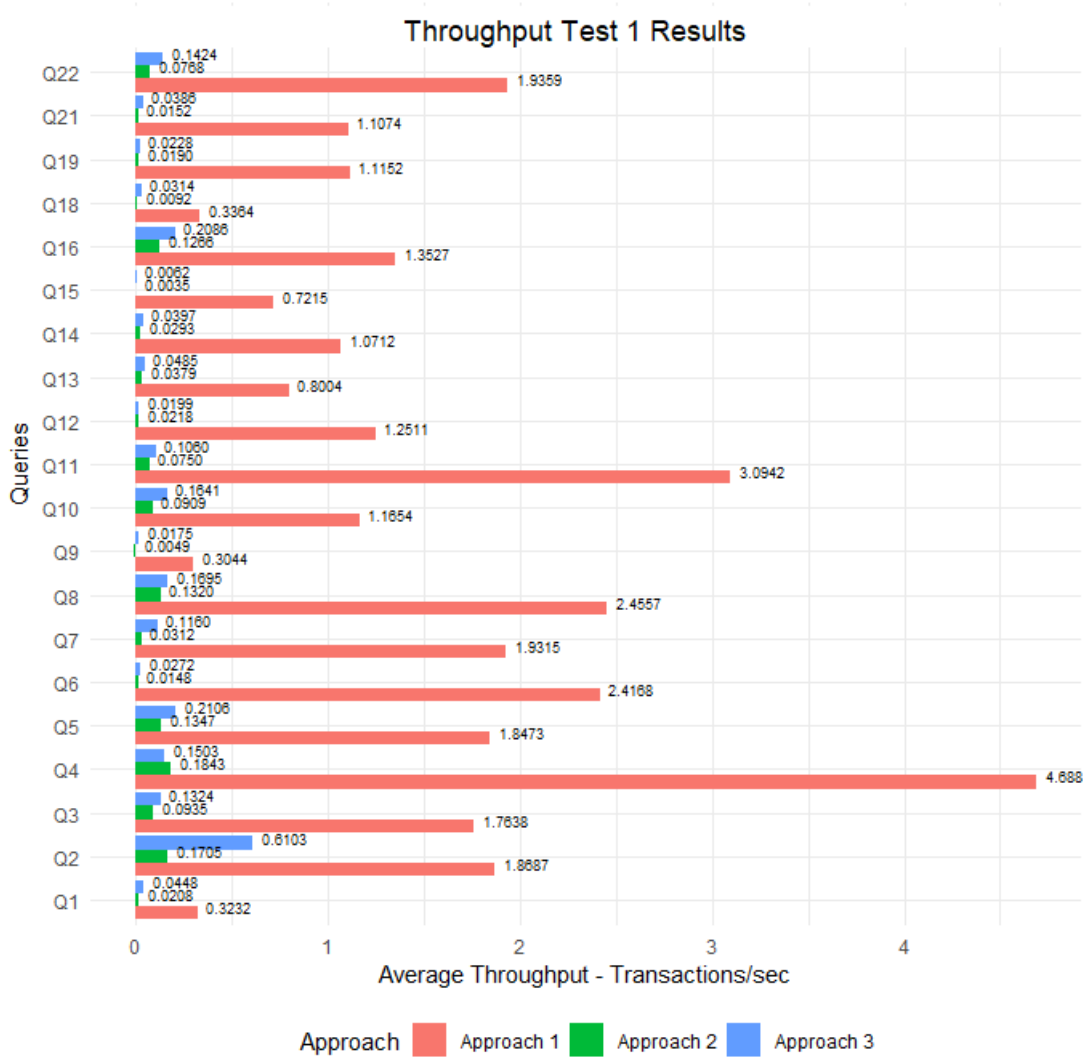

Figure 7.3: Steady Request Rate test results for queries Q1 to Q22 excluding Q17 and Q20.

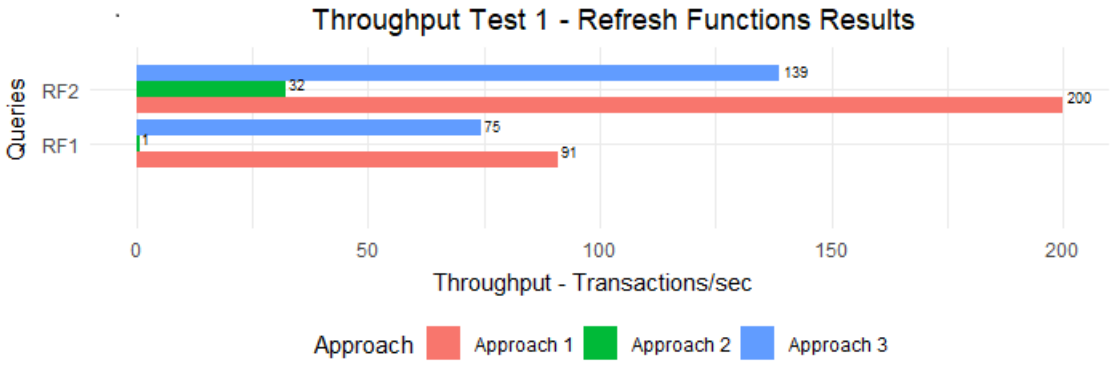

Figure 7.4: Steady Request Rate test results for refresh functions.

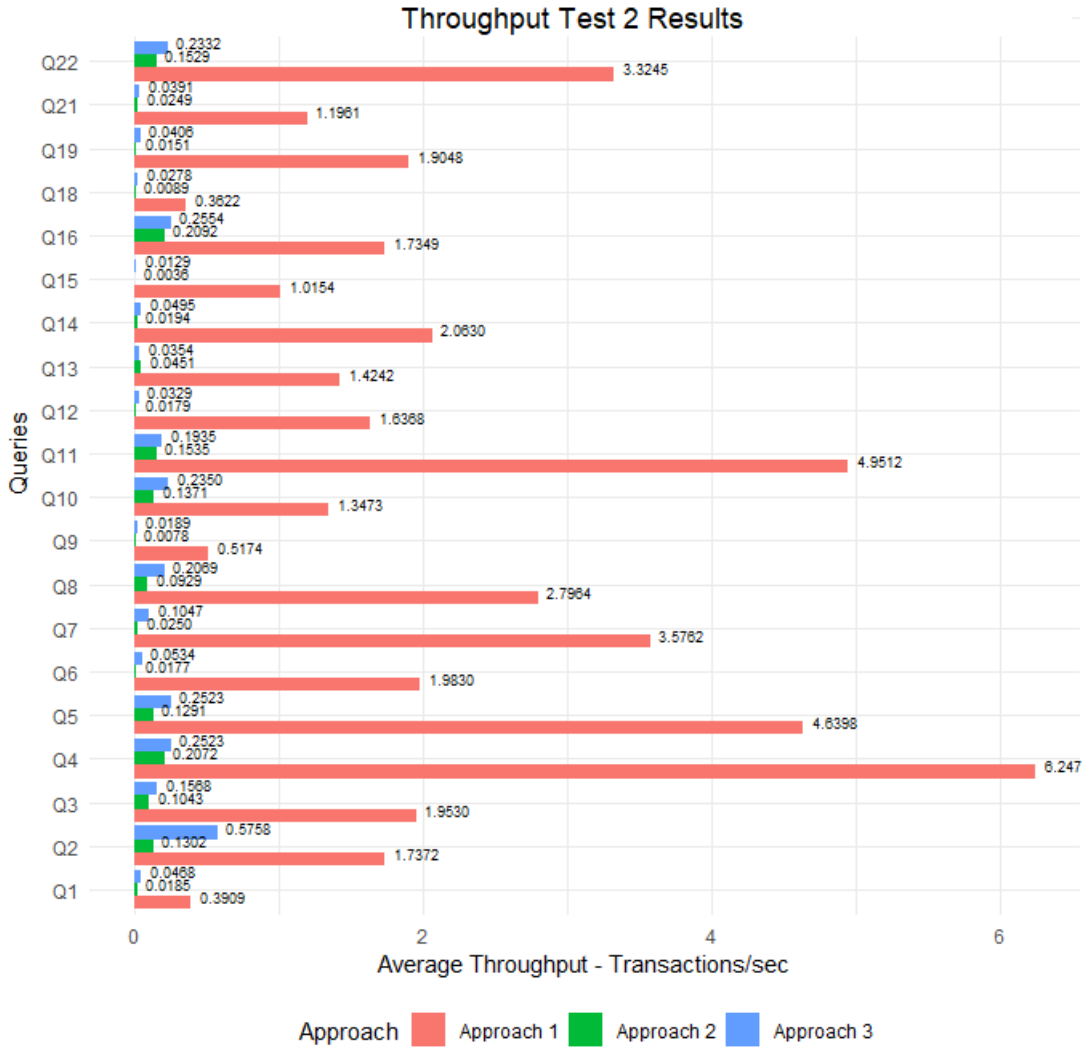

Figure 7.5: Gradual Increase test results for queries Q1 to Q22 excluding Q17 and Q20.

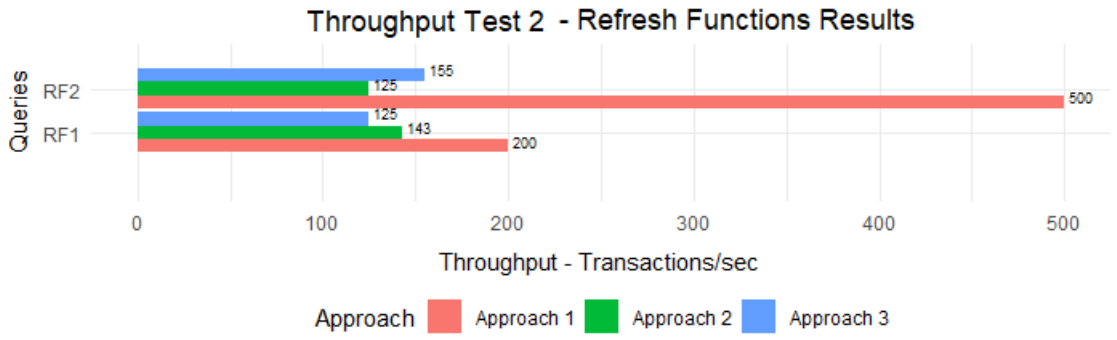

Figure 7.6: Gradual Increase test results for refresh functions.

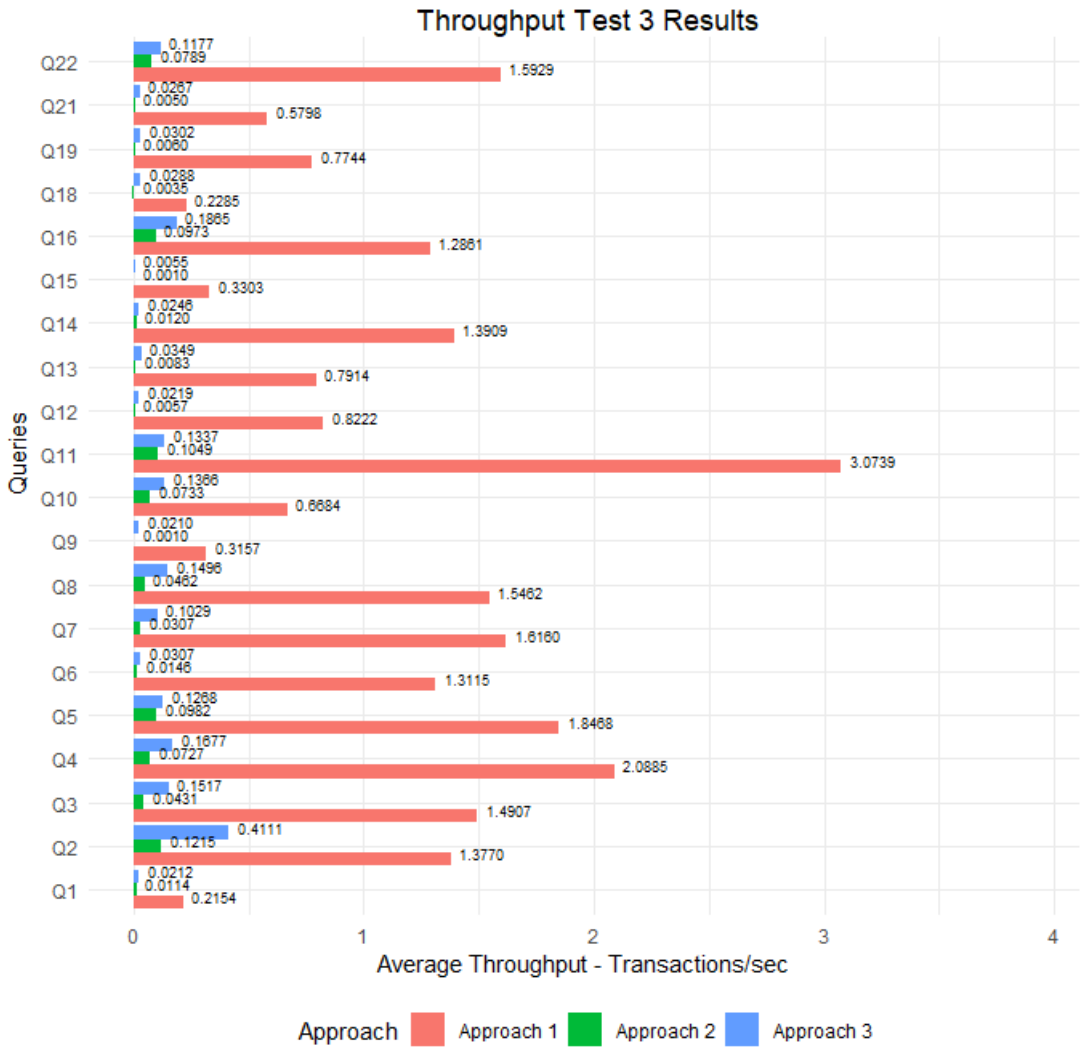

Figure 7.7: Spike Traffic test results for queries Q1 to Q22 excluding Q17 and Q20.

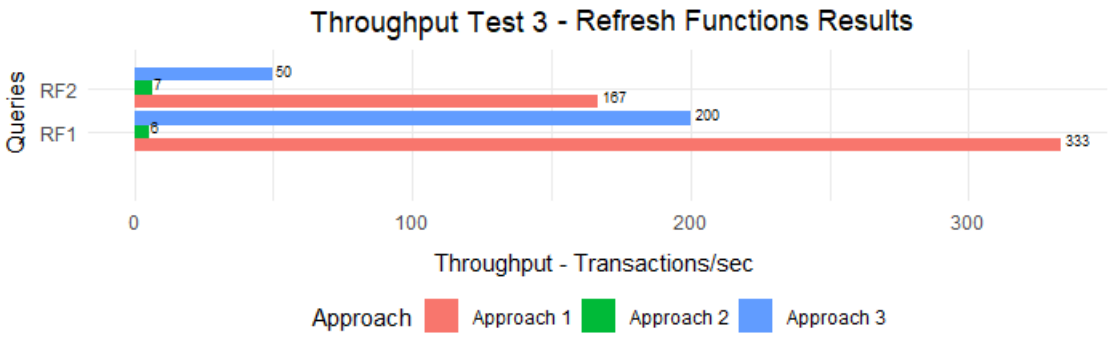

Figure 7.8: Spike Traffic test results for refresh functions.

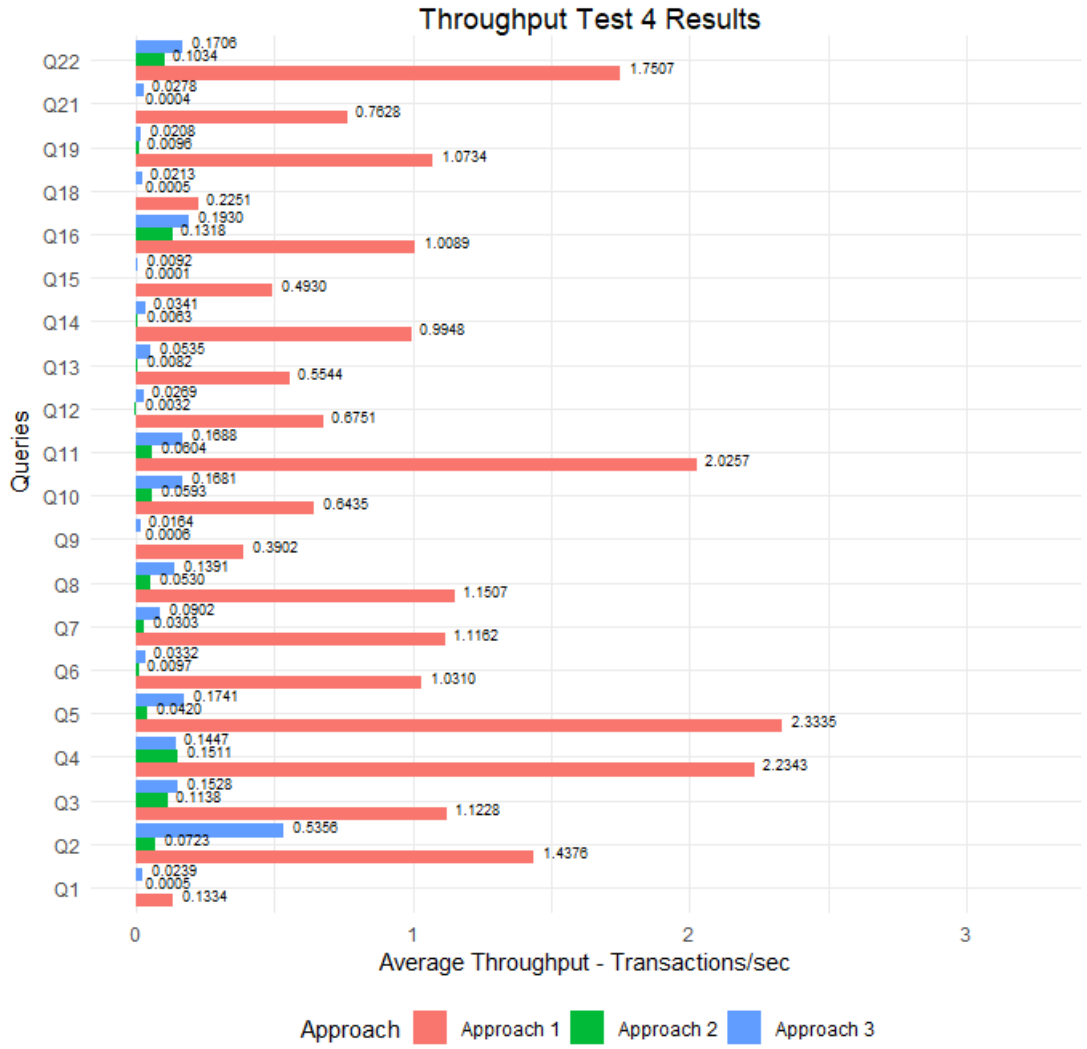

Figure 7.9: Random Traffic test results for queries Q1 to Q22 excluding Q17 and Q20.

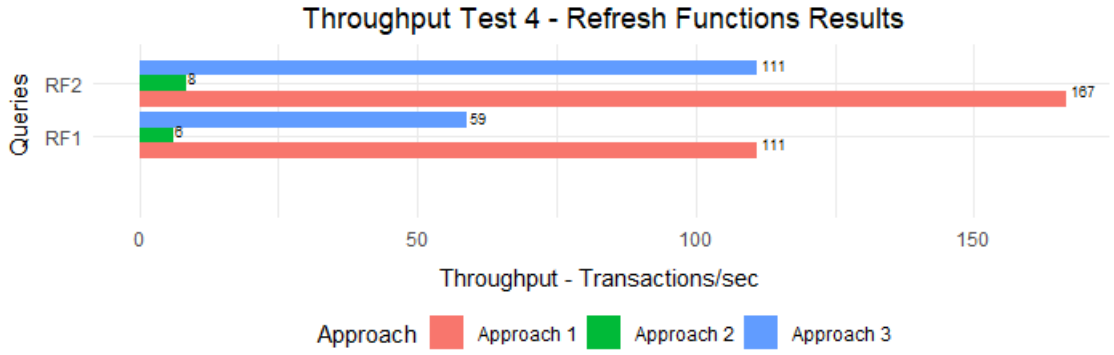

Figure 7.10: Random Traffic test results for refresh functions.

7.2 Performance analysis

In this section I analyze and compare the aspect of performance. The graphics show a better throughput performance from approach 1 in both the Power test and all Throughput test sessions and revealed that there is a very significant performance difference between approach 1 and the other two.

For example, analyzing Q4, which is the query with the biggest performance difference in 3 of the 5 test results, if we divide the throughput result of approach 1 by the result of approach 3, for all 5 tests, and then calculate the average of those divisions, the result is 19.48 times better in approach 1 than it is in the third. Redoing this to compare approach 1 with approach 2, the first has 22.78 times better throughput, and comparing approach 2 with approach 3 in the same way, the latter has 1.272 times better throughput performance. This suggests that the difference in performance is directly related to the size of the data queried.

Furthermore, the graphics also revealed that approach 3 almost always has better performance than the second approach, which tells that grouping tenants improves performance, by reducing the amount of tenant-specific data to search through during query processing. This means that the difference in performance will vary according to the amount of grouped data, where the throughput performance will always be better in approach 3 until the percentage of total grouped data reaches the point of diminishing returns, where the queries have to search through similar amounts of record as in approach 2.

However, there are some cases where this is not true, namely Q8, Q19, and Q21 from the Power test, Q12 and Q4 from the Throughput test 1, Q13 from the Throughput test 2, and Q4 from the Throughput test 4, where throughput is the same or slightly better in the second approach. This is most likely due to the amount of tenant-specific data queried in these specific queries, which corresponds to a tenant with more data.

Analyzing the refresh function graphics, the results verify the previous findings in terms of performance difference. However, here this difference is not so drastic between approaches 1 and 3. Doing the same calculation as before, for RF1, approach 1 is only 1.6 times better than approach 3 and 4.7 times better than approach 2, in the throughput test results. This is probably because these are results from a single thread execution only. The individual query results for every test can be viewed, by running the index.html file generated by JMeter, inside the results folder in my public Github repository [37].

Now I move on to present the throughput results calculated with the TPC-H Benchmark metrics by using the formulas presented in section 5.7. For the Power test formula, the sum of the execution times of every query is considered, but in the Throughput test results, these times from each approach, correspond to the sum of the 4 test sessions performed.

In the Throughput formula, the response times of the refresh functions are excluded because these were only executed by one thread in the four throughput test sessions. This makes the refresh functions not suitable to use in the calcula-

tions, and because TPC-H benchmark does not make refresh functions mandatory, I decided to not include them in these calculations. The result from the composite query metric is given in queries per hour (Q/H).

- Approach 1

$$\text{Power@30} = \frac{3600 \cdot 30}{\sqrt[22]{29.329 \cdot 10}} = 83420.7 \quad \text{Q/H} \tag{7.1}$$

$$\text{Throughput@30} = \frac{4 \cdot 20 \cdot 3600}{2138.564} \cdot 30 = 4040 \quad \text{Q/H} \tag{7.2}$$

$$\text{QphH@30} = \sqrt{83420.7 \cdot 4040} = 18358.1 \quad \text{Q/H} \tag{7.3}$$

- Approach 2

$$\text{Power Test} = \frac{3600 \cdot 30}{\sqrt[22]{1326.758 \cdot 13}} = 69317.8 \quad \text{Q/H} \tag{7.4}$$

$$\text{Throughput@30} = \frac{4 \cdot 20 \cdot 3600}{526965.277} \cdot 30 = 16.3 \quad \text{Q/H} \tag{7.5}$$

$$\text{QphH@30} = \sqrt{69317.8 \cdot 16.3} = 1063 \quad \text{Q/H} \tag{7.6}$$

- Approach 3

$$\text{Power Test} = \frac{3600 \cdot 30}{\sqrt[22]{1876.334 \cdot 12}} = 68483 \quad \text{Q/H} \tag{7.7}$$

$$\text{Throughput@30} = \frac{4 \cdot 20 \cdot 3600}{39186.746} \cdot 30 = 220.4 \quad \text{Q/H} \tag{7.8}$$

$$\text{QphH@30} = \sqrt{68483 \cdot 220.4} = 3885.1 \quad \text{Q/H} \tag{7.9}$$

Analyzing the composite query metric results, the throughput of approach 1 is 17.27 times better than approach 2, it is 4.73 times better than approach 3, and approach 3 is 3.65 times better than approach 2. This again reinforces the idea that the performance difference is directly related to the table sizes and the amount of data queried. In approach 1 these tables are much smaller than in the other two as contain millions of records instead of hundreds of millions.

In the case of approach 3, the tables have the same size as in approach 2, but the amount of data queried is smaller, because by grouping data there are fewer rows to search through when querying group-specific data, or tenant-specific data. It's worth emphasizing that while the composite query metric provides an insightful snapshot of system performance, the overall assessment should take into account other dimensions such as the complexity and scalability of these solutions, which I analyze next.

7.3 Scalability and complexity analysis

In terms of scalability these approaches scale well with using more RAM in the database machines. I concluded this by executing an incremental testing process where I started using the cheapest hardware available in the RDS instances, which was a 2-core CPU with 1 GB of RAM, it took over 12 hours without being able to reach the end of the tests, and from there I went through the process of scaling the RAM and number of cores used but always with the same result. The tests would slow down after about 6 hours where memory swap operations would take the majority of the execution. Only when I used an 8-core CPU with 32 GB of RAM was I able to get results for the tests for approaches 2 and 3.

In terms of complexity of implementation, it was expected that approach 1 would be the most complex to handle, however, I encountered that all three approaches were of similar complexity to implement and I will explain by referring to the several steps I took. Starting with the database generation process, for approach 1 I had to create several schemas for all the tenants, for approach 3 I had to generate one schema and the two extra tables to be able to group tenants, and for approach 2 I had only to generate one schema and an extra table, but overall all of these processes were handled with auxiliary scripts that were not to complicated to create.

The data generation process was simple because of the use of the provided *dbgen* file. I only generated the data set once and reused it in all three approaches. Moreover, the query generation process was similar for all approaches, only requiring similar changes to all query sets, that were handled with simple auxiliary scripts. Lastly, the data loading process was more complex to handle in approaches 2 and 3 than it was in approach 1. Because of the need to reuse the generated data for the first approach, adapting such data without generating primary and foreign key conflicts took more time and thought.

These three phases are crucial steps to take into consideration during the development process of the porting tool, be it in the case where it is required that such tool is able to covert some or all existing data from the database of the system being ported. But also in the case where just new data in the database is required to be multi-tenant, the different levels of isolation implemented and tested, give a starting point and guidelines for the porting tool development process.

Chapter 8

Planning

This Chapter is used to present the planned versus real internship schedules, and the risks associated with the project. For better visualization of the schedule Gantt charts are used.

8.1 First semester

The schedule for this semester was divided into four separate timelines, the first was dedicated to defining the application to port, which was defined by the Altice Labs (ALB) collaborators, the second to do research on the state of the art and technologies available to do the porting, the third was aimed at defining the requirements and architecture, and the last timeline would be dedicated to write the intermediate report. Figure 2.1 shows the Gantt chart for this semester.

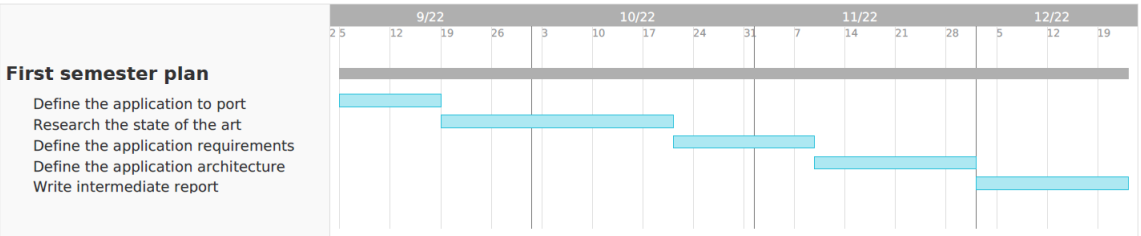

Figure 8.1: Schedule of the first Semester

8.2 Second semester Planned vs Real schedule

The second semester contains the implementation phase, which is divided in five stages. The first is for setting up and installing the application, the second is aimed at porting the application and testing, the third is define the analysis in detail, the fourth stage is dedicated to testing and analyzing the results, and the final stage is dedicated to writing the dissertation. A Gantt chart with the schedule is presented in Figure 2.2.

The figure 8.2 shows the original plan and figure 8.3 shows the actual schedule executed in the second semester.

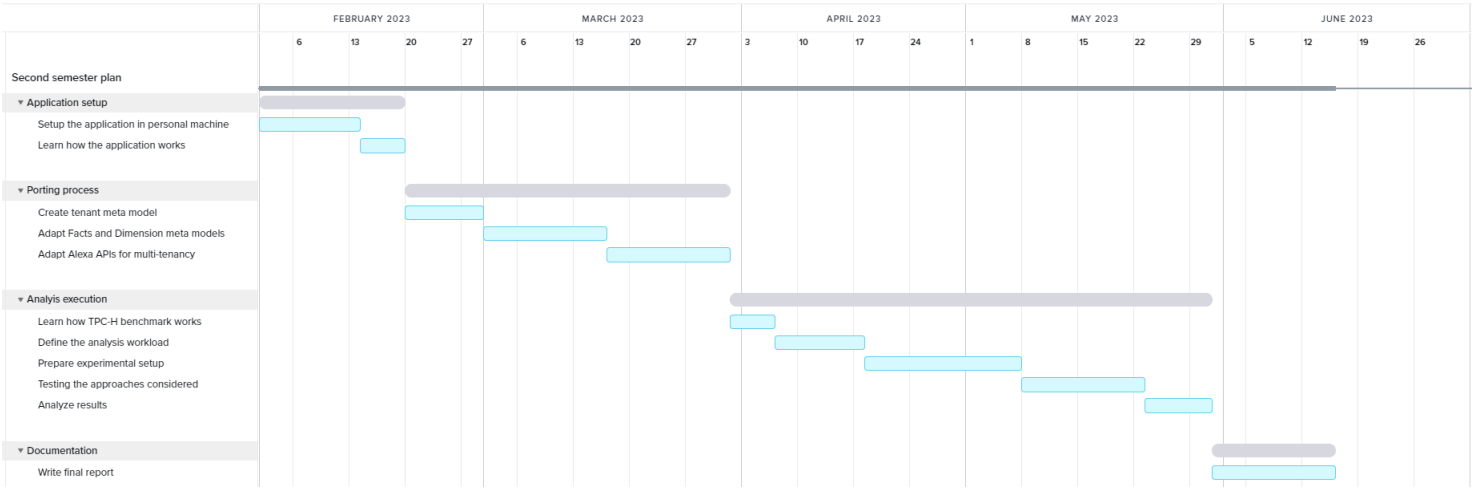

Figure 8.2: Planned schedule of the second Semester

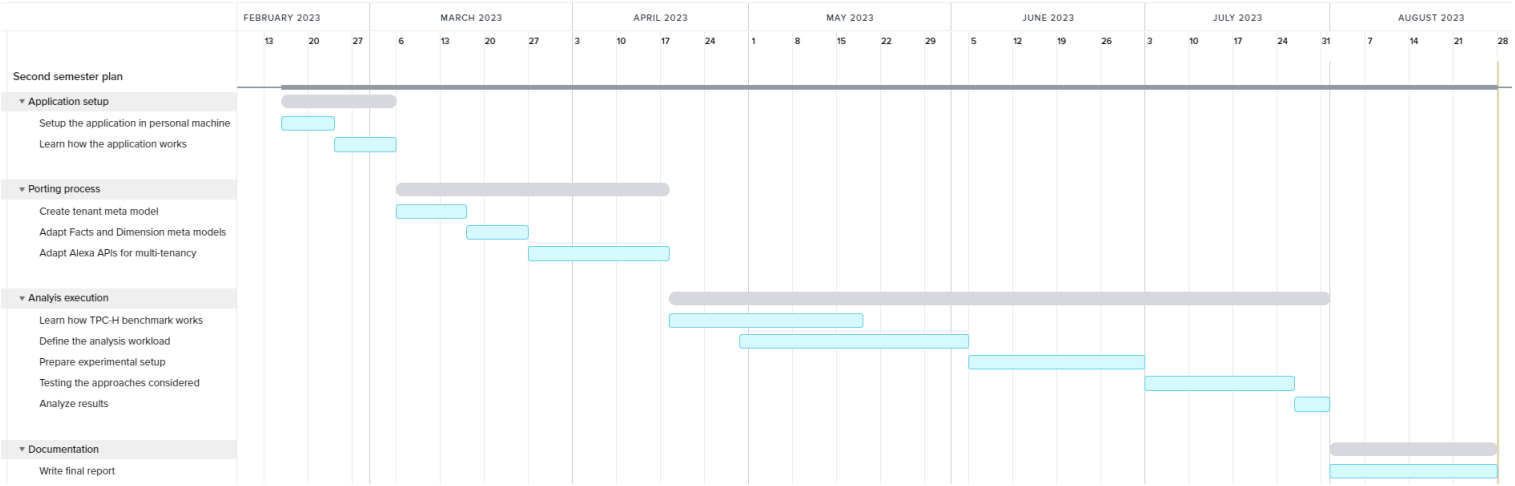

Figure 8.3: Real schedule of the second Semester

The schedule for the second semester ended up being longer than expected due to several factors. First, in the process of porting Alexa, there were delays in access to the system, which meant I was only able to set it up and understand fully how it works, later on. Also during this phase, I had to redo the same task several times, because there were discrepancies between my implementations and the requirements that the ALB collaborators wanted. This made the Alexa porting process more time-consuming the initially expected.

Furthermore, during the analysis definition, almost all of the tasks took longer than predicted. Because I had no experience working with such a benchmark or doing an experimental setup, the expected timelines were too ambitious. For this reason, it was necessary to extend the time budget for the project by two months.

8.3 Risk Assessment

Here I outline the possible risks that did not impact and the ones that did impact the development and analysis process as well as, the decisions taken during the whole process, the list includes:

- Delays in the access to the Alexa microservice code. This did happen which caused a delay in the start of the development phase.
- Problems with integration of the technologies and tools, selected by ALB collaborators for the porting process of Alexa. This did not happen, the setup process was straightforward without problems.
- Changes in requirements of Alexa during the implementation phase, causing delays. This did not happen.
- Requirements of the Alexa microservice are not well defined, causing delays during development. This did happen which caused extra iterations in the development process to occur.
- Not having access to a real business data set to validate the analysis. This was an issue that led to the decision to use the TPC-H Benchmark.
- Costs restrictions impacting the hardware selection to run the tests. The cost of using Amazon Web services impacted the selection of the hardware used in the tests performed. The decision to use cheaper options led to a downsize in the scale of the test threads and database size used, due to the extremely long execution times recorded in the initial failed test tries.
- The time available to run the tests impacts the hardware selection. Despite what is reported in the previous risk, the time of the initial test executions forced an incremental process of selecting hardware with increasing memory capacity, until the times were within the time frame with not more than 4 hours.

Chapter 9

Conclusion

In this concluding chapter, I synthesize the key findings and insights drawn from the analysis, comment about future work, and provide some thoughts on this experience.

9.1 Analysis Reflection

In this section I present my final reflection regarding the results and insights gathered. Starting with the porting process of the Alexa microservice, this stage of the project allowed me to test different solutions to multi-tenancy studied in the state-of-the-art phase, in a complete system with all three layers, business, logic, and persistence.

After the requirements for the Alexa microservice were implemented, I gained a better understanding of the details involved in porting a data warehouse microservice based on a star schema. I gained knowledge on how to convert the database to a multi-tenant one, and also how to adapt the business and logic layers to work with the new database. However, these findings were specific to the system at hand, and I wanted to expand this into a more broad implementation to apply to something like the porting tool.

From the previous stage, I moved on to the analysis, where in terms of throughput performance approach 1 is the best-performing solution as expected, and approaches 2 and 3 have similar performance that will vary according to the amount of data grouped among tenants, the number of tenants each group has, and the amount of shared data from each tenant. These results provide insights on the performance impact of each solution, that are relevant when developing the porting tool because they better help define to what type of systems the tool should be developed for.

With these results it is easy to conclude that approach 1 is the best way for systems where performance is the most critical aspect, however, it must be noted that throughput performance is only one aspect of a system, there might be other more important aspects to consider when developing a system, and that these

tests were executed without any performance enhancement technics used. Like the use of indexes that might speed up the throughput and reduce the performance deficit between these three solutions.

9.2 Future work

As stated before throughout this document, this work is intended to be used as a guideline for the development process of a porting tool. Giving insights on three possible solutions for multi-tenancy on data warehouse databases, which help in identifying the requirements of such tool when deciding what type of systems is the tool aimed at. The results gathered here point to the fact that joining multiple tenants can have significant impact on performance but allow more flexibility in menaging the data, which helps when developing a porting tool aimed at many different types of systems.

9.3 Final Thoughts

In conclusion this thesis gave me practical experience on how to work with data warehouse systems based on star schema and snowflake schema. Allowed me to learn how to do a detailed and comprehensive experimental setup, and bettering my skills of working with cloud-based solutions like Amazon Web Services, and also learning what is the TPC-H Benchmark, a relevant benchmarking standard, and how I can use it perform data analysis.

References

[1] Anand Akela. 4 cluster management tools to compare. https://www.appdynamics.com/blog/product/4-cluster-management-tools-to-compare/, 2016. Last visited on 27/10/2022.

[2] Amazon. Ec2 instances hardware. https://aws.amazon.com/pt/ec2/instance-types/?trk=e011bdeb-247d-4bc7-92eb1b11f87bfc8a&sc_channel=ps&ef_id=CjwKCAjwivemBhBhEiwAJxNWN-AGgkOfl21-cSmTrRY3dDg7ICM2AQeomCyECNDZeccYH2Po0yLNBwE:G:s&s_kwcid=AL!4422!3!536392690705!p!!g!!amazon%20web%20services%20ec2%20instance!12195830327!119606866280. Last visited on 17/8/2023.

[3] Amazon. Rds instances hardware. https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.DBInstanceClass.html#Concepts.DBInstanceClass.Types. Last visited on 17/8/2023.

[4] The Kubernetes Authors. Kubernetes components. https://kubernetes.io/docs/concepts/overview/components/, 2022. Last visited on 26/10/2022.

[5] The Kubernetes Authors. Kubernetes definition. https://kubernetes.io/, 2022. Last visited on 07-10-2022.

[6] Cor-Paul Bezemer and Andy Zaidman. Multi-tenant saas applications: Maintenance dream or nightmare? https://azaidman.github.io/publications/bezemerIWPSE2010.pdf, 2010. Last visited on 11/10/2022.

[7] Kevin Casey. What's the difference between a pod, a cluster, and a container? https://enterprisersproject.com/article/2020/9/pod-cluster-container-what-is-difference, 2020. Last visited on 28/10/2022.

[8] Wesley Chai, Kate Brush, and Stephen J. Bigelow. What is paas? platform as a service definition and guide. https://www.techtarget.com/searchcloudcomputing/definition/Platform-as-a-Service-PaaS, 2022. Last visited on 08-10-2022.

[9] Mark Drake. Understanding database sharding. https://www.digitalocean.com/community/tutorials/understanding-database-sharding, 2022. Last visited on 26-10-2022.

[10] IBM Cloud Education. What is a database schema? https://www.ibm.com/cloud/learn/database-schema, 2021. Last visited on 06/11/2022.

[11] EMnify. What is iaas? infrastructure as a service explained. https://www.emnify.com/iot-glossary/iaas?utm_term=&utm_campaign=SEA-EN-EUR_EN-MC-NB-NoFu-Performance_Max%7CPilot&utm_source=google&utm_medium=cpc&hsa_acc=2935385868&hsa_cam=17491136806&hsa_grp=&hsa_ad=&hsa_src=x&hsa_tgt=&hsa_kw=&hsa_mt=&hsa_net=adwords&hsa_ver=3&gclid=CjOKCQjw-fmZBhDtARIsAH6H8qhCTBjytzn80pCSfOYgDdcOk1X9FYvDG2blyuMV1E6Nlu4DRSJg_oQaAhIrEALw_wcB, 2021. Last visited on 08-10-2022.

[12] Fedora. Fedora coreos documentation. https://docs.fedoraproject.org/en-US/fedora-coreos/, 2022. Last visited on 28/10/2022.

[13] The Apache Software Foundation. What is mesos? a distributed systems kernel. https://mesos.apache.org/, 2022. Last visited on 29/10/2022.

[14] The PostgreSQL Global Development Group. What is postgresql? https://www.postgresql.org/docs/current/intro-whatis.html. Last visited on 06/11/2022.

[15] Chang Jie Guo1, Wei Sun1, Ying Huang2, Zhi Hu Wang1, and Bo Gao1. A framework for native multi-tenancy application development and management. https://ieeexplore.ieee.org/abstract/document/4285271/authors#authors, 2007. Last visited on 12/10/2022.

[16] Docker Inc. Docker overview. https://docs.docker.com/get-started/overview/. Last visited on 07-10-2022.

[17] Docker Inc. How nodes work. https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/. Last visited on 27-10-2022.

[18] Red Hat Inc. What is etcd? https://www.redhat.com/en/topics/containers/what-is-etcd, 2019. Last visited on 30/10/2022.

[19] Red Hat Inc. What is multitenancy? https://www.redhat.com/en/topics/cloud-computing/what-is-multitenancy, 2020. Last visited on 28/09/2022.

[20] VMware Inc. What is a virtual machine? https://www.vmware.com/topics/glossary/content/virtual-machine.html. Last visited on 06-10-2022.

[21] Wix.com Inc. Software as a service (saas). https://www.wix.com/encyclopedia/definition/software-as-a-service-saas?utm_source=google&utm_medium=cpc&utm_campaign=13708482660^124757113432&experiment_id=S0755701269ffDSA&gclid=CjOKCQjw-fmZBhDtARIsAH6H8qgDBfP5QKqZKsmbXQ5Tfuadqpw4rzxVSFb4tULCooj2qdPezgE0iHkaAwcB. Last visited on 05-10-2022.

[22] JavaTpoint. Postgresql features. https://www.javatpoint.com/postgresql-features. Last visited on 06/11/2022.

[23] Impelsys Private Limited. Importance of multi tenancy – true architecture for software-as-a-service (saas). https://www.impelsys.com/blog/importance-of-multi-tenancy-true-architecture-for-software-as-a-service-saas 2022. Last visited on 27/09/2022.

[24] Microsoft. Noisy neighbor antipattern. https://learn.microsoft.com/en-us/azure/architecture/antipatterns/noisy-neighbor/noisy-neighbor. Last visited on 25-10-2022.

[25] Microsoft. Multi-tenant saas database tenancy patterns. https://learn.microsoft.com/en-us/azure/azure-sql/database/saas-tenancy-app-design-patterns?view=azuresql, 2022. Last visited on 25-09-2022.

[26] Lukonde Mwila. Kubernetes multi-tenancy | best practices in 2022. https://www.containiq.com/post/kubernetes-multi-tenancy, 2022. Last visited on 27/10/2022.

[27] Isaac Odun-Ayo, Sanjay Misra, Olusola Abayomi-Alli, and Olasupo Ajayi. Cloud multi-tenancy: Issues and developments. https://dl.acm.org/doi/abs/10.1145/3147234.3148095?casa_token=3Qk1Wv4rbLoAAAAA:aJsUiwm0Y7xZIsmnIN0SH44CfkgRfepXuEZo9uF4pCUrU3Wa_hLH8NqfAm5sGLo6r-T9j5lg5Gytldg, 2017. Last visited on 11/10/2022.

[28] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm (extended version). https://raft.github.io/raft.pdf, 2022. Last visited on 27-10-2022.

[29] Oracle. What is a data warehouse? https://www.oracle.com/pt/database/what-is-a-data-warehouse/. Last visited on 05/12/2022.

[30] Oracle. What is a database? https://www.oracle.com/database/what-is-database/. Last visited on 05-10-2022.

[31] Tal Perry. Database multi-tenancy for saas. https://www.lighttag.io/blog/database-multi-tenancy/. Last visited on 25-09-2022.

[32] Chris Richardson. What are microservices? https://microservices.io/. Last visited on 06/10/2022.

[33] Joel Shore. 7 challenges in multi-tenancy testing and their solutions. https://www.netsolutions.com/insights/multi-tenancy-testing-top-challenges-and-solutions/, 2020. Last visited on 10/10/2022.

[34] Joel Shore. cloud application. https://www.techtarget.com/searchcloudcomputing/definition/cloud-application, 2021. Last visited on 05-10-2022.

[35] Simplilearn. What is multitenancy: Definition, importance, and applications. https://www.simplilearn.com/what-is-multitenancy-article#benefits_of_multitenancy_architecture, 2022. Last visited on 26/09/2022.

[36] SmartBear Software. Openapi specification. https://swagger.io/specification/. Last visited on 03/12/2022.

[37] Eurico Sousa. My github repository. https://github.com/Eurico-98/Multi-tenancy-support-scripts.git. Last visited on 2/8/2023.

[38] TPC. Tpc-h description. https://www.tpc.org/tpch/default5.asp. Last visited on 1/6/2023.

[39] TPC. Tpc-h documentation. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf. Last visited on 1/6/2023.

[40] Avneesh Vashistha and Pervez Ahmed. Saas multi-tenancy isolation testingchallenges and issues. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.477.1823&rep=rep1&type=pdf, 2012. Last visited on 10/10/2022.

[41] PostgreSQL Tutorial Website. What is a database transaction. https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-transaction/. Last visited on 06/11/2022.

Appendices

Appendix A

SQL code used to setup the database

```
CREATE TABLE nation(
    n_nationkey INTEGER  NOT NULL,
    n_name      CHAR(25) NOT NULL,
    n_regionkey INTEGER  NOT NULL,
    n_comment   VARCHAR(152));

CREATE TABLE region(
    r_regionkey INTEGER  NOT NULL,
    r_name      CHAR(25) NOT NULL,
    r_comment   VARCHAR(152));

CREATE TABLE part(
    p_partkey     INTEGER        NOT NULL,
    p_name        VARCHAR(55)    NOT NULL,
    p_mfgr        CHAR(25)       NOT NULL,
    p_brand       CHAR(10)       NOT NULL,
    p_type        VARCHAR(25)    NOT NULL,
    p_size        INTEGER        NOT NULL,
    p_container   CHAR(10)       NOT NULL,
    p_retailprice DECIMAL(15, 2) NOT NULL,
    p_comment     VARCHAR(23)    NOT NULL);

CREATE TABLE supplier(
    s_suppkey   INTEGER        NOT NULL,
    s_name      CHAR(25)       NOT NULL,
    s_address   VARCHAR(40)    NOT NULL,
    s_nationkey INTEGER        NOT NULL,
    s_phone     CHAR(15)       NOT NULL,
    s_acctbal   DECIMAL(15, 2) NOT NULL,
    s_comment   VARCHAR(101)   NOT NULL);

CREATE TABLE partsupp(
    ps_partkey    INTEGER        NOT NULL,
    ps_suppkey    INTEGER        NOT NULL,
```

```
    ps_availqty   INTEGER        NOT NULL,
    ps_supplycost DECIMAL(15, 2) NOT NULL,
    ps_comment    VARCHAR(199)   NOT NULL);

CREATE TABLE customer(
    c_custkey    INTEGER        NOT NULL,
    c_name       VARCHAR(25)    NOT NULL,
    c_address    VARCHAR(40)    NOT NULL,
    c_nationkey  INTEGER        NOT NULL,
    c_phone      CHAR(15)       NOT NULL,
    c_acctbal    DECIMAL(15, 2) NOT NULL,
    c_mktsegment CHAR(10)       NOT NULL,
    c_comment    VARCHAR(117)   NOT NULL);

CREATE TABLE orders(
    o_orderkey      INTEGER        NOT NULL,
    o_custkey       INTEGER        NOT NULL,
    o_orderstatus   CHAR(1)        NOT NULL,
    o_totalprice    DECIMAL(15, 2) NOT NULL,
    o_orderdate     DATE           NOT NULL,
    o_orderpriority CHAR(15)       NOT NULL,
    o_clerk         CHAR(15)       NOT NULL,
    o_shippriority  INTEGER        NOT NULL,
    o_comment       VARCHAR(79)    NOT NULL);

CREATE TABLE lineitem(
    l_orderkey      INTEGER        NOT NULL,
    l_partkey       INTEGER        NOT NULL,
    l_suppkey       INTEGER        NOT NULL,
    l_linenumber    INTEGER        NOT NULL,
    l_quantity      DECIMAL(15, 2) NOT NULL,
    l_extendedprice DECIMAL(15, 2) NOT NULL,
    l_discount      DECIMAL(15, 2) NOT NULL,
    l_tax           DECIMAL(15, 2) NOT NULL,
    l_returnflag    CHAR(1)        NOT NULL,
    l_linestatus    CHAR(1)        NOT NULL,
    l_shipdate      DATE           NOT NULL,
    l_commitdate    DATE           NOT NULL,
    l_receiptdate   DATE           NOT NULL,
    l_shipinstruct  CHAR(25)       NOT NULL,
    l_shipmode      CHAR(10)       NOT NULL,
    l_comment       VARCHAR(44)    NOT NULL);
```

ALTER TABLE region ADD PRIMARY KEY (r_nregionkey);
ALTER TABLE nation ADD PRIMARY KEY (n_nnationkey);
ALTER TABLE customer ADD PRIMARY KEY (c_ncustkey);
ALTER TABLE supplier ADD PRIMARY KEY (s_nsuppkey);
ALTER TABLE part ADD PRIMARY KEY (p_npartkey);

```
ALTER TABLE partsupp ADD PRIMARY KEY (ps_npartkey, ps_nsuppkey);
ALTER TABLE orders ADD PRIMARY KEY (o_norderkey);
ALTER TABLE lineitem ADD PRIMARY KEY (l_norderkey, l_nlinenumber);
ALTER TABLE nation ADD FOREIGN KEY (n_nregionkey) REFERENCES region(r_nregionkey);
ALTER TABLE supplier ADD FOREIGN KEY (s_nnationkey) REFERENCES nation(n_nnationkey);
ALTER TABLE customer ADD FOREIGN KEY (c_nnationkey) REFERENCES nation(n_nnationkey);
ALTER TABLE partsupp ADD FOREIGN KEY (ps_nsuppkey) REFERENCES supplier(s_nsuppkey);
ALTER TABLE partsupp ADD FOREIGN KEY (ps_npartkey) REFERENCES part(p_npartkey);
ALTER TABLE orders ADD FOREIGN KEY (o_ncustkey) REFERENCES customer(c_ncustkey);
ALTER TABLE lineitem ADD FOREIGN KEY (l_norderkey) REFERENCES orders(o_norderkey);
ALTER TABLE lineitem ADD FOREIGN KEY (l_npartkey,l_nsuppkey) REFERENCES partsupp(ps_npartkey,ps_nsuppkey);
```

Appendix B

Analysis approach 1 Physical Diagram

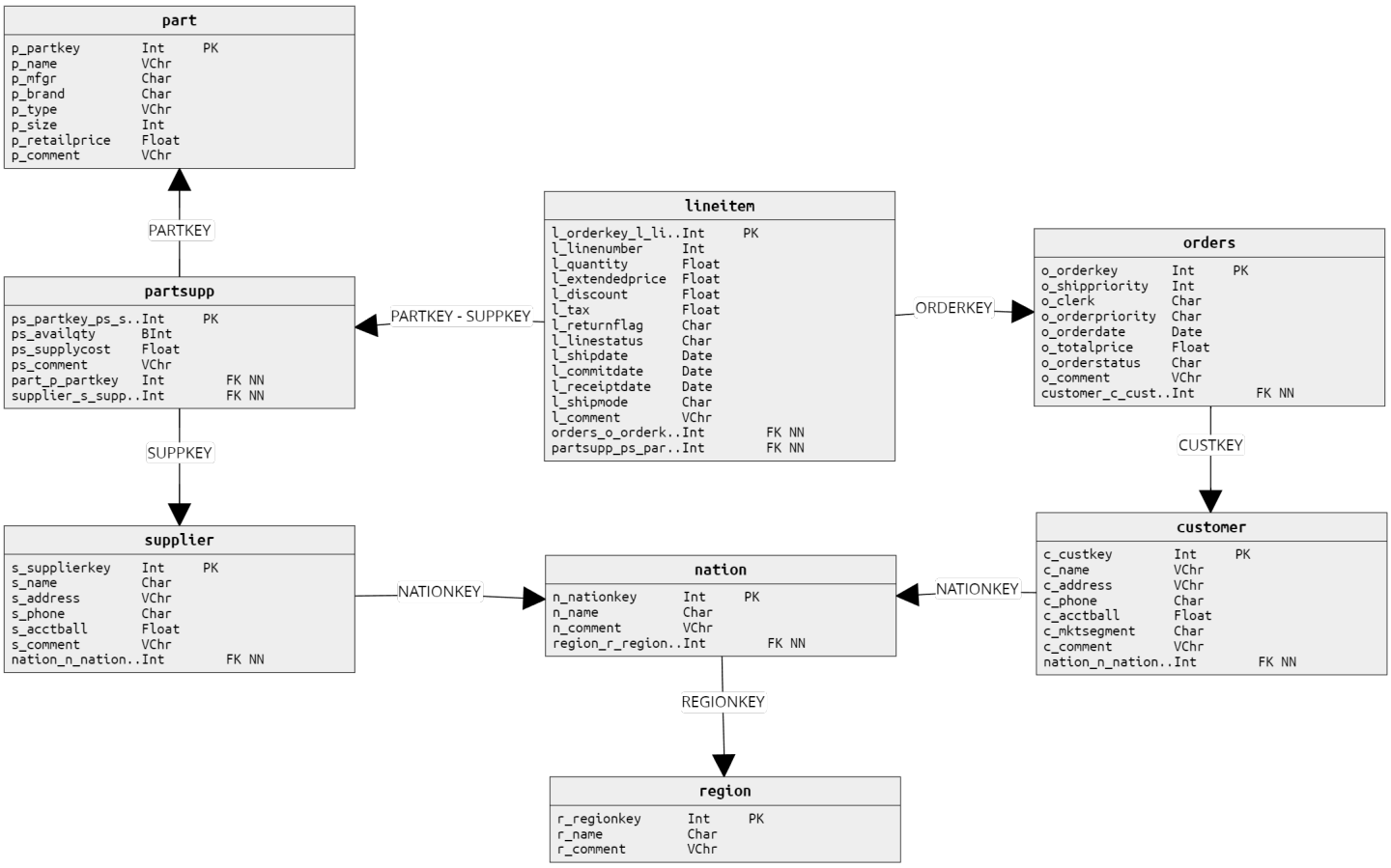

Figure B.1: Approach 1 Physical Diagram

Appendix C

TPC-H Benchmark Query set

The parameters DELTA, REGION, SIZE, TYPE, SEGMENT, DATE, DISCOUNT, QUANTITY, NATION, COLOR, FRACTION, SHIPMODE, WORD, STREAM_ID, BRAND, CONTAINER, represent the randomly selected values by the qgen file when generating query variants. The SCHEMA parameter is only used in approach one and it represents the tenant schema selected during the throughput tests.

Q1 - Pricing Summary Report Query reports the amount of business that was billed, shipped, and returned.

```
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
    sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
    SCHEMA.lineitem
where
    l_shipdate <= date '1998-12-01' - interval 'DELTA' day (3)
group by
    l_returnflag,
    l_linestatus
order by
    l_returnflag,
    l_linestatus;
```

Q2 - Minimum Cost Supplier Query finds which supplier should be selected to place an order for a given part in a given region.

```
select
    s_acctbal,
    s_name,
    n_name,
    p_partkey,
    p_mfgr,
    s_address,
    s_phone,
    s_comment
from
    SCHEMA.part,
    SCHEMA.supplier,
    SCHEMA.partsupp,
    SCHEMA.nation,
    SCHEMA.region
where
    p_partkey = ps_partkey
    and s_suppkey = ps_suppkey
    and p_size = SIZE
    and p_type like '%TYPE'
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'REGION'
    and ps_supplycost = (
        select
            min(ps_supplycost)
        from
            SCHEMA.partsupp,
            SCHEMA.supplier,
            SCHEMA.nation,
            SCHEMA.region
        where
            p_partkey = ps_partkey
            and s_suppkey = ps_suppkey
            and s_nationkey = n_nationkey
            and n_regionkey = r_regionkey
            and r_name = 'REGION'
    )
order by
    s_acctbal desc,
    n_name,
    s_name,
    p_partkey;
```

Q3 - Shipping Priority Query retrieves the 10 unshipped orders with the highest value.

```
select
    l_orderkey,
    sum(l_extendedprice*(1-l_discount)) as revenue,
    o_orderdate,
    o_shippriority
from
    SCHEMA.customer,
    SCHEMA.orders,
    SCHEMA.lineitem
where
    c_mktsegment = 'SEGMENT'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate < date 'DATE'
    and l_shipdate > date 'DATE'
group by
    l_orderkey,
    o_orderdate,
    o_shippriority
order by
    revenue desc,
    o_orderdate;
```

Q4 - Order Priority Checking Query determines how well the order priority system is working and gives an assessment of customer satisfac-tion.

```
select
    o_orderpriority,
    count(*) as order_count
from
    SCHEMA.orders
where
    o_orderdate >= date 'DATE'
    and o_orderdate < date 'DATE' + interval '3' month
    and exists (
        select
            *
        from
            SCHEMA.lineitem
        where
            l_orderkey = o_orderkey
            and l_commitdate < l_receiptdate
    )
group by
    o_orderpriority
order by
    o_orderpriority;
```

Q5 - Local Supplier Volume Query lists the revenue volume done through local suppliers.

```
select
    n_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue
from
    SCHEMA.customer,
    SCHEMA.orders,
    SCHEMA.lineitem,
    SCHEMA.supplier,
    SCHEMA.nation,
    SCHEMA.region
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and l_suppkey = s_suppkey
    and c_nationkey = s_nationkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'REGION'
    and o_orderdate >= date 'DATE'
    and o_orderdate < date 'DATE' + interval '1' year
group by
    n_name
order by
    revenue desc;
```

Q6 - Forecasting Revenue Change Query quantifies the amount of revenue increase that would have resulted from eliminating certain company-wide discounts in a given percentage range in a given year. Asking this type of "what if" query can be used to look for ways to increase revenues.

```
select
    sum(l_extendedprice * l_discount) as revenue
from
    SCHEMA.lineitem
where
    l_shipdate >= date 'DATE'
    and l_shipdate < date 'DATE' + interval '1' year
    and l_discount between DISCOUNT - 0.01 and DISCOUNT + 0.01
    and l_quantity < QUANTITY;
```

Q7 - Volume Shipping Query determines the value of goods shipped between certain nations to help in the re-negotiation of shipping contracts.

```
select
    supp_nation,
    cust_nation,
    l_year,
    sum(volume) as revenue
from (
    select
        n1.n_name as supp_nation,
        n2.n_name as cust_nation,
        extract(year from l_shipdate) as l_year,
        l_extendedprice * (1 - l_discount) as volume
    from
        SCHEMA.supplier,
        SCHEMA.lineitem,
        SCHEMA.orders,
        SCHEMA.customer,
        SCHEMA.nation n1,
        SCHEMA.nation n2
    where
        s_suppkey = l_suppkey
        and o_orderkey = l_orderkey
        and c_custkey = o_custkey
        and s_nationkey = n1.n_nationkey
        and c_nationkey = n2.n_nationkey
        and (
            (n1.n_name = 'NATION1' and n2.n_name = 'NATION2')
            or (n1.n_name = 'NATION2' and n2.n_name = 'NATION1')
        )
        and l_shipdate between date '1995-01-01' and date '1996-12-31'
    ) as shipping
group by
    supp_nation,
    cust_nation,
    l_year
order by
    supp_nation,
    cust_nation,
    l_year;
```

Q8 - National Market Share Query determines how the market share of a given nation within a given region has changed over two years for a given part type.

```
select
    o_year,
    sum(case
        when nation = 'NATION'
        then volume
        else 0
    end) / sum(volume) as mkt_share
from (
    select
        extract(year from o_orderdate) as o_year,
        l_extendedprice * (1 - l_discount) as volume,
        n2.n_name as nation
    from
        SCHEMA.part,
        SCHEMA.supplier,
        SCHEMA.lineitem,
        SCHEMA.orders,
        SCHEMA.customer,
        SCHEMA.nation n1,
        SCHEMA.nation n2,
        SCHEMA.region
    where
        p_partkey = l_partkey
        and s_suppkey = l_suppkey
        and l_orderkey = o_orderkey
        and o_custkey = c_custkey
        and c_nationkey = n1.n_nationkey
        and n1.n_regionkey = r_regionkey
        and r_name = 'REGION'
        and s_nationkey = n2.n_nationkey
        and o_orderdate between date '1995-01-01' and date '1996-12-31'
        and p_type = 'TYPE'
    ) as all_nations
group by
    o_year
order by
    o_year;
```

Q9 - Product Type Profit Measure Query determines how much profit is made on a given line of parts, broken out by supplier nation and year.

```
select
    nation,
    o_year,
    sum(amount) as sum_profit
from (
    select
        n_name as nation,
        extract(year from o_orderdate) as o_year,
        l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity
        as amount
    from
        SCHEMA.part,
        SCHEMA.supplier,
        SCHEMA.lineitem,
        SCHEMA.partsupp,
        SCHEMA.orders,
        SCHEMA.nation
    where
        s_suppkey = l_suppkey
        and ps_suppkey = l_suppkey
        and ps_partkey = l_partkey
        and p_partkey = l_partkey
        and o_orderkey = l_orderkey
        and s_nationkey = n_nationkey
        and p_name like '%COLOR%'
    ) as profit
group by
    nation,
    o_year
order by
    nation,
    o_year desc;
```

Q10 - Returned Item Reporting Query identifies customers who might be having problems with the parts that are shipped to them.

```
select
    c_custkey,
    c_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    c_acctbal,
    n_name,
    c_address,
    c_phone,
    c_comment
from
    SCHEMA.customer,
    SCHEMA.orders,
    SCHEMA.lineitem,
    SCHEMA.nation
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate >= date 'DATE'
    and o_orderdate < date 'DATE' + interval '3' month
    and l_returnflag = 'R'
    and c_nationkey = n_nationkey
group by
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
    n_name,
    c_address,
    c_comment
order by
    revenue desc;
```

Q11 - Important Stock Identification Query finds the most important subset of suppliers' stock in a given nation.

```
select
    ps_partkey,
    sum(ps_supplycost * ps_availqty) as value
from
    SCHEMA.partsupp,
    SCHEMA.supplier,
    SCHEMA.nation
where
    ps_suppkey = s_suppkey
    and s_nationkey = n_nationkey
    and n_name = 'NATION'
group by
    ps_partkey having
        sum(ps_supplycost * ps_availqty) > (
            select
                sum(ps_supplycost * ps_availqty) * FRACTION
            from
                SCHEMA.partsupp,
                SCHEMA.supplier,
                SCHEMA.nation
            where
                ps_suppkey = s_suppkey
                and s_nationkey = n_nationkey
                and n_name = 'NATION'
    )
order by
    value desc;
```

Q12 - Shipping Modes and Order Priority Query determines whether selecting less expensive modes of shipping is negatively affecting the critical-prior-ity orders by causing more parts to be received by customers after the committed date.

```
select
    l_shipmode,
    sum(case
        when o_orderpriority = '1-URGENT'
            or o_orderpriority = '2-HIGH'
            then 1
        else 0
    end) as high_line_count,
    sum(case
        when o_orderpriority <> '1-URGENT'
            and o_orderpriority <> '2-HIGH'
            then 1
        else 0
    end) as low_line_count
from
    SCHEMA.orders,
    SCHEMA.lineitem
where
    o_orderkey = l_orderkey
    and l_shipmode in ('SHIPMODE1', 'SHIPMODE2')
    and l_commitdate < l_receiptdate
    and l_shipdate < l_commitdate
    and l_receiptdate >= date 'DATE'
    and l_receiptdate < date 'DATE' + interval '1' year
group by
    l_shipmode
order by
    l_shipmode;
```

Q13 - Customer Distribution Query seeks relationships between customers and the size of their orders.

```
select
    c_count,
    count(*) as custdist
from(
    select
        c_custkey,
        count(o_orderkey) as c_count
    from
        SCHEMA.customer left outer join SCHEMA.orders on
            c_custkey = o_custkey
            and o_comment not like '%WORD1%WORD2%'
    group by
        c_custkey
    ) as c_orders (c_custkey, c_count)
group by
    c_count
order by
    custdist desc,
    c_count desc;
```

Q14 - Promotion Effect Query monitors the market response to a promotion such as TV advertisements or a special campaign.

```
select
    100.00 * sum(case
        when p_type like 'PROMO%'
            then l_extendedprice * (1 - l_discount)
        else 0
    end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue
from
    SCHEMA.lineitem,
    SCHEMA.part
where
    l_partkey = p_partkey
    and l_shipdate >= date 'DATE'
    and l_shipdate < date 'DATE' + interval '1' month;
```

Q15 - Top Supplier Query determines the top supplier so it can be rewarded, given more business, or identified for special recogni-tion.

```
create view SCHEMA.revenueSTREAM_ID (supplier_no, total_revenue) as
    select
        l_suppkey,
        sum(l_extendedprice * (1 - l_discount))
    from
        SCHEMA.lineitem
    where
        l_shipdate >= date 'DATE'
        and l_shipdate < date 'DATE' + interval '3' month
    group by
        l_suppkey;


select
    s_suppkey,
    s_name,
    s_address,
    s_phone,
    total_revenue
from
    SCHEMA.supplier,
    SCHEMA.revenueSTREAM_ID
where
    s_suppkey = supplier_no
    and total_revenue = (
        select
            max(total_revenue)
        from
            SCHEMA.revenueSTREAM_ID
    )
order by
    s_suppkey;

drop view revenueSTREAM_ID;
```

Q16 - Parts/Supplier Relationship Query finds out how many suppliers can supply parts with given attributes. It might be used, for example, to determine whether there is sufficient number of suppliers for heavily ordered parts.

```
select
    p_brand,
    p_type,
    p_size,
    count(distinct ps_suppkey) as supplier_cnt
from
    SCHEMA.partsupp,
    SCHEMA.part
where
    p_partkey = ps_partkey
    and p_brand <> 'BRAND'
    and p_type not like 'TYPE%'
    and p_size in (SIZE1, SIZE2, SIZE3, SIZE4, SIZE5, SIZE6, SIZE7, SIZE8)
    and ps_suppkey not in (
        select
            s_suppkey
        from
            SCHEMA.supplier
        where
            s_comment like '%Customer%Complaints%'
    )
group by
    p_brand,
    p_type,
    p_size
order by
    supplier_cnt desc,
    p_brand,
    p_type,
    p_size;
```

Q17 - Small-Quantity-Order Revenue Query determines how much average yearly revenue would be lost if orders were no longer filled for small quantities of certain parts. This may reduce overhead expenses by concentrating sales on larger shipments.

```
select
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    SCHEMA.lineitem,
    SCHEMA.part
where
    p_partkey = l_partkey
    and p_brand = 'BRAND'
    and p_container = 'CONTAINER'
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            SCHEMA.lineitem
        where
            l_partkey = p_partkey
    );
```

Q18 - Large Volume Customer Query ranks customers based on their having placed a large quantity order. Large quantity orders are defined as those orders whose total quantity is above a certain level.

```
select
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice,
    sum(l_quantity)
from
    SCHEMA.customer,
    SCHEMA.orders,
    SCHEMA.lineitem
where
    o_orderkey in (
        select
            l_orderkey
        from
            SCHEMA.lineitem
        group by
            l_orderkey having
                sum(l_quantity) > QUANTITY
    )
    and c_custkey = o_custkey
    and o_orderkey = l_orderkey
group by
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice
order by
    o_totalprice desc,
    o_orderdate;
```

Q19 - Discounted Revenue Query reports the gross discounted revenue attributed to the sale of selected parts handled in a particular manner. This query is an example of code as might be produced programmatically by a data mining tool.

```
select
    sum(l_extendedprice* (1 - l_discount)) as revenue
from
    SCHEMA.lineitem,
    SCHEMA.part
where
    (
        p_partkey = l_partkey
        and p_brand = 'BRAND1'
        and p_container in ( 'SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
        and l_quantity >= QUANTITY1 and l_quantity <= QUANTITY1 + 10
        and p_size between 1 and 5
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    )
    or
    (
        p_partkey = l_partkey
        and p_brand = 'BRAND2'
        and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
        and l_quantity >= QUANTITY2 and l_quantity <= QUANTITY2 + 10
        and p_size between 1 and 10
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    )
    or
    (
        p_partkey = l_partkey
        and p_brand = 'BRAND3'
        and p_container in ( 'LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
        and l_quantity >= QUANTITY3 and l_quantity <= QUANTITY3 + 10
        and p_size between 1 and 15
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    );
```

Q20 - Potential Part Promotion Query identifies suppliers in a particular nation having selected parts that may be can-didates for a promotional offer.

```
select
    s_name,
    s_address
from
    SCHEMA.supplier,
    SCHEMA.nation
where
    s_suppkey in (
        select
            ps_suppkey
        from
            SCHEMA.partsupp
        where
            ps_partkey in (
                select
                    p_partkey
                from
                    SCHEMA.part
                where
                    p_name like 'COLOR%'
            )
        and ps_availqty > (
            select
                0.5 * sum(l_quantity)
            from
                SCHEMA.lineitem
            where
                l_partkey = ps_partkey
                and l_suppkey = ps_suppkey
                and l_shipdate >= date 'DATE'
                and l_shipdate < date 'DATE' + interval '1' year
        )
    )
    and s_nationkey = n_nationkey
    and n_name = 'NATION'
order by
    s_name;
```

Q21 - Suppliers Who Kept Orders Waiting Query identifies certain suppliers who were not able to ship required parts in a timely manner.

```
select
    s_name,
    count(*) as numwait
from
    SCHEMA.supplier,
    SCHEMA.lineitem l1,
    SCHEMA.orders,
    SCHEMA.nation
where
    s_suppkey = l1.l_suppkey
    and o_orderkey = l1.l_orderkey
    and o_orderstatus = 'F'
    and l1.l_receiptdate > l1.l_commitdate
    and exists (
        select
            *
        from
            SCHEMA.lineitem l2
        where
            l2.l_orderkey = l1.l_orderkey
            and l2.l_suppkey <> l1.l_suppkey
    )
    and not exists (
        select
            *
        from
            SCHEMA.lineitem l3
        where
            l3.l_orderkey = l1.l_orderkey
            and l3.l_suppkey <> l1.l_suppkey
            and l3.l_receiptdate > l3.l_commitdate
    )
    and s_nationkey = n_nationkey
    and n_name = 'EGYPT'
group by
    s_name
order by
    numwait desc,
    s_name;
```

Q22 - Global Sales Opportunity Query identifies geographies where there are customers who may be likely to make a purchase.

```
select
    cntrycode,
    count(*) as numcust,
    sum(c_acctbal) as totacctbal
from (
    select
        substring(c_phone from 1 for 2) as cntrycode,
        c_acctbal
    from
        SCHEMA.customer
    where
        substring(c_phone from 1 for 2) in
            ('20', '40', '22', '30', '39', '42', '21')
        and c_acctbal > (
            select
                avg(c_acctbal)
            from
                SCHEMA.customer
            where
                c_acctbal > 0.00
                and substring(c_phone from 1 for 2) in
                    ('20', '40', '22', '30', '39', '42', '21')
        )
        and not exists (
            select
                *
            from
                SCHEMA.orders
            where
                o_custkey = c_custkey
        )
    ) as custsale
group by
    cntrycode
order by
    cntrycode;
```

Appendix D

Commands used to import the database to Amazon Relational Database Instance (RDS)

$ docker exec <*container name*> pg_dump -U <*username*> > <*output file path*>
$ psql -f ./<*output file path*> –host <*RDS instance endpoint*> –port <*port*> –username <*username*>

Appendix E

JSR223 PreProcessor script

Code used in the preprocessor to handle query 15 from the query set.

```
// Read the three lines from the CSV file
String queryPart1 = vars.get("QUERY_PART1");
String queryPart2 = vars.get("QUERY_PART2");
String queryPart3 = vars.get("QUERY_PART3");

// Combine the three lines to form the complete Query 15 variant
String fullQuery15 = queryPart1 + ";\n" + queryPart2 + ";\n" + queryPart3 + ";";

// Store the complete query in a JMeter variable
vars.put("FULL_QUERY_15", fullQuery15);
```

Appendix F

TPC-H Guidelines for Database size

The Scale Factor (SF) corresponds to the size of the data base in GB, and the Streams (S) to the minimum number of through put tests to execute.

SF	S(Streams)
1	2
10	3
30	4
100	5
300	6
1000	7
3000	8
10000	9
30000	10
100000	11

Figure F.1: Database size guidelines