Copyright Undertaking

This thesis is protected by copyright, with all rights reserved.

By reading and using the thesis, the reader understands and agrees to the following terms:

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the thesis.
2. The reader will use the thesis for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.
3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

IMPORTANT

If you have reasons to believe that any materials in this thesis are deemed not suitable to be distributed in this form, or a copyright owner having difficulty with the material being included in our database, please contact lbsys@polyu.edu.hk providing details. The Library will look into your claim and consider taking remedial action upon receipt of the written requests.

Pao Yue-kong Library, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

http://www.lib.polyu.edu.hk

VAULT: AN OPEN-SOURCE PARALLEL DATABASE AS A SERVICE

IP YAT FUNG

MPhil

The Hong Kong Polytechnic University

2019

The Hong Kong Polytechnic University

Department of Computing

Vault: An Open-source Parallel Database as a Service

Ip Yat Fung

A thesis submitted in partial fulfillment of the requirements

for the degree of

Master of Philosophy

May 2018

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

. . . .

Ip Yat Fung

May 2018

Abstract

In recent years, parallel processing technology has become remarkably popular for enterprise big data analytics. However, the traditional IT infrastructure sets a barrier to enterprise big data analytics because of its limitations on scalability and high total cost of ownership. Migrating the parallel database system to the cloud platform offers enterprises scale up or down on demand without consideration of on-site hardware investment (e.g., on-site hardware maintenance and repair). Besides, the multi-tenancy property in a cloud platform can minimise total cost of ownership by sharing the parallel database system among multiple tenants.

This thesis presents Vault, an open source cloud-based service which aims to provide **parallel database-as-a-service (PDaaS)** at a low operational cost with the service level agreement, SLA (i.e., a commitment governing the minimal level of service agreed between a service provider and tenants). Vault is built on top of the cloud platform, OpenStack, which is an open-source software that offers a cloud infrastructure for the parallel database system to carry out data analytics. With the advent of resource sharing in the multi-tenant environment, the service provider gains advantages of maximising resource utilisation and minimising the operational cost. Our experiments present that Vault serves tenants with only 55.2% of the requested nodes in OpenStack cloud while a 99% query-latency SLA is still guaranteed with high availability.

Acknowledgements

By taking this opportunity, I would like to express my thankfulness and appreciation for those people without whom it would not be possible to complete this thesis.

First of all, I wish to express my deepest gratitude to my former supervisor, Dr. Eric Lo for his guidance. He gave me lots of valuable advices and also questions to motivate me to see things from different perspectives. I greatly appreciate his support throughout the period of Mphil study.

Many thanks also to my chief supervisor, Dr. Ken Yiu who gave me many supports and constructive ideas in my last year study period.

I also would like to show my appreciation to my colleagues, Chris Liu and Lili Zhang who share their experiences and gave me suggestions with great generosity on this thesis.

Last but not least, I am very grateful for the unconditional support from my family who encourages me without expecting for anything in return. I also would like to thank my friends who provide me with continuous support and encouragement, especially in these years.

Table of contents

Declaration **i**

Abstract **iii**

Acknowledgements **v**

Table of contents **vii**

List of Figures **xi**

List of Tables **xv**

1 Introduction **1**

2 Background **5**

2.1 OpenStack 5

2.1.1 Core Services 5

2.1.2 Trove 6
2.2 Vertica 7
3 Vault 9
3.1 Tenant View 9
3.2 Service Provider View 11
3.3 System Overview 12
3.3.1 API Server 13
3.3.2 Dashboard 13
3.3.3 Backend Database 14
3.3.4 Tenant Activity Monitor 15
3.3.5 Deployment Advisor 15
3.3.6 Deployment Master 15
3.3.7 Query Router 16
3.4 Example Workflows 16
3.4.1 A Vertica Instance is Created 17
3.4.2 Latency SLA Guarantee of Tenants is Violated 18
3.4.3 A Tenant's Query Request is Submitted 20
4 Uniform Tenant Driven Design 21

4.1 Cluster Design and Tenant Placement 21

4.2 Query Routing 25

4.3 Elastic Scaling 26

4.4 SLA Maintenance 28

5 Experimental Evaluation 29

5.1 Experimental Design and Methodology 29

5.2 Evaluation under Different System Configurations 31

5.3 Evaluation under Different Tenant Characteristics 37

5.4 Evaluation under Different Active Tenant Ratio 40

5.5 Elastic Scaling Evaluation 42

6 Related Work 47

7 Conclusion 51

Bibliography 53

List of Figures

3.1 A Tenant Requesting for a 2-node Vertica Instance 10

3.2 Demonstration of a Vertica Instance Details 11

3.3 Real-Time Analytics on the Query Activity of a Tenant 12

3.4 Real-Time Analytics on the Performance of a Tenant Group . . . 13

3.5 System Architecture and Interaction between OpenStack and Trove 14

3.6 Sequence Diagram of Creating a Vertica Instance 18

3.7 Sequence Diagram of Executing Consolidation 19

4.1 Demonstration of Solving the Latency SLA Violation Issue in a Consolidation Cycle 23

5.1 Varying Memory Size 32

(a) Consolidation Effectiveness 32

(b) Execution Time for Grouping Tenants 32

(c) Average Tenant Group Size 32
(d) Query Error 32
5.2 Varying Epoch Size 34
(a) Consolidation Effectiveness 34
(b) Execution Time for Grouping Tenants 34
(c) Average Tenant Group Size 34
5.3 Varying Query Latency SLA 35
(a) Consolidation Effectiveness 35
(b) Execution Time for Grouping Tenants 35
(c) Average Tenant Group Size 35
5.4 Varying Replication Factor 36
(a) Consolidation Effectiveness 36
(b) Execution Time for Grouping Tenants 36
(c) Average Tenant Group Size 36
5.5 Varying Number of Tenants 38
(a) Consolidation Effectiveness 38
(b) Execution Time for Grouping Tenants 38
(c) Average Tenant Group Size 38
5.6 Varying Tenant Distribution 39

(a) Consolidation Effectiveness 39

(b) Execution Time for Grouping Tenants 39

(c) Average Tenant Group Size 39

5.7 Varying Active Tenant Ratio 41

(a) Consolidation Effectiveness 41

(b) Execution Time for Grouping Tenants 41

(c) Average Tenant Group Size 41

5.8 Execution Time for Tenant Migration in OpenStack Cloud 43

5.9 Elastic Scaling in a Tenant group 45

(a) RT-TTP (with elastic scaling) 45

(b) Query Performance (with elastic scaling) 45

(c) RT-TTP (without elastic scaling) 45

(d) Query Performance (without elastic scaling) 45

List of Tables

4.1 A Workflow Example in Separating Tenants into groups 22

5.1 Experimental Parameters under Different System Configurations 31

5.2 Experimental Parameters under Different Tenant Characteristics 37

Chapter 1

Introduction

In recent years, parallel processing technology has become remarkably popular for enterprise big data analytics [1, 2]. However, the traditional IT infrastructure sets a barrier to enterprise big data analytics because of its limitations on scalability and high total cost of ownership. In fact, there are 80% of enterprises intended to move big data analytics to the cloud platform [3]. Migrating the parallel database system to the cloud platform offers enterprises scale up or down on demand without consideration of on-site hardware investment (e.g., on-site hardware maintenance and repair) [4]. Besides, the multi-tenancy property in a cloud platform can minimise total cost of ownership by sharing the parallel database system among multiple tenants [5]. As a result, the rise of cloud computing can deliver an efficient scalable and cost-effective solution for big data analytics.

This thesis presents Vault, an open source cloud-based service which aims to provide **parallel database-as-a-service (PDaaS)** at a low operational cost

with the service level agreement, SLA (i.e., a commitment governing the minimal level of service agreed between a service provider and tenants). Vault is built on top of the cloud platform, OpenStack, which is an open-source software that offers a cloud infrastructure for the parallel database system to carry out data analytics [6]. We deploy the multi-tenancy model to provision Vertica instances (i.e., the virtual machines installing Vertica parallel database system) in OpenStack cloud. By consolidating tenants onto the shared Vertica instances in the cloud, multiple co-located tenants can share the virtual computing, storage and networking resources. With the advent of resource sharing in the multi-tenant environment, the service provider gains advantages of maximising resource utilisation and minimising the operational cost.

Despite the benefits offered by the multi-tenancy model, it unavoidably poses a challenge of satisfying the SLA guarantee for every tenant [7]. Vault is designed for analytical purposes specialising in online analytical processing (OLAP) environment. The concurrent analytical workloads posed by multiple co-located tenants would result in rigorous resource competition or even resource shortage, especially when most of the analytical workloads are I/O intensive or memory-intensive. In other words, the emergence of concurrent analytical query processing within a shared Vertica instance is highly possible to violate the SLA. In coping with this challenge, Vault decides how tenants co-locate in the group to share resources by identifying whose workload pattern is complementary to each other (e.g., one is in Asia time zone, another is in US time zone). This multi-tenancy arrangement approach avoids resource competition between the co-located tenants, especially when processing concurrent analytical queries. And therefore it helps maintain the SLA guarantee for every tenant. As an example, the tenants

T_1, T_4 and T_7 also have the same parallelism requirement (i.e., the number of requested nodes per Vertica instance) with a complementary workload patterns. According to the above mentioned example, they could be packed into the same Vertica instance for resource sharing after consolidation. In reality, the approach of multi-tenancy arrangement can be applied to different applications of big data analytics like intelligent transportation and financial market trading [8]. For example, there are frequent and large amount of financial data retrieved from the market regularly and the automated trading systems can make effective decisions to discover the financial opportunities and risks from those historical data. As the automated trading systems run with a fixed behaviour pattern (i.e., run in a fixed stock market trading session), the regular and recurrent financial activities encourage the formation of complementary activity patterns and therefore the multi-tenancy arrangement approach is feasible in real world.

We introduce Vault - a real implementation of PDaaS in this thesis, whereas the theoretical foundation of Vault comes from Thrifty [9]. We first present the architecture of Vault with the example workflow (Chapter 3) to show the interactions between Vault and OpenStack in a more practical perspective. And then, we discuss the design principle of Vault (Chapter 4) and present how this principle can be applied to reduce the operational cost and guarantee the performance of tenants in OpenStack cloud. Meanwhile, we also explore the details about elastic scaling approach employed by Vault which satisfies the SLA guarantee at runtime. Last but not least, a set of experiments is carried out to mainly evaluate the consolidation effectiveness (i.e., overall percentage of the requested nodes saved) and the performance guarantee of Vault under different configurations (Chapter 5). Our experiments present that Vault serves tenants with only 55.2%

of the requested nodes in OpenStack cloud while a 99% query-latency SLA is still guaranteed with high availability.

Chapter 2

Background

2.1 OpenStack

Vault is designed to run on OpenStack, an open source software for building and managing private and public clouds. Due to the collaboration of developers from community or enterprises, OpenStack has a fast-paced development on providing a feature-riched, massively scalable and reliable cloud platform.

2.1.1 Core Services

To set up OpenStack environment, there are 6 OpenStack core services that should be installed, which consist of:

(i) **Nova:** supports the deployment and management of Nova instances (i.e., virtual machines provisioned in OpenStack cloud) in the cloud.

(ii) **Keystone:** authenticates and authorises the service provider and tenants to access OpenStack services in the cloud. Additionally, it also provides the endpoints service (i.e., seeking a contact point for accessing the requested OpenStack service) to facilitate the interactions between Vault and other existing OpenStack services.

(iii) **Glance:** manages the virtual machine image (e.g., Ubuntu, Fedora) for Nova services. Glance stores different images in the cloud, and Nova must retrieve the requested image from Glance in order to launch a Nova instance.

(iv) **Neutron:** manipulates the virtual network devices and services, such as router, firewall and DHCP server. In particular, Neutron mainly interacts with Nova to provide the network connectivity for Nova instances.

(v) **Cinder:** provides additional volume attachment on Nova instances. The volume could be attached or detached from Nova instance to adjust the disk space allocated on the instance.

(vi) **Horizon:** serves as a web-based user interface to ease the use of different OpenStack services such as Nova and Cinder.

2.1.2 Trove

In addition to the core services, Vault relies on OpenStack Trove to offer PDaaS in the cloud. Trove is an open source project with the aim of providing highly-reliable and scalable databases-as-service (DaaS) in OpenStack cloud [10]. Trove is mainly composed of:

(i) **Message Bus:** enables delivering messages between Trove components. A RabbitMQ message system is commonly deployed in Trove to serve as a transport mechanism.

(ii) **Task Manager:** listens on Trove's Message Bus for the messages that execute operations such as database backup or database restoration. It performs these database-specific operations by communicating with other existing OpenStack services. For instance, Trove's Task Manager performs a database expansion operation by requesting Cinder to resize the volume attached to Vertica instance.

(iii) **Guest Agent:** runs a database engine (i.e., starting/stopping the database server) inside a Vertica instance [11] and listens on Trove's Message Bus for the messages sent from other Trove components. As an example, Trove's Guest Agent receives the message sent from Trove's Task Manager to execute the user creation operation. The guest agent then creates a new database user for the database server running in that Vertica instance.

(iv) **API server:** is the entry point of Trove which receives the external requests and passes on the requests to Trove's Task Manager or Guest Agent for executing the corresponding database-specific operations.

2.2 Vertica

As mentioned above, Trove is important for Vault to provision Vertica instances since it requests Trove to install the parallel database system inside Nova instance(s). For this reason, the parallel database system should be supported

by Trove. There are several parallel database systems available for OLAP, such as Greenplum, Netezza and Vertica. But only Vertica is currently supported by Trove.

In Vertica, it aims to improve performance through parallelism of database operations like queries, bulk insertions or deletions [12]. Vertica supports building multi-tenant environment in which multiple tenants can share the resources on the same Vertica instance. Besides, the security concern posed by the multi-tenancy model [13] can be addressed by Vertica. Vertica ensures each individual tenant can only access to their own data in order to secure the privacy of tenants.

On the basis of the above considerations, Vertica should be the most ideal parallel database system for Vault to provide PDaaS in OpenStack cloud.

Chapter 3

Vault

Vault is an open source project with the aim of offering PDaaS in OpenStack cloud. In this chapter, we first introduce the fundamental services provided by Vault from the perspective of service provider or tenant. Then, the architecture of Vault will be discussed in detail through the example workflows.

3.1 Tenant View

Vault provides tenants PDaaS with the following main functions:

(i) **Creating a Vertica Instance:** tenants can create and configure their Vertica instance. For example, a tenant creates a 2-node Vertica instance by specifying the tenant name, node quantity and flavor (i.e., OpenStack resources given in the instance such as memory and vCPU) as shown in figure 3.1.

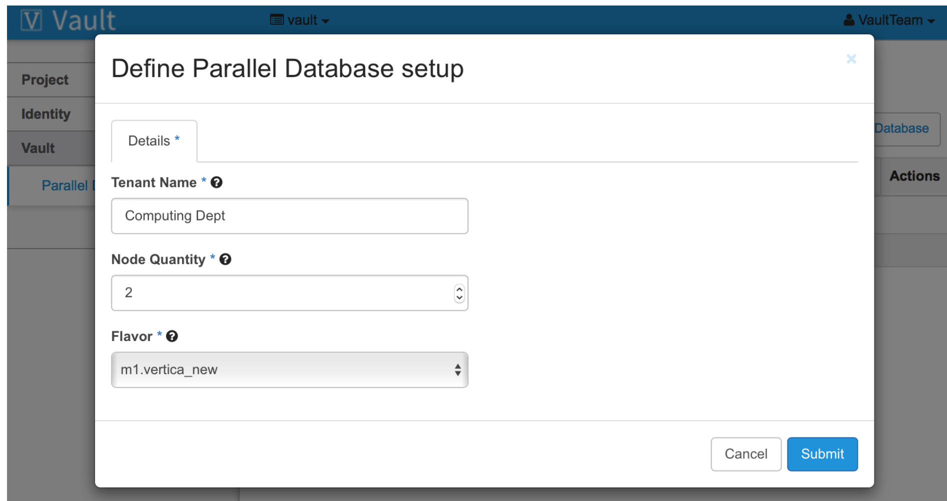


Figure 3.1. A Tenant Requesting for a 2-node Vertica Instance

(ii) **Monitoring and Managing Vertica Instances:** the critical information of a Vertica instance is shown in figure 3.2. After a tenant creates a Vertica instance, the tenant can review the information of the instance, such as the status of Vertica instance. A tenant can also decide to create more Vertica instance(s) or delete the unnecessary Vertica instance(s) on demand.

(iii) **Billing Tenants:** the system usage of each tenant is identified such as how much CPU, memory and storage are used. Those information could be tracked and collected for billing.

(iv) **Querying Vertica Instance:** a unique JDBC connection string (see Connection String part in figure 3.2) is defined for every Vertica instance. The connection string enables tenants to establish a database connection to Vertica instance for submitting queries. There are three properties of the connection string including host address, database name and tenant ID which specify the credentials to authenticate the connection to the database.

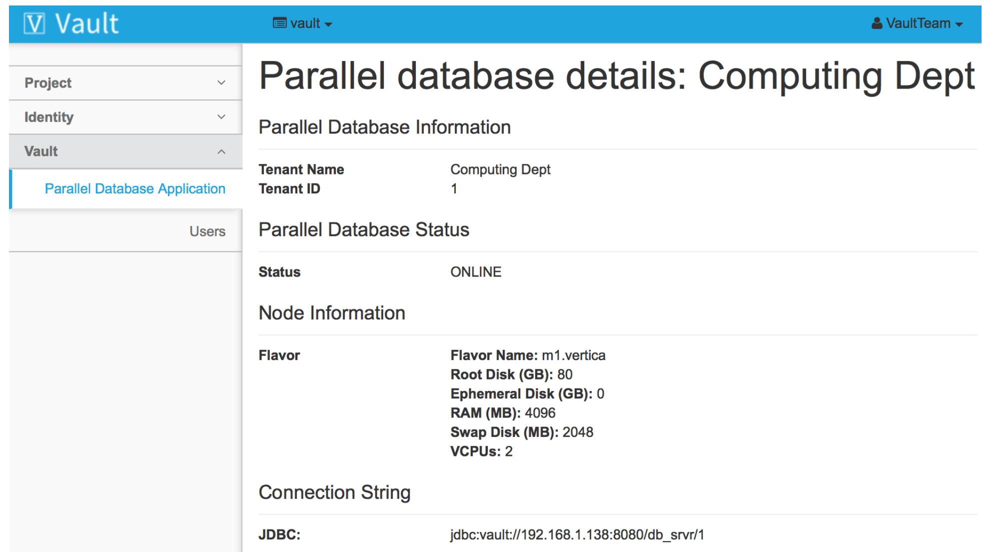


Figure 3.2. Demonstration of a Vertica Instance Details

3.2 Service Provider View

The primary functions provisioned for the service provider in Vault are as follows:

(i) **Monitoring Query Status of a Tenant:** the changes of the tenant activity are identified continuously over time. The chart below shows the tenant activity dynamically based on the collected query history. Figure 3.3 shows that the value of tenant activity always stayed at 0 which means the tenant kept in idle status. After that, the activity value rose as 1 while the tenant was submitting queries to the database.

(ii) **Monitoring Status of Tenant Groups:** the integrated result of the tenant activities in each tenant group is measured in terms of performance. In figure 3.4, the performance area chart shows the query finished execution

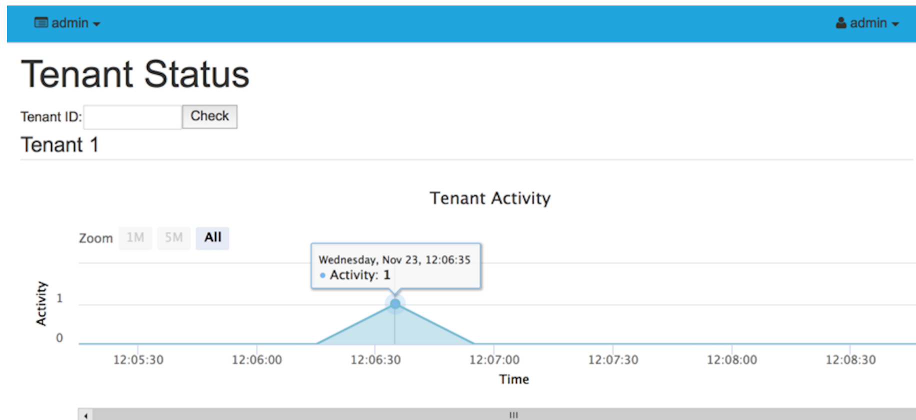


Figure 3.3. Real-Time Analytics on the Query Activity of a Tenant

times of the tenant group when it measured in an isolation environment with a real-time update. As the chart shows that the performance of tenant group TG_1 is 2.0 which indicates that the query of TG_1 has finished execution 2 times slower than when it measured in an isolated environment.

(iii) **Managing Tenants and Tenant groups:** the service provider could review the critical information of all tenants and tenant groups in OpenStack cloud, for example, the member list of every tenant group.

3.3 System Overview

Vault integrates with Trove and other OpenStack core services to provide PDaaS as shown in figure 3.5. In particular, a unique endpoint is specified for every OpenStack service. Vault is then allowed to interact with Trove and other OpenStack services through their endpoints which avoid reimplementation of the existing OpenStack functionalities. To illustrate how Vault interacts with Trove

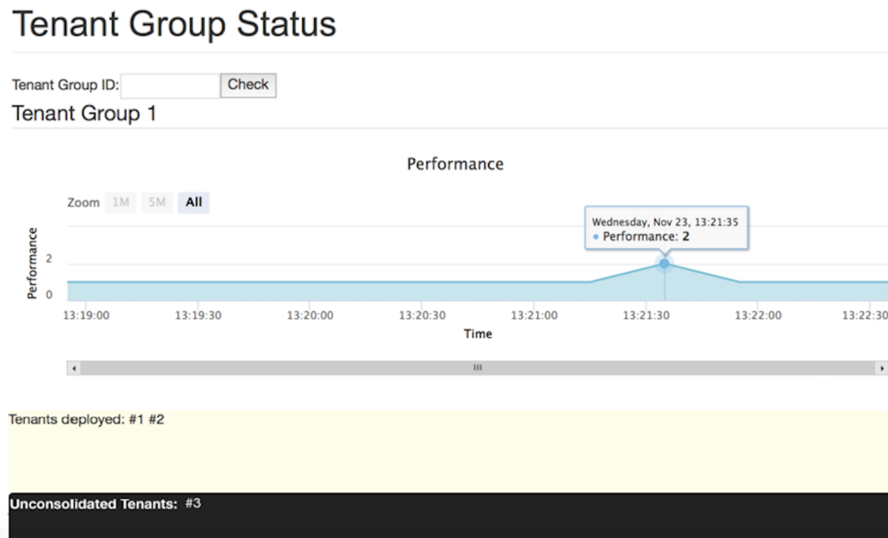


Figure 3.4. Real-Time Analytics on the Performance of a Tenant Group

and OpenStack, we first introduce seven key components of Vault:

3.3.1 API Server

Vault's API server is the entry point to receive the requests from the tenants and the service provider. It is also responsible to authenticate the credentials of the tenants and the service provider against OpenStack Keystone in order to get the permission for further access to the underlying Vault services.

3.3.2 Dashboard

Vault's Dashboard is built on OpenStack Horizon which eases the use of Vault services through the user-friendly graphical user interface.

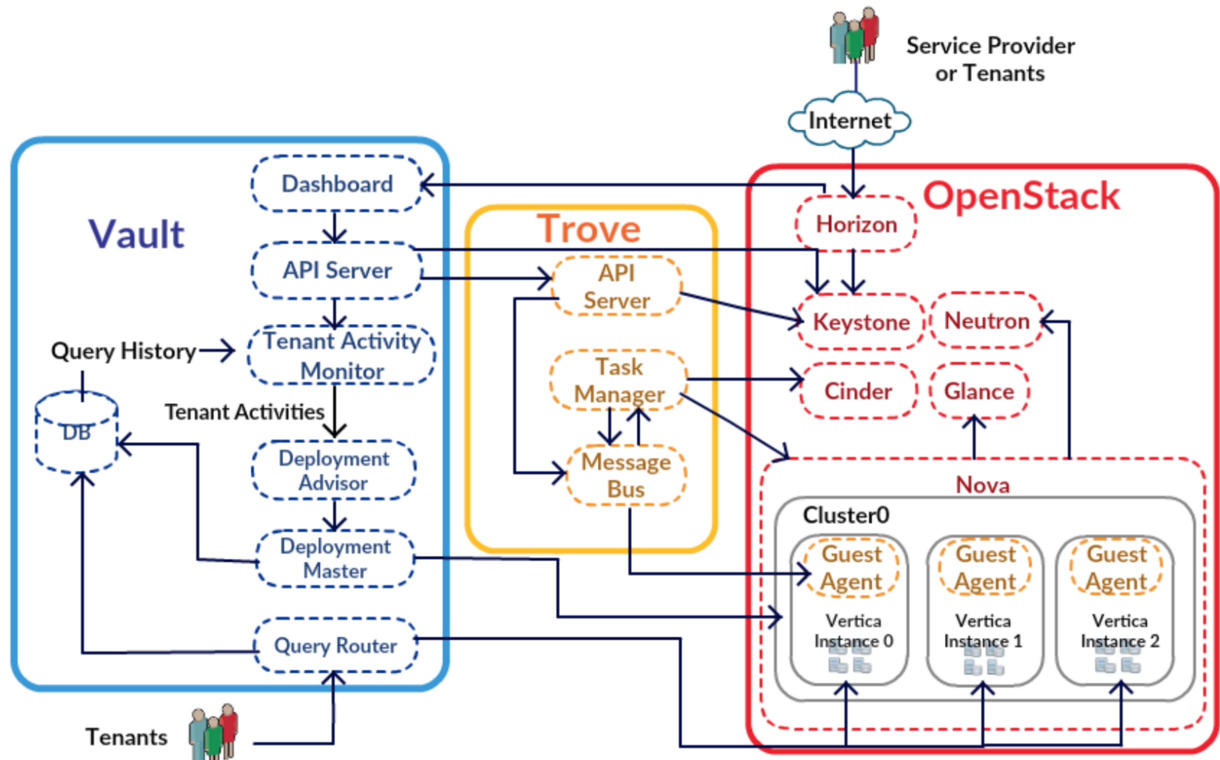


Figure 3.5. System Architecture and Interaction between OpenStack and Trove

3.3.3 Backend Database

Vault's Backend Database stores the information displayed on Vault's Dashboard such as the details of Vertica instances in OpenStack cloud. The database also stores the query history (i.e., tracking and recording all submitted SQL queries) of each tenant that can be used for consolidation.

3.3.4 Tenant Activity Monitor

Vault's Tenant Activity Monitor is responsible for collecting the tenants' query history from Vault's Backend Database to derive the tenant activities. The derived information is intended to support Vault's Deployment Advisor in further analysis.

3.3.5 Deployment Advisor

Vault's Deployment Advisor can optimise OpenStack resource deployment in the cloud by grouping the tenants for sharing resources on the same cluster (i.e., a group of one or more Vertica instances). With three inputs, tenant activities, replication factor and query-latency SLA guarantee, a deployment plan can be devised accordingly. A deployment plan can be broken into two parts, (1) Cluster Design and (2) Tenant Placement (Section 4.1), for example, $\{TG_0$: $([V_0,V_1,V_2],[T_1,T_2,T_3,T_4,T_7,T_8])$, ...$\}$. As the cluster design defines how Vertica instances can be arranged into clusters, the above example proposes to join Vertica instances V_0, V_1 and V_2 to the same cluster, say $Cluster_1$. In addition, the tenant placement specifies which tenants should be deployed on the cluster. In this case, the tenants T_1, T_2, T_3, T_4, T_7 and T_8 are grouped and put on $Cluster_1$ to share the resources among Vertica instances V_0, V_1 and V_2.

3.3.6 Deployment Master

Vault's Deployment Master put the deployment plan into practice by executing consolidation in OpenStack cloud. During the consolidation process, it

always involves the migration of tenants' data and maybe involve the creation of new cluster(s) for deploying the tenants in newly-formed group(s). Likewise, Vault's Deployment Master executes re-consolidation process regularly to sustain the query-latency SLA over a long period of time.

3.3.7 Query Router

Vault's Query Router determines to which Vertica instance in OpenStack cloud a query should be routed. It routes queries based on the categories of the SQL query, while there are two major categories (Section 3.4.3): IUD Queries (i.e., Insert, Update and Delete operations) and SELECT Queries. In particular, Vault applies the load balancing policy (Section 4.3) to SELECT queries for balancing the workload across Vertica instances in the cluster. Additionally, Vault exploits ROWA (Read-One Write-All) protocol to guarantee a strong mutual data consistency among Vertica instances in the cluster [14].

3.4 Example Workflows

In this section, we aim to bring a better understanding of Vault architecture and investigate how Vault fully utilises the existing OpenStack services for providing PDaaS. Therefore, we introduce the mechanism of Vault by discussing the scenarios when (1) a Vertica instance is created (see figure 3.6), (2) the latency SLA guarantee of tenants is violated (see figure 3.7), (3) a tenant's query request is submitted.

3.4.1 A Vertica Instance is Created

To begin with, a tenant requires to log in OpenStack Horizon with authentication credentials (step 1). The provided credentials are used to request OpenStack Keystone to obtain an authentication token as a key for accessing Vault, Trove or other OpenStack services in the following steps (step 2). Vault's Dashboard is a user-interface plugin for Horizon to enable PDaaS in OpenStack cloud, and hence the tenant can request to launch a Vertica instance through the user-interface (step 3). Vault's Dashboard makes an API call directly with the authentication token to invoke Vault's API server to create a Vertica instance (step 4). Vault validates the authentication token against Keystone to permit access to Vault services (step 5) and then invokes Trove's API server to create a Vertica instance (step 6). Trove also needs to validate the authentication token against Keystone to permit access to Trove services (step 7), after that, Trove's API server posts a message of Vertica instance creation to Trove's Message Bus (step 8). Then, Trove's Task Manager receives the message from the message bus and start the instance creation procedure (step 9). The task manager first requests OpenStack Nova to create Nova instances (step 10). After validating the authentication token (step 11), Nova locates the virtual machine image in OpenStack Glance to create Nova instances (step 12). Likewise, Glance needs to validate against Keystone before delivering the virtual machine image to Nova (step 13). Nova also requests OpenStack Neutron for the networking services (step 14). After validating the authentication token (step 15), Neutron attaches Nova instances to the network. In addition, the task manager also requests OpenStack Cinder for adding persistent storage (i.e., Cinder volume) to Nova instances (step 16). Cinder can create and mount a volume on Nova instances

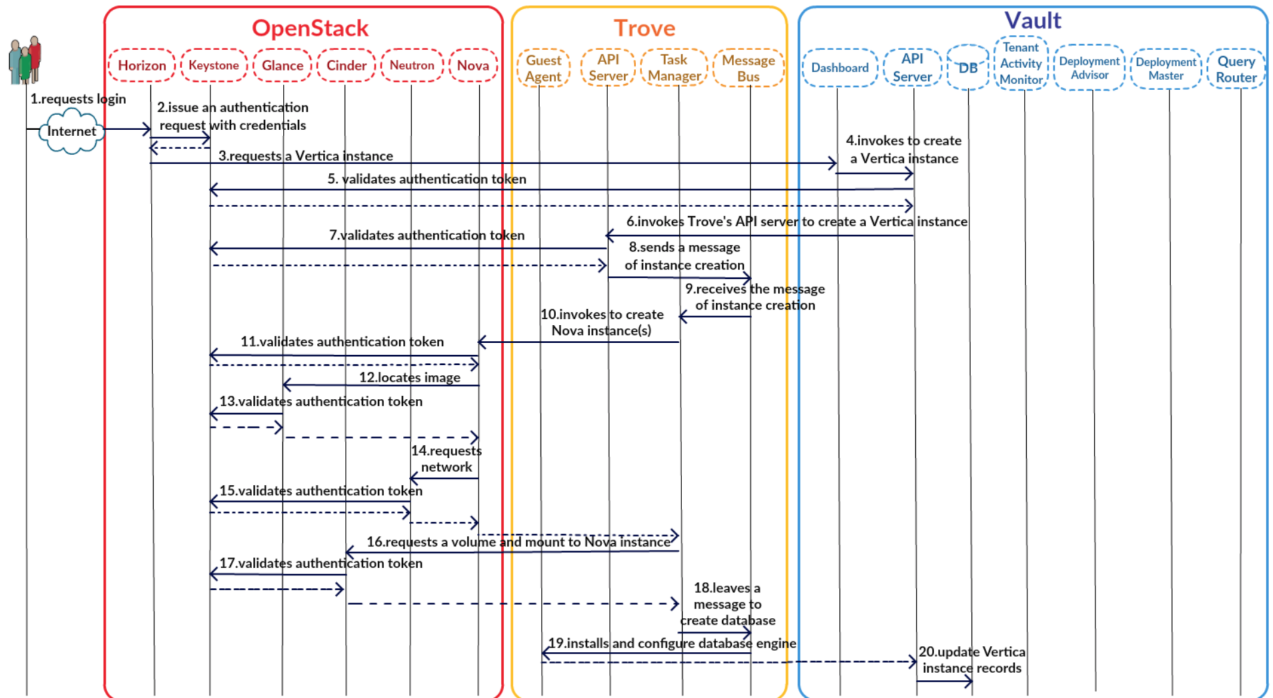


Figure 3.6. Sequence Diagram of Creating a Vertica Instance

after validating authentication token (step 17). The task manager leaves a message to invoke Trove's Guest Agent to create a Vertica database in the instances (step 18). The guest agent receives the message to install and configure the database engine in the instances (step 19). Finally, Vertica instance is created and Vault's API server updates Vertica instance information in Vault's Backend Database (step 20).

3.4.2 Latency SLA Guarantee of Tenants is Violated

As Vault aims at maintaining the latency SLA guarantee in the system, it executes (re-)consolidation process regularly to handle the arrival/departure of

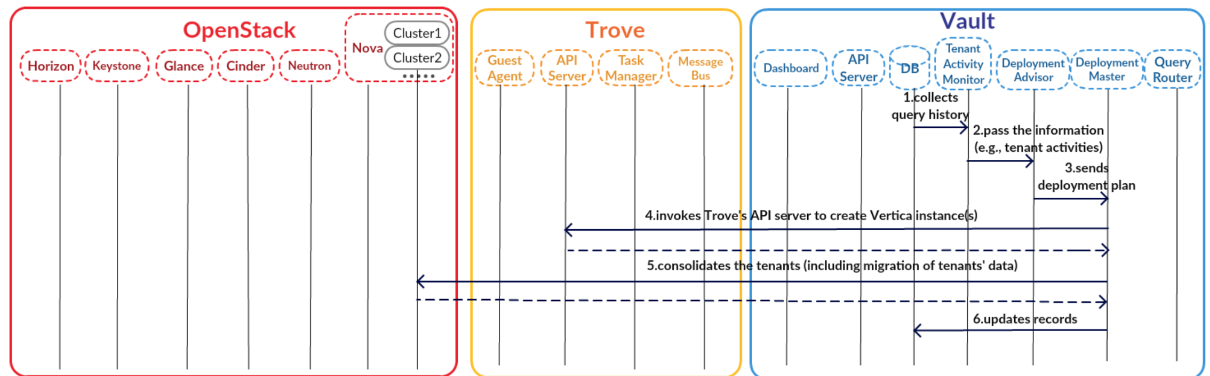


Figure 3.7. Sequence Diagram of Executing Consolidation

tenants or the tenants with changed activity patterns. First, Vault's Tenant Activity Monitor collects the query history from Vault's Backend Database (step 1). The monitor derives the tenant activities from the query history and transfers the information to Vault's Deployment Advisor for further analysis, including tenant activities, replication factor and latency SLA guarantee (step 2). The deployment advisor generates the deployment plan based on the received information and then passes it to Vault's Deployment Master for executing consolidation (step 3). During executing consolidation process, the deployment master may create Vertica instances for the new cluster(s) to deploy the tenants in newly-formed group(s) (step 4). In particular, the process of creating Vertica instances is the same as described as the step 6 to step 20 in figure 3.6. Then, the deployment master consolidates the tenants into the clusters by migrating the tenants' data to the corresponding cluster as devised by the deployment plan (step 5). Lastly, it updates the records of tenants and Vertica instances in Vault's Backend Database (step 6).

3.4.3 A Tenant's Query Request is Submitted

(1) **IUD Queries:**

A tenant first establishes a connection to Vault's Query Router for submitting a SQL Insert query (step 1). With ROWA, the query router identifies and builds database connection to all Vertica instances V_i within the cluster (step 2). This enables the tenants to maintain the data consistency among Vertica instances by routing the SQL Insert query to all identified Vertica instances (i.e., Write(V_i), $\forall V_i$) in the corresponding cluster (step 3). After finishing the query execution, it is recorded in Vault's Backend Database for keeping track of the tenant's query history (step 4).

(2) **SELECT Queries:**

A tenant establishes a connection to Vault's Query Router for submitting a SQL Select query (step 1). With the query routing algorithm mentioned in Section 4.3, the SQL Select query can be routed to the available Vertica instance in the cluster by Vault's Query Router (step 2) to ensure the query performance under the latency SLA guarantee in OpenStack cloud. Finally, the SQL Select query is recorded in Vault's Backend Database (step 3) for keeping track of the tenant's query history and for the use of deriving tenant activities when executing consolidation.

Chapter 4

Uniform Tenant Driven Design

On the basis of the design principle called **Uniform Tenant Driven Design (UTDD)**, it designs the way of how Vault separates tenants into groups for resource sharing, while also considering the guarantee of query-latency SLA [9]. Besides, UTDD ensures that the co-located tenants could be served exclusively through proper query routing. In summary, UTDD consists of the following parts (1) Cluster Design and Tenant Placement, (2) Query Routing, (3) SLA Maintenance and (4) Elastic Scaling:

4.1 Cluster Design and Tenant Placement

UTDD suggests utilising a high-availability cluster for serving each tenant group, so there will be one or more Vertica instances residing in the same cluster to provide an increased reliance on Vault. Specifically, there will be R (i.e., replication factor specified by the service provider) Vertica instances in the cluster,

Tenant Group	TG_0	TG_1	TG_2	G_4^1
Hosting (Loop0)	...	...	...	$T_1 - T_{10}$
Hosting (Loop1)	T_2	$-$	$-$	$T_1, T_3 - T_{10}$
...	...	...	$-$	...
Hosting (Loop4)	$T_2 - T_5$	$-$	$-$	$T_1, T_6 - T_{10}$
Hosting (Loop5)	$T_1 - T_5$	$-$	$-$	$T_6 - T_{10}$
Hosting (Loop6)	$T_1 - T_5$	T_6	$-$	$T_7 - T_{10}$
...	...	...	$-$	...
Hosting (Loop9)	$T_1 - T_5$	T_6, T_7, T_9, T_{10}	$-$	T_8
Hosting (Loop10)	$T_1 - T_5$	T_6, T_7, T_9, T_{10}	T_8	$-$

Table 4.1. A Workflow Example in Separating Tenants into groups

each of them contains the data of all tenants who are members of the tenant group. As a result, the co-located tenants can access any Vertica instances in the cluster depending on their workload, so UTDD ensures not only high availability but also load balancing. The cluster design defines how Vertica instances can be arranged into clusters. The cluster is designed to best suit the workload of tenants in each tenant group, so the tenants who have higher workloads may need their own dedicated Vertica instance.

Tenant placement concerns which tenants should be deployed on the particular cluster [9]. First of all, the formation of tenant groups mainly depends on their parallelism requirement and query history to separate the tenants into groups. Different from Thrfity, it also specifies the matching between clusters and tenant groups with the objective of minimising the frequency of tenant migration between the clusters (i.e., bulk data loading). The above arrangement details about cluster design and tenant placement will be devised as a deployment plan.

Let's illustrate the concepts of cluster design and tenant placement with the

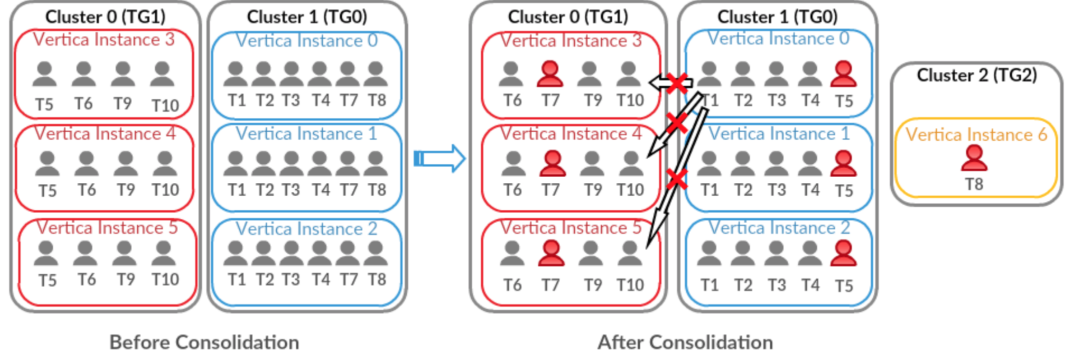


Figure 4.1. Demonstration of Solving the Latency SLA Violation Issue in a Consolidation Cycle

idea of UTDD's tenant grouping presented in Algorithm 1. In our example, there are ten tenants T_1-T_{10} and each of them requests a 4-node Vertica instance with the same flavor (i.e., Flavor ID 1). The parallelism requirement of tenants T_1-T_{10} is the same, so they will be grouped into the same initial group (i.e., G_4^1) in the beginning. As tenant T_2 is the least active tenant, T_2 is put into tenant group TG_0 first as shown in table 4.1. Supposing the workload patterns of $T_2 - T_5$ are complementary to each other, so they will be consolidated into TG_0 without leading to the drop of TTP while TTP indicates the time percentage of R or less concurrent active tenants submitting queries in a group. At loop 5 in table 4.1, supposing that it identifies T_1 and T_{10} are the members of $T_{candidate}$ who can also minimise the increase in time percentage of the maximum number of concurrent active tenants in TG_0. At this time, we need to pay attention to the issue of tenant migration. Figure 4.1 presents that the previous deployment plan is devised as $\{TG_0$: $([V_0,V_1,V_2],[T_1,T_2,T_3,T_4,T_7,T_8])$, TG_1: $([V_3,V_4,V_5],[T_5,T_6,T_9,T_{10}])$, $...\}$. Since T_1 is originally located in TG_0, T_1 should be the best tenant put into TG_0 to avoid bulk loading T_1's data from $Cluster_1$ to $Cluster_0$ (for 3 times) as

shown in figure 4.1. The remaining tenants $T_6 - T_{10}$ can no longer be put into TG_0 because TG_0's TTP is lower than 99% after placing any of them, and a new tenant group TG_1 is created as a result. Based on the same tenant grouping principle, T_6, T_7, T_9 and T_{10} are packed into TG_1. As T_8 is a noisy neighbour for TG_0 or TG_1, it would be excluded and formed as another tenant group TG_2. Hence, three tenant groups, TG_0, TG_1 and TG_2, are eventually formed as shown in figure 4.1.

Algorithm 1 UTDD Tenant Grouping Algorithm

Input: Tenants T, Replication Factor R, Query Latency SLA Guarantee P%

1: Pack all T with the same parallelism requirement N and flavor type F into the same initial group G_N^F;
2: Let i = 1;
3: **for** each initial group G_N^F **do**
4: Create a tenant group TG_i;
5: Determine the potential tenant(s) $T_{candidate} \in G_N^F$ that minimise the increase in time percentage of the maximum number of concurrent active tenants in TG_i;
6: **if** the number of $T_{candidate}$ >1 **then**
7: Identify the tenant $T_{best} \in T_{candidate}$ who is co-located with the most tenants among TG_i before consolidation;
8: **else**
9: $T_{best} = T_{candidate}$
10: **if** TG_i's TTP is still greater than P% after placing tenant T_{best} to TG_i **then**
11: Put T_{best} to TG_i and exclude T_{best} from G_N^F;
12: Goto Line 5;
13: **else**
14: i++;
15: Goto Line 4;

When deciding the corresponding cluster for each tenant group, we will compare with the last tenant placement in the cluster. The new grouping result presents that TG_0 contains the tenants $T_1 - T_5$. Compared with the new grouping result, there is only one common tenant, T_5, placed in $Cluster_0$ (before consolidation). By contrast, there are four common tenants, $T_1 - T_4$, placed in

$Cluster_1$ (before consolidation). Since $Cluster_1$ contains the greatest number of common tenants in TG_0, TG_0 should be assigned to $Cluster_1$. Note that there is no cluster available for TG_2, therefore a new cluster $Cluster_2$ is created for hosting T_8 in TG_2.

In most cases, the cluster design exploits the replication factor R to determine the cluster size (i.e., the number of Vertica instances) where R=3 is applied in our scenario. The existing clusters $Cluster_0$ and $Cluster_1$ are reserved for serving TG_1 and TG_0 with three Vertica instances. However, in order to reduce the resource redundancy, the cluster size should be equal to the size of the tenant group only if the number of tenants in that tenant group is smaller than R. By this reason, only one Vertica instance in $Cluster_2$ is designed to serve T_8 because only one tenant is put in TG_2. After consolidation, Vault minimises the total number of nodes required to build in OpenStack cloud, and therefore Vault provides PDaaS under a low operation cost with latency SLA guarantee.

4.2 Query Routing

The principle of query routing algorithm behind UTDD is to route a tenant to a free (or available) Vertica instance, which can balance the distribution of workloads across Vertica instances among the group. Let's use some examples to illustrate the idea of query routing in Vault presented in Algorithm 2. For the most simple case, if there is any free Vertica instance in the group, tenant T_i can be routed to the first identified, free Vertica instance V_y. In case Vertica instances are all busy in the group, the coming active tenants will be routed to one of Vertica instances for concurrent query execution. Actually, it is known as

an overactive tenant issue caused by the tenants whose activities deviate from the query history at runtime. Like those irregular tenants whose activities become unpredictable and they will be active as located in the time zone different from before which results in an excessive workload for the group. In addition, those tenants who become extremely active all the time will also give heavy workload for the group. To reduce the query latency when processing concurrent queries, it requires to balance the workload among Vertica instances in the group. Unlike Thrifty, the load balancing decision depends on identifying the amount of query processing inside each Vertica instance. The tenant will be routed to a Vertica instance $V_{candidate}$ with the least workload to achieve load balancing in the group. If $V_{candidate}$ is more than one, the tenant will be routed to the first identified Vertica instance V_f identified in $V_{candidate}$. It is noticeable that if tenant T_i has an existing connection to a Vertica instance V_x built before, the incoming queries submitted by tenant T_i (maybe from different users) will be routed to Vertica instance V_x. With this query routing algorithm, each tenant can often be served by a dedicated Vertica instance exclusively that fits their parallelism request.

4.3 Elastic Scaling

Since some tenants may become exceptionally active different from the past tenant activities at runtime, there is much chance that the number of concurrent active tenants will be more than R at runtime. In other words, there will be a frequent emergence of overactive tenants in the tenant group. When the emergence of overactive tenants accumulates more than 1% of time in the past 24 hours, it causes the RT-TTP of the tenant group to drop below the SLA

Algorithm 2 UTDD Query Routing Algorithm

Input: Tenant T_i, Query Q

```
1: if T_i has established a connection to a Vertica instance V_x then
2:     Route Q to V_x;
3: else
4:     if there is any free Vertica instances V_y then
5:         Route Q to V_y;
6:     else
7:         Identify a Vertica instance V_candidate that has the
            minimum number of processing queries;
8:         if there is more than one V_candidate then
9:             Pick the first identified Vertica instance V_f ∈ V_candidate;
10:            V_best = V_f;
11:        else
12:            V_best = V_candidate;
13:        Route Q to V_best for concurrent query execution;
```

guarantee of 99%. Continuing the example in figure 8, the idea of Vault's Elastic Scaling will be illustrated as the following procedures. To begin with, it first identifies the tenants in TG_0 whose activities deviate from the query history and put them into a new group TG_3. Then, it creates a new Vertica instance and migrates the data of those tenants to the newly provisioned Vertica instance. As those tenants are moved into TG_3, they would be removed from TG_0 which implies their tenant activities would also be excluded from TG_0, so the RT-TTP of TG_0 will be restored to 99% or above to meet the SLA guarantee of the tenant group. Instead of loading the entire tenant group's data, only the data of those overactive tenants needs to be migrated to the new Vertica instance. This will be a benefit to save the data loading time and thus reduce the time for restoring the performance level of the tenant group.

4.4 SLA Maintenance

The SLA guarantee provides transparency between the service provider and the tenants to clearly define the required level of service in Vault. For the query-latency SLA guarantee, more than R concurrent active tenants submitting queries would lead to increased query latency. Therefore, the implication of concurrent query execution occurring in a Vertica instance infers that the tenants could not be exclusively served by a dedicated Vertica instance. In case the latency SLA guarantee is 99%, when more than R tenants submits the concurrent queries and last for more than 1% of time percentage in the group, the RT-TTP (Run Time Total Time Percentage - for TTP measurement at runtime) will fall below 99%. UTDD tries to solve the latency SLA violation issue in every (re)-consolidation cycle to prevent recurrent resource competition between tenants in OpenStack cloud.

Besides, Vault provides the migration SLA guarantee to ensure the performance of data migration. In other words, the cost of data migration is guaranteed to be minimised in order to reduce the impact on query execution. There are two circumstances that involve data migration process in Vault which includes executing consolidation or elastic scaling. It is guaranteed that the query execution and data migration could be processed simultaneously without downtime during executing consolidation or elastic scaling.

Chapter 5

Experimental Evaluation

The purpose of this chapter is to share our experience in running Vault as PDaaS in OpenStack cloud. For all our experiments, the tenants submit read-only query/batch queries (i.e., a bundle of queries) to Vertica instances, and the query history is tracked and transformed to tenant activity logs continuously. For the sake of a better understanding of the experimental observations, we analyse the results with the aid of tenant activity logs found in Vault's backend database. We present different experimental results to fully evaluate Vault performance in real world practice.

5.1 Experimental Design and Methodology

All the experiments are performed with Nova instances (1 vCPU, 8 GB RAM, 20 GB Root Disk) running as Vertica instances in OpenStack cloud. We deploy OpenStack in multi-node configuration with the use of 4 physical machines

(24 cores, 6.7 TB hard disk and 260 GB RAM per physical machine) comprising of a controller node and compute nodes (also acts as storage nodes). In particular, a single physical machine is configured as the controller node for supplying shared services in the cloud (e.g., dashboard, identity service), while the remaining physical machines are used for hosting Vertica instances and offering persistent storage to them. To ensure the performance of the instances, we disable the default overcommitting ratio in OpenStack and set the overcommitting ratio of CPU and RAM to 1:1 which means the total number of Vertica instances created in OpenStack cloud would not exceed the total number of physical cores in our experimental environment.

In our experiments, there are 3 tenant types (i.e., a tenant may request 2/3/4-node Vertica instance) and each tenant holds 5 GB TPC-H data. Each tenant picks either time offset +0, + 8 or +16 to determine the tenants who may be located at different timezones. To imitate a real tenant behaviour, the tenants obey the following rules to submit query/batch queries in our experiments. First of all, each tenant has U active users where $1 \leq U \leq 3$. And each user submits a random TPC-H query or a batch of K random TPC-H queries regularly to a Vertica instance where $1 \leq K \leq 3$. After submitting a single query or batch queries, the user becomes idle. Once the processing of query or batch queries is completed, the user becomes active again after L seconds where $120 \leq L \leq 300$. Table 5.1 and 5.2 represent the experimental parameters for the assessment of Vault performance while the default values are displayed in boldface.

The experiments are carried out to examine the idea of taking advantage of the complementary tenant activities for consolidation. We can put the idea into practice by carrying out each experiment for a day in order to collect 24-hour

tenant activities throughout daytime and nighttime.

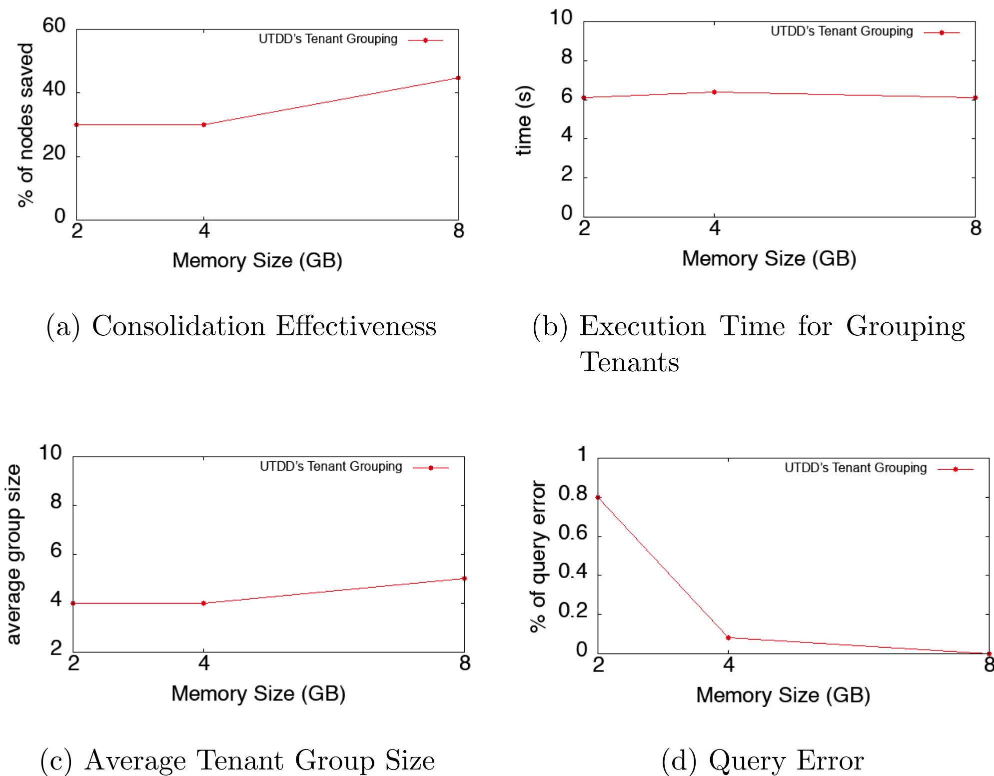
5.2 Evaluation under Different System Configurations

System Configurations	
Experimental Parameter	Range
Memory Size	2GB, 4GB, **8GB**
Epoch Size	1s, 5s, **10s**, 30s, 90s, 600s
Query Latency SLA	80%, 93%, 95%, 97%, **99%**
Replication Factor	1, 2, **3**, 4

Table 5.1. Experimental Parameters under Different System Configurations

Varying Memory Size: Vault pricing is determined by the instance type chosen in OpenStack cloud. In numerous instance types, a memory optimised instance features higher memory to vCPU ratio which gives advantage for relational database server [15], like Vertica. The experiment below finds out the most appropriate memory to vCPU ratio that should be chosen for optimising performance and cost in OpenStack cloud. Figure 5.1a shows the consolidation effectiveness increases from 29.9% to 44.8% when the memory size increases from 2 GB to 8 GB. Since higher memory to vCPU ratio can deliver faster performance for query processing, it helps in reducing the time percentage of the emergence of overactive tenants. For this reason, it is most likely to bring about the result that the tenant activities are in complementary distribution for consolidation, so it results in a higher consolidation effectiveness. At the same time, figure 5.1c also supports the findings that more tenants could be packed into the same group for sharing OpenStack resources due to the growth of memory size. On the other hand, it is relatively easy to fill up the memory and causes the query failure when there is only 2 GB or 4 GB memory size in Vertica instance.

The percentage of query error drops to 0% when the memory size increases to 8 GB. Hence, we adopt the memory optimised instance with 8 GB memory size in OpenStack cloud. Figure 5.1c shows that the execution time of UTDD's tenant grouping keeps around 6 seconds because the tenant grouping algorithm needs to scan through the same duration (i.e., 24 hrs) of the tenant activities to form the resulting tenant groups even though memory size is increased.



(a) Consolidation Effectiveness

(b) Execution Time for Grouping Tenants

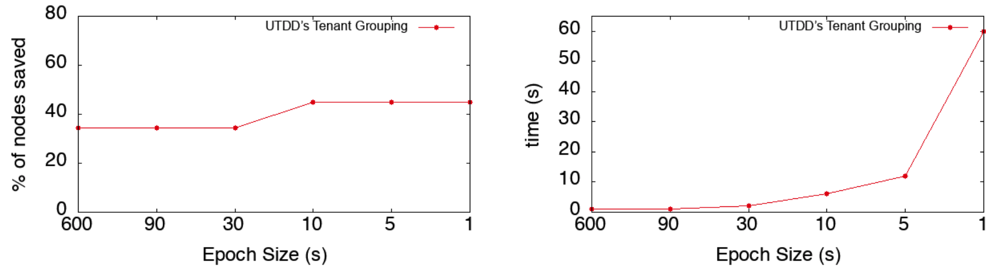
(c) Average Tenant Group Size

(d) Query Error

Figure 5.1. Varying Memory Size

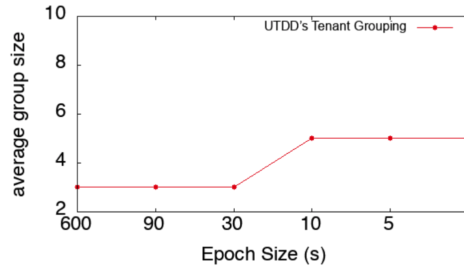
Varying Epoch Size: As mentioned earlier, UTDD's tenant grouping is based on the tenant activities of which it is derived from the history (i.e., query logs stored in Vault's backend database) for a specific duration (i.e., 24 hours). The

derivation of tenant activities is on a time basis while the adoption of different time epochs would produce slightly different derivation results, and so it may cause a different result for UTDD's tenant grouping. The experiment evaluates how long the time epoch should be used to derive tenant activities from the history. Figure 5.2a shows that the consolidation effectiveness grows from 34.3% to 44.8% when the time epoch drops from 600s to 10s. Figure 5.2b shows that UTDD's tenant grouping could be finished within 10s if the time epoch is more than or equal to 10s and the execution time of UTDD's tenant grouping sharply rises when time epoch falls to 1s. Despite the fact that a larger time epoch leads to a shorter execution time of tenant grouping, it also has its drawback if a large time epoch is used in Vault. A larger time epoch could only roughly derive the tenant activities while a shorter time epoch could bring tenant activities to become more closer to the real situation. For example, a tenant only submits a query to a Vertica instance in the past 10 minutes (i.e., 600s) and it just takes 25s to process the query. When deriving the tenant activity with 600s time epoch, the derivation result still indicates that the tenant is active all the time in OpenStack cloud within the epoch range. If a shorter time epoch is used, say, 10s time epoch derives the tenant activity to indicate the tenant is active for 30 seconds (i.e., three 10s time epochs) and inactive for the remaining 570 seconds. Figure 5.2c shows that the average group size grows because a shorter time epoch can bring about the result that the tenant activities are in complementary distribution, and hence it will be easier to place more tenants into the same group. With the above reasons, the ideal epoch size should be 10s to ensure the consolidation effectiveness and the execution time of tenant grouping are at a reasonable level.



(a) Consolidation Effectiveness

(b) Execution Time for Grouping Tenants

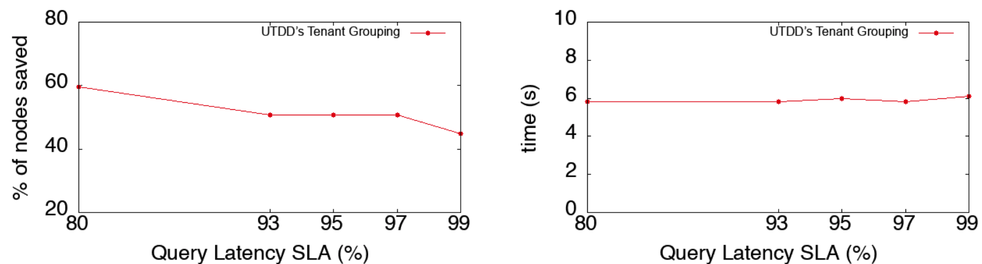


(c) Average Tenant Group Size

Figure 5.2. Varying Epoch Size

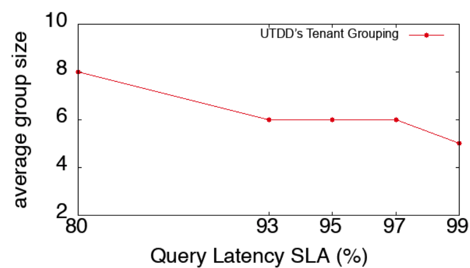
Varying Query Latency SLA: This experiment evaluates how the query-latency SLA promised between the service provider and tenants will influence the consolidation effectiveness in OpenStack cloud. Figure 5.3a shows that UTDD's tenant grouping ensures the consolidation effectiveness of Vault could be at least 44.8%. With a loose query-latency SLA guarantee, the consolidation effectiveness can be higher and even be 59.7% when only 80% query-latency SLA guarantee is promised. The reason is that a loose query-latency SLA guarantee could tolerate a higher time percentage of emergence of overactive tenants, hence overactive tenants will be easier to be packed into the same tenant group. Figure 5.3c

shows the average group size grows from 5 to 8 when the query-latency SLA falls from 99% to 80% because more tenants could be packed into the same group with a loose query-latency SLA guarantee. Figure 5.3b shows that the execution time of UTDD's tenant grouping keeps around 6 seconds because the tenant grouping process requires to scan through the same duration (i.e., 24 hrs) of the tenant activities every time to form the resulting tenant groups, even though the query-latency SLA is varied.



(a) Consolidation Effectiveness

(b) Execution Time for Grouping Tenants

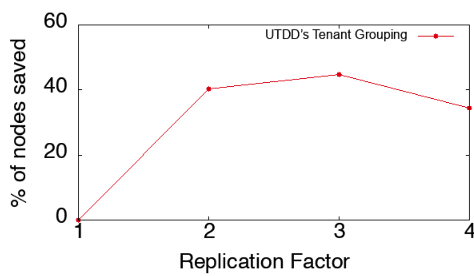


(c) Average Tenant Group Size

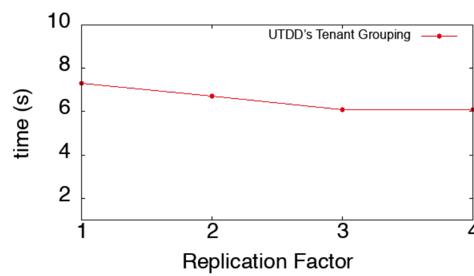
Figure 5.3. Varying Query Latency SLA

Varying Replication Factor:

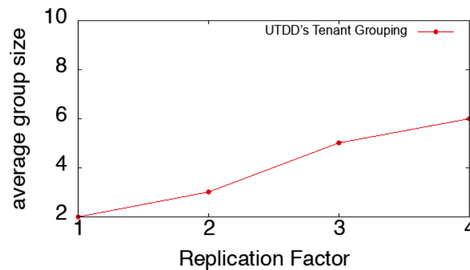
This experiment evaluates the ideal replication factor R to enable load balancing in our experiment. The replication factor decides the number of Vertica instances in each cluster, so a higher replication factor supports more concurrent active tenants in OpenStack cloud.



(a) Consolidation Effectiveness



(b) Execution Time for Grouping Tenants



(c) Average Tenant Group Size

Figure 5.4. Varying Replication Factor

Figure 5.4a shows that the consolidation effectiveness increases dramatically from 0% to 44.8% when R increases from 1 to 3, because more Vertica instances are available to handle the queries from the concurrent active tenants. The higher value of R can be more likely to guarantee that the number of concurrent active tenants in a group would not exceed R for P% of time. Figure 5.4c shows

the average group size grows from 2 to 5 accordingly which verify that more tenants are consolidated into the same tenant group with more Vertica instances provided. Although the average group size still grows to 6 when R is 4, the consolidation effectiveness drops back to 34.3% as shown in figure 5.4a because too many Vertica instances provided in the same group would cause resource redundancy. In most of the time, three Vertica instances are able to handle the queries from the concurrent active tenants, and so the extra Vertica instance becomes idle which causes the drop of consolidation effectiveness. Figure 5.4b shows that the execution time of UTDD's tenant grouping keeps around 6-7 seconds because the tenant grouping algorithm need to scan through the same duration (i.e., 24 hrs) of the tenant activities to form the resulting tenant groups even though R is increased.

5.3 Evaluation under Different Tenant Characteristics

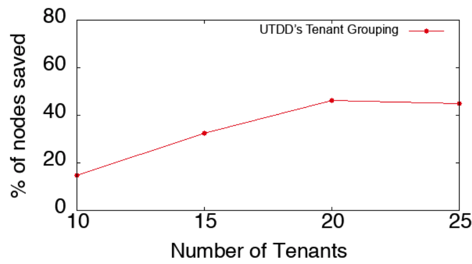
Tenant Characteristics	
Experimental Parameter	Range
Tenant Distribution	0.3, 0.6, **0.9**
Number of Tenants	15, 20, **25**

Table 5.2. Experimental Parameters under Different Tenant Characteristics

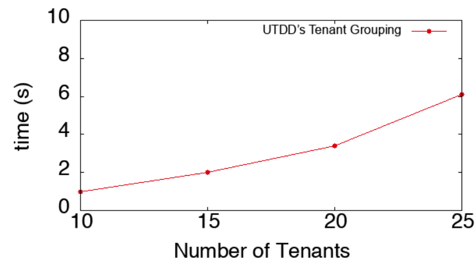
Varying Number of Tenants:

This experiment evaluates the efficiency of resource sharing in OpenStack cloud to meet the growing number of tenants. Figure 5.5a shows a significant increase of consolidation effectiveness from 14.8% to 32.5% when the number of

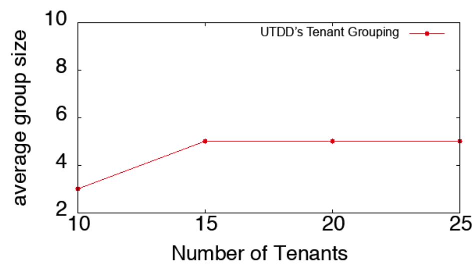
tenants increases from 10 to 15. Figure 5.5c shows the growth in average size of the tenant group which supports the observation that more tenants are packed into the group.



(a) Consolidation Effectiveness



(b) Execution Time for Grouping Tenants



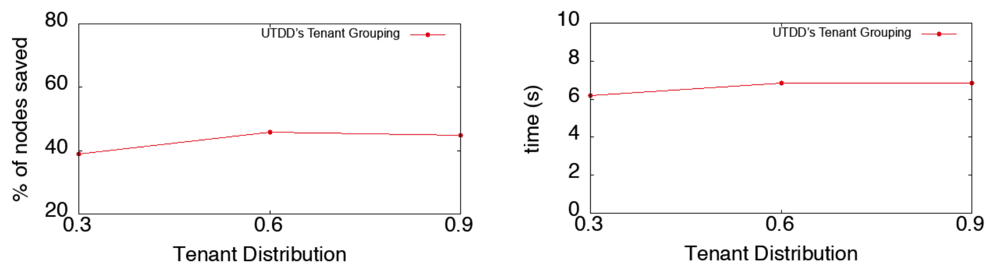
(c) Average Tenant Group Size

Figure 5.5. Varying Number of Tenants

Although the consolidation effectiveness still gradually increases to 44.8% when the number of tenants increases from 15 to 25, the average group size remains unchanged. To investigate this reason, we checked the backend database of Vault and found that the number of tenant group is slightly increased by 1 when the number of tenants is increased from 15 to 20 or from 20 to 25. This finding supports the result of a gradual increase in consolidation effectiveness

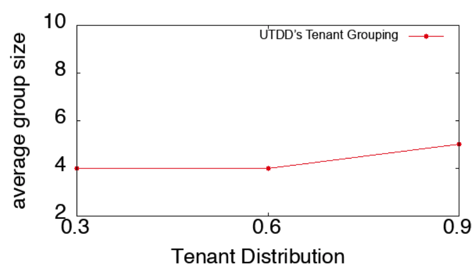
when more tenants participate in the cloud. As the average group size is greater than R, the increasing number of tenant group is favourable for better resource sharing in the cloud and therefore the consolidation effectiveness still increases gradually. Figure 5.5c shows the execution time of UTDD's tenant grouping steadily increases because more tenants are required to be analysed for filtering the overactive tenants and choosing the most suitable tenant group for each tenant.

Varying Tenant Distribution: In our experiment, it comprises of 3 tenant



(a) Consolidation Effectiveness

(b) Execution Time for Grouping Tenants



(c) Average Tenant Group Size

Figure 5.6. Varying Tenant Distribution

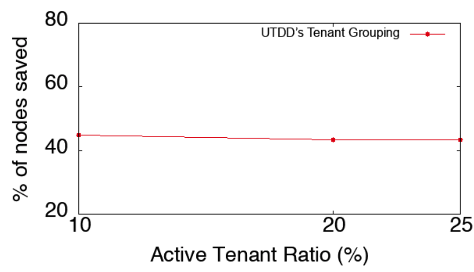
types including the tenant who can request a 2-node, 3-node or 4-node Vertica

instance in OpenStack cloud. Concerning the tenant distribution, the number of tenants of each tenant type is determined by the Zipf distribution with parameters n and α. The parameter n is the total number of tenants in OpenStack cloud, and the parameter $0 < \alpha < 1$ which determines the shape of Zipf distribution. The greater value of α results in a greater difference of the number of tenants among different tenant types, and vice versa. Figure 5.6a shows that there is no significant change in the consolidation effectiveness with an increasing α (i.e., tenant distribution). It indicates that there is no relation between tenant distribution and consolidation effectiveness in Vault. Figure 5.6c shows that the average group size also keeps unchanged with varying tenant distribution which supports the conclusion that tenant distribution should not be a consideration for UTTD's tenant grouping.

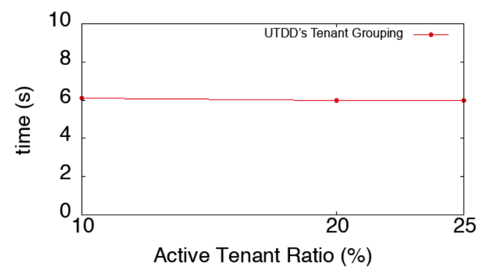
5.4 Evaluation under Different Active Tenant Ratio

In this section, the experiment exploits the same default values listed in table 5.1 and 5.2 to evaluate the capability of handling overactive tenants in OpenStack cloud. As we concentrate the study of consolidation effectiveness under a higher active tenant ratio, the tenants could only get either time offset +0 (i.e., tenants are from England) or +12 (i.e., tenants are from New Zealand), and therefore it leads to 20% active tenant ratio. To obtain a higher active tenant ratio, the tenants could get either time offset +0 or +12 without lunch hour which results in 25% tenant active ratio.

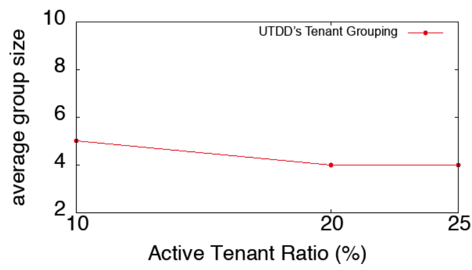
When the active tenant ratio is high up to 25%, the number of concurrent active tenants should be theoretically increased from 3 to 6 on average in each



(a) Consolidation Effectiveness



(b) Execution Time for Grouping Tenants



(c) Average Tenant Group Size

Figure 5.7. Varying Active Tenant Ratio

time epoch. Owing that the principle of consolidation is to promise the number of concurrent active tenants in a tenant group should not exceed R for P% of time, more concurrent active tenants pose difficulty in packing them into the same tenant group. Although fewer tenants can be put into the same tenant group as shown in figure 5.7c, the consolidation effectiveness can also maintained above 40% when active tenant ratio is increased. So, we ensure that Vault can save at least 40% of the total requested nodes in OpenStack cloud even though there is a certain amount of overactive tenants.

5.5 Elastic Scaling Evaluation

In order to evaluate the influence of Vault's elastic scaling approach adopted in OpenStack cloud, it is necessary to focus on one of the tenant groups and observe its behaviour over time. The chosen tenant group involves 6 tenants that request 3-node Vertica instances while it is a new tenant group formed after consolidation (with R=3). At the outset, the RT-TTP of the tenant group was 100% which means there were at most three concurrent active tenants from the last consolidation execution time till now. At time W, three tenants started to submit queries continuously to those three Vertica instances in the cluster. As those three tenants were allocated to be individually served by a separate Vertica instance, it would not lead to the drop of RT-TTP. At time X, an extra tenant (i.e., the fourth active tenant) submitted queries concurrently with the other three tenants. As all Vertica instances served those three active tenants, the extra tenant was allocated to be served by Vertica Instance V_0 (i.e. one of the busy Vertica instances). As a result, the query performance on V_0 was 1.3 as

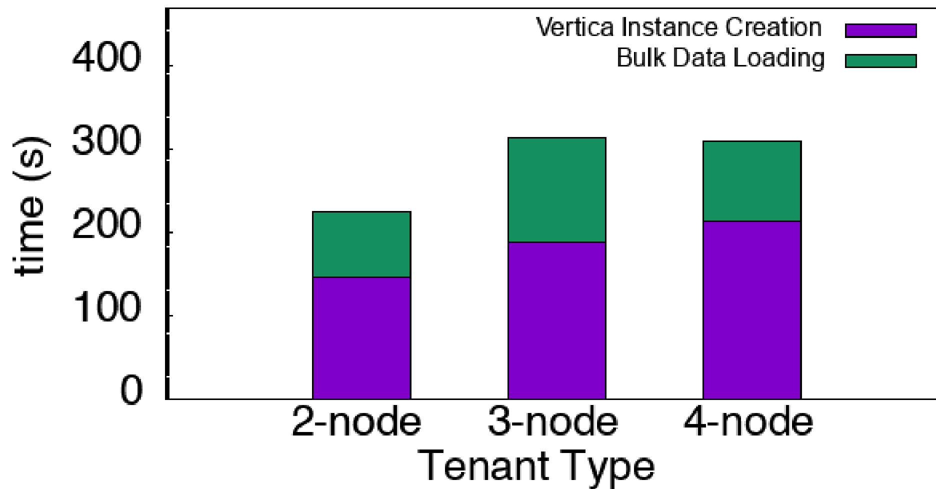
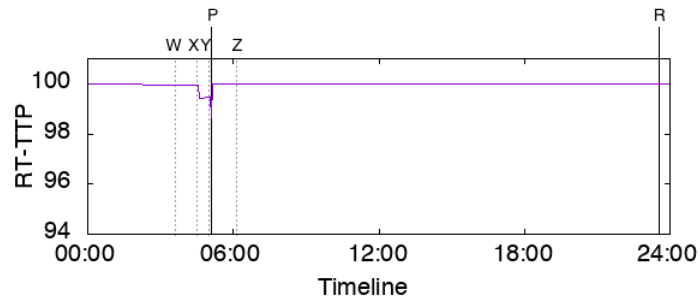


Figure 5.8. Execution Time for Tenant Migration in OpenStack Cloud

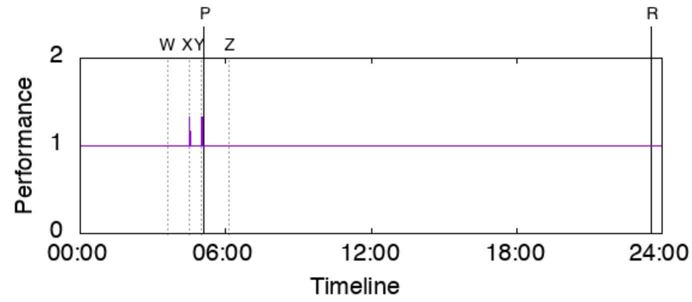
shown in figure 5.9b which means the query would be 1.3 slower than processing the query in a free Vertica instance. The extra tenant's query could not be processed exclusively, hence the RT-TTP of the tenant group fell to 99.2%.

The impact of applying Vault's elastic scaling in OpenStack cloud could be apparently observed started from time Y. During the period from time Y to time Z, an extra tenant became active once again and therefore four concurrent active tenants started to submit queries constantly. As a consequence, the tenant group accumulated more than 1% of time more than three concurrent active tenants submitting queries from the last consolidation execution time till now. For this reason, the RT-TTP of the tenant group further dropped over 99% which violated latency SLA guarantee. To meet the SLA guarantee, Vault started elastic scaling by identifying the overactive tenant(s) in the group first. Then, Vault prepared a new Vertica instance and loaded the overactive tenant(s)' data to the new Vertica instance. Vault spent total execution time of 300s in elastic scaling by this time. In fact, the total execution time required to carry out elastic scaling

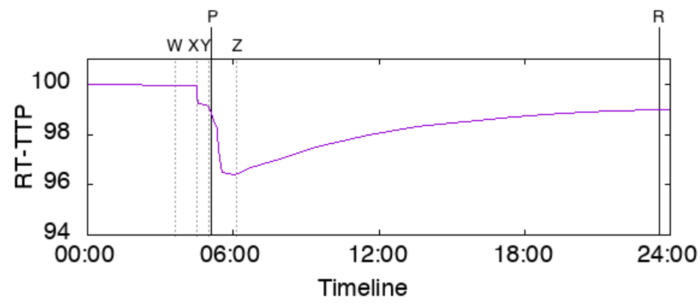
mainly depends on the tenant types. Figure 5.8 shows that the instance creation time increases with the increasing number of requested nodes, in other words, the total execution time taken in executing elastic scaling tends to increase when more nodes is requested by a tenant. At time P, the new Vertica instance was well-prepared for the overactive tenant(s), so the incoming queries submitted by the overactive tenant(s) could be routed to the new Vertica instance. Due to the fact that the tenant group excluded the activities of the overactive tenant(s) after elastic scaling, the RT-TTP of the tenant group reverted to above 99% at the same time. As the overactive tenant(s) was moved to the new group, even though those four tenants were concurrently active from time P to time Z, the RT-TTP of the tenant group still kept above 99%. In contrast, disabling elastic scaling in Vault would be difficult to meet the SLA guarantee. Without elastic scaling, the RT-TTP of the tenant group dropped over 99% to hit a low of 96.1% between time Y and time Z. After time Z, there were at most only three concurrent active tenants submitting queries. As time elapsed, the total time percentage of having more than three concurrent active tenants was decreased. In consequence, the RT-TTP of the tenant group gradually reverted to 99% or above but it took about 16 hours as shown in figure 5.9c. The significant time difference between time P and R is the time saved after adopting elastic scaling approach in OpenStack cloud.



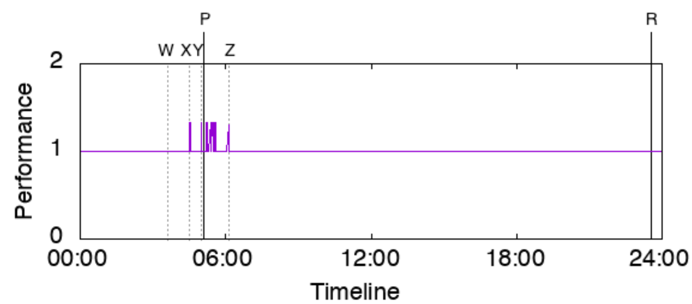
(a) RT-TTP (with elastic scaling)



(b) Query Performance (with elastic scaling)



(c) RT-TTP (without elastic scaling)



(d) Query Performance (without elastic scaling)

Figure 5.9. Elastic Scaling in a Tenant group

Chapter 6

Related Work

Cloud Database Projects There are some existing projects which provides database-as-a-service (DaaS) including (1) Amazon Relational Database Service (RDS) and (2) Microsoft SQLVM:

(1) Amazon RDS is designed to run on Amazon Web Services Cloud with a goal similar to Trove which aims to relieve the administrative burden by automating the tasks like backup and replication, and simplifies the use of relational database including provisioning or scaling RDS database instances in cloud. Amazon RDS provides the data migration service with no downtime which means the source database is still operational during the database migration process just like Vault. However, Amazon RDS generally enables resource sharing at the infrastructure level [4, 16]. As a consequence, the database creation request from each tenant can only bring an individual RDS database instance which limits the possibility of consolidation in cloud. For this reason, Amazon has developed and offered Amazon Aurora (i.e., a relational database compatible with MySQL and Post-

greSQL) to provide DaaS [17] and enable the possibility of resource sharing at VM level. For example, MySQL enables partitioning tables across instances by implementing sharding solution. To reduce resource consumption in cloud, hundreds or more of MySQL shards (instances) can be consolidated to a single Aurora instance, however, the outcome of consolidation effectiveness is determined by the number of MySQL shards owned by the tenant. By contrast, Vault can consolidate different tenants to a Vertica instance based on their parallelism requirement, the tenants who request a Vertica instance with fewer nodes (e.g., a 2-node Vertica instance) can also result in a high consolidation effectiveness in cloud and therefore Vault employs a more reliable and cost-effective approach to provide database services.

(2) Microsoft SQLVM [18] provides a cost-effective strategy for the service provider to offer multi-tenant DaaS in cloud. SQLVM enables sharing resources effectively among the tenants with the promise of reservation of key resources (i.e., CPU and memory) to guarantee sufficient resources provisioning for the co-located tenants. Narasayya et al. [19] proposes the consolidation concern which is how to share buffer pool memory among tenants with SLA. As buffer pool keeps the hot data pages, it reduces frequent disk access and therefore improves the query performance. SQLVM is designed to share buffer pool memory among the co-located tenants after consolidation, but it guarantees the tenants can be served as if the service provider reserves a dedicated, promised size of buffer pool memory for each of them to maintain the query performance. Similar to the case of memory reservation, Das et al. [20] proposes how SQLVM shares CPU effectively among the tenants. After undergoing consolidation, each tenant is promised to be allocated a minimum CPU utilisation. In other words, it meets tenants' demand

for CPU even though the tenants contend for CPU under a shared environment, especially for executing the CPU-intensive tasks. Different from the works mentioned above, Vault concerns the tenant consolidation challenge of identifying the complementary workload patterns among tenants, but not on the considerations of a particular resource. Therefore, Vault promises that each tenant can be served by dedicated resources as if they can use a Vertica instance exclusively.

Load Balancing In order to achieve a better resource utilisation, load balancing is a way to distribute the workload effectively among the nodes in cloud [21]. Load balancing could be classified as static load balancing and dynamic load balancing [22, 23]. For static load balancing algorithm, it does not consider the system state (e.g., live connections, server workloads) while it defines the load allocation strategy by using prior knowledge of the system like the processors performance and the amount of nodes in the system. Many cloud databases like Xeround has proposed a **round robin scheduling algorithm** to balance the workload among the nodes. Actually, it is a static load balancing algorithm to select nodes within a group in sequential order. But, it cannot give an efficient resource utilisation comparing to dynamic load balancing because the load balancing decision does not involve the current system state and so the load might not be allocated to the lightest node in the system. In contrast, dynamic load balancing algorithm directs load to the node with the system state consideration in order to meet the circumstances change. Vault employs the **least loaded server policy** while it is a dynamic load balancing algorithm to balance the workload among Vertica instances by identifying the least workload among them as discussed in section 4.2. In fact, the **least connection scheduling algorithm** [24] is also an alternative dynamic load balancing algorithm that could

be applied to Vault. It directs the load to the node with the least number of live connections. However, the live connections information exchange between query routers may lead to an extra communication delay, so the least loaded server policy is more favourable in Vault.

Elastic Scaling Elastic scaling is one of the key features in cloud computing to meet the workload changes at any point of time by dynamically growing or shrinking the resources in the cloud. CloudScale [25] and PRESS [26] are some examples of systems which automate elastic scaling in the cloud to ensure the system performance with a minimum resource cost. Both systems adjust the resource allocation to satisfy time-varying resource demand by predicting the cloud resource demand based on historical data. However, they require to train an accurate prediction model with the resource usage information for avoiding over-estimation (which causes resource redundancy) or under-estimation (which causes SLA violation) of the resources. Different from CloudScale and PRESS, Vault maintains the system performance by analysing the recent tenant activities and setting up a simple rule as discussed in section 4.4 with a specific performance threshold (i.e., consolidation effectiveness) in the absence of building a prediction model. Therefore, Vault does not need a series of preliminary preparations for building and training a prediction model, but it uses a simple and reliable scaling approach to maintain the system performance.

Chapter 7

Conclusion

In this thesis, we present Vault, an implemented system that offers parallel database-as-a-service with UTDD design principle. We provide insight into the real implementation of PDaaS, thereby revealing the system architecture of Vault on top of OpenStack. With UTDD, a deployment plan is devised to guide how OpenStack cloud resources are shared efficiently among the tenants. Besides, the query routing algorithm is designed to achieve load balancing and high availability in cloud. It accomplishes the goal by distributing the workload across multiple Vertica instances in the cluster. In order to maintain the query-latency SLA guarantee at runtime, Vault rely on the elastic scaling approach to meet the workload changes at any point of time. The outcome of experiments are in the context with different configurations for investigating Vault performance in real world practice. We believe that Vault is a promising system that serves tenants at a low cost with query latency SLA guarantee.

Bibliography

[1] Claudia Loebbecke, Joerg Bienert, and Ali Sunyaev. A parallel platform for big data analytics : A design science approach. *IJCSET*, 3:152–156, May 2013.

[2] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. Mad skills: new analysis practices for big data. *Proc. VLDB Endow.*, 2(2):1481–1492, August 2009.

[3] Forrester's Enterprise Architecture research group. Going big data? you need a cloud strategy. Technical report, Forrester Research, Inc, January 2017.

[4] Affreen Ara and Aftab Ara. Cloud for big data analytics trends. *IOSR Journal of Computer Engineering*, 18:1–6, September 2016.

[5] Rajkumar Buyya, James Broberg, and Andrzej Goscinski. *Cloud Computing Principles and Paradigms.* Wiley Publishing, Mar 2011.

[6] Shui Yu and Song Guo, editors. *Big Data Concepts, Theories, and Applications.* Springer, 2016.

[7] Rouven Krebs, Simon Spinner, Nadia Ahmed, and Samuel Kounev. Resource usage control in multi-tenant applications. In *CCGrid IEEE CS*, pages 122–131, 2014.

[8] Nader Mohamed and Jameela Al-Jaroodi. Real-time big data analytics: Applications and challenges. In *The 2014 International Conference on High Performance*, Bologna, Italy, 2014.

[9] Petrie Wong, Zhian He, and Eric Lo. Parallel analytics as a service. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 25–36, New York, NY, USA, 2013. ACM.

[10] Alok Shrivastwa and Sunil Sarat. *OpenStack Trove Essentials*. Packt Publ., March 2016.

[11] Amrith Kumar and Douglas Shelley. *OpenStack Trove*. Apress, 1 edition, 2015.

[12] Sherif Sakr and Mohamed Gaber. *Large Scale and Big Data: Processing and Management*. Auerbach Publications, Boston, MA, USA, 2014.

[13] K.Venkataramana and Prof.M.Padmavathamma. Multi-tenant data storage security in cloud using data partition encryption technique. *International Journal of Scientific Engineering Research*, 4, July 2013.

[14] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer-Verlag New York, 3 edition, 2011.

[15] Joseph D'Antoni and Scott Klein. *Provisioning SQL Databases*. Pearson Education, August 2017.

[16] Carlo Curino, Evan P. C. Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Sam Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: A database-as-a-service for the cloud. April 2011.

[17] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD*, SIGMOD '17, Chicago, IL, USA, May 2017. ACM.

[18] Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1077–1080, New York, NY, USA, January 2013. ACM.

[19] Vivek Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chaudhuri. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proc. VLDB Endow.*, 8(7):726–737, February 2015.

[20] Sudipto Das, Vivek R. Narasayya, Feng Li, and Manoj Syamala. Cpu sharing techniques for performance isolation in multi-tenant relational database-as-a-service. *Proc. VLDB Endow.*, 7(1):37–48, September 2013.

[21] Samarsinh Prakash Jadhav and Priya R. Deshpande. Load balancing in cloud computing. In *International Journal of Science and Research (IJSR)*, volume 3, 2014.

[22] Ramesh Prajapati, Dushyantsinh Rathod, and Samrat Khanna. Comparison of static and dynamic load balancing in grid computing. In *International Journal For Technological Research In Engineering*, volume 2, 2015.

[23] Foram F Kherani and Jignesh Vania. Load balancing in cloud computing. In *International Journal of Engineering Development and Resesarch (IJEDR)*, volume 2, 2014.

[24] Jyoti Vashistha and Anant Kumar Jayswal. Comparative study of load balancing algorithms. In *IOSR Journal of Engineering (IOSRJEN)*, volume 3, pages 45–50, 2013.

[25] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[26] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference*, Niagara Falls, Canada, October 2010. IEEE.