Parallel Analytics as a Service

WONG PETRIE KE FANG

M.Phil

The Hong Kong
Polytechnic University
2014

The Hong Kong Polytechnic University

Department of Computing

Parallel Analytics as a Service

WONG Petrie Ke Fang

A thesis submitted in partial fulfillment of the requirements

for the degree of

Master of Philosophy

December 2013

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.

. . . .

WONG Petrie Ke Fang

Abstract

Recently, massively parallel processing relational database systems (MPPDBs) have gained much momentum in the big data analytic market. With the advent of hosted cloud computing, this thesis envisions that the offering of MPPDB-as-a-Service (MPPDBaaS) will become attractive for companies having analytical tasks on only hundreds gigabytes to some ten terabytes of data because they can enjoy high-end parallel analytics at a cheap cost. This thesis presents *Thrifty*, a prototype implementation of MPPDB-as-a-service. The major research issue is how to achieve a lower total cost of ownership by consolidating thousands of MPPDB tenants on to a shared hardware infrastructure, with a performance SLA that guarantees the tenants can obtain the query results as if they are executing their queries on dedicated machines. Thrifty achieves the goal by using a *tenant-driven design* that includes (1) a *cluster design* that carefully arranges the nodes in the cluster into groups and creates an MPPDB for each group of nodes, (2) a *tenant placement* that assigns each tenant to several MPPDBs (for high availability service through replication), and (3) a *query routing* algorithm that routes a tenant's query to the proper MPPDB at run-time. Experiments show that in a MPPDBaaS with 5000 tenants, where each tenant requests 2 to 32 nodes MPPDB to query against 200GB to 3.2TB of data, Thrifty can serve all the tenants with a 99.9% performance SLA guarantee and a high availability replication factor of 3, using only 18.7% of the nodes requested by the tenants.

Acknowledgements

First of All, I would like to express my deepest sense of gratitude to my advisor, Dr. Eric Lo, for the guidance during my study. He is not only the one who gives me the chance to come to the academic research world, but also the one who gives me a lot of invaluable advices in different research projects. Without him, I would not have the chance to complete this work.

Next, I would like to thank Prof. Benjamin Kao for giving me so many advices on various research projects. All his advices in research direction and problem solving skill are useful for completing this work.

Dr. Man-Lung Yiu has given me invaluable advices on programming and algorithm design, for which I am extremely grateful.

I am grateful to Andy He for providing me with his opinion which open my mind. He also gave me a lot of fun when I was bored with my work.

The comments and critiques on this work and throughout my study that have been received from various other parties are also highly appreciated. Thanks to Dr. Wilfred Lin, Duncan Yung, Cliz Sun, Yu Li, Qiang Zhang, Yifeng Luo, Ziqiang Feng, Wenjian Xu, Jianguo Wang, Capital Li, Bo Tang, Tanya Aldynool, Jeppe Thomsen, Evan Feng, Ho Leung Li et al. for their comments and critiques on various parts of this work.

I thankfully remember Dr. Siu-Wai Lit, Red Hall Warden, for her love and prayers. Besides, all my colleges in Red hall, Carol Lee, Kiki Hui, Alice Yao, Sophie Huang, Wayne Wu, Erica Lam et al. gave me a memorable hall life.

I would like to thank Erica Zhang for her personal support and great patience at all times. My mother, sister and Rex Chan have given me their unequivocal support throughout, as always, for which my mere expression of thanks likewise does not suffice. I sincerely thanks to Rose Wong and Jun Chen for their selfless support, encouragement and advices to my personal development.

Contents

List of Figures

List of Tables

Chapter 1

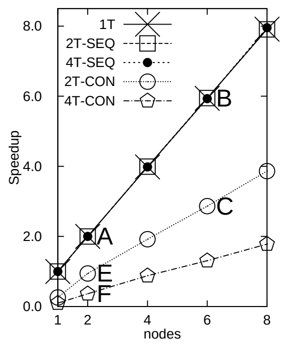
Introduction

1.1 Introduction

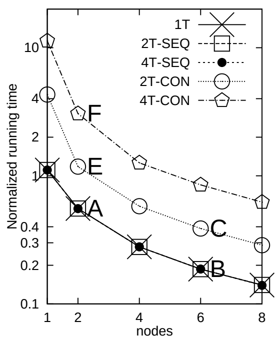
Recently, massively parallel processing relational database systems (MPPDBs) like Vertica, Microsoft SQL Server Parallel Database Warehouse, and Greenplum have gained much momentum in the big data analytic market. These MPPDBs feature SQL interface, fast data loading, easy scale-out, fast recovery, and short analytical query processing time. With the advent of hosted cloud computing, we envision that the offering of **MPPDB-as-a-Service (MPPDBaaS)** will become attractive for companies having analytical tasks on hundreds gigabytes to some ten terabytes of data. From the viewpoint of tenants who rent the service, they can enjoy parallel analytics on their "big" data, but the hardware cost, the operation cost (e.g., administration), and especially the software license (which costs about USD 15K per core or USD 50K per TB of data for a commercial MPPDB we know) they pay for a share of the service are likely to be much lower than running everything themselves. From the viewpoint of the service providers, they can achieve a lower total cost of ownership by consolidating the tenants on to a shared hardware infrastructure, so that the service can be run using far fewer machine nodes than the sum of the number of the machine nodes requested by the tenants. When sharing resources among tenants, it is challenging to ensure the service level agreements (SLA) for the tenants are met after consolidation. Ideally, a tenant that rents a 4-node MPPDB should have the feeling that the queries are really executed by a MPPDB that runs on four dedicated machine nodes. Therefore, a service provider that offers MPPDB-as-a-service should regard the *query latency before consolidation* as the performance SLA.

One approach to offer database-as-a-service (DaaS) is based on the use of virtual machines (VM) [25]. The VM approach offers hard isolation among tenants and thus performance isolation among tenants could be easily achieved. However, the VM approach incurs significant redundancy on the resources (e.g., multiple copies of the OS and DBMS, etc.) [8, 7]. Therefore, the recent trend for DaaS is to use the *shared-process multi-tenancy* [13] model (i.e., sharing the same database process among tenants). Relational Cloud [8, 7] and SQL Azure [4] are using this model so that a much higher consolidation effectiveness could be obtained. But so far they focus on tenants with *transactional workloads*.

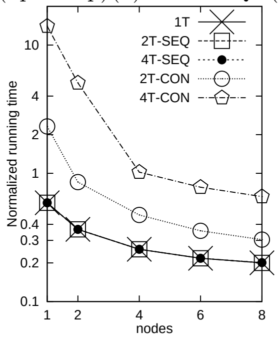
Let us see some interesting results when applying the shared-process multi-tenancy model to host tenants with *parallel analytical workloads*. Figures 1.1a and b show the query performance of TPC-H Q1 on a commercial MPPDB. We use this MPPDB to host multiple tenants using the same database process (the detailed experimental setup is given in Chapter 7). Each tenant holds a TPC-H



(a) TPC-H Q1 (Speed-up)



(b) TPC-H Q1 (Query Latency)



(c) TPC-H Q19 (Query Latency)

Figure 1.1. Query performance in a commercial MPPDB with multi-tenants. xT-SEQ means there are x tenants submitting queries sequentially xT-CON means there are x active tenants submitting queries concurrently.

scale factor 100 dataset. In Figure 1.1a, the line 1T shows the speedup of Q1 in a single-tenant setting. We can see that Q1 scales out linearly with the number of nodes.

Transactional (OLTP) workloads generally touch a small amount of data per transaction. So, OLTP-database-as-a-service using the shared-process model could still support a large number of concurrent query executions after consolidation [8, 7]. In contrast, analytical workloads are I/O-intensive, so concurrent query executions on the same database instance *easily increase the query latency* and violate the SLA. In Figure 1.1a, the lines 2T-CON and 4T-CON show the speedup of Q1 (with respect to 1-node) when there are respectively two tenants and four tenants sharing the same database instance, and the tenants submit Q1 together. We can see that Q1's performance are 2$\times$ (in 2-tenant setting) and 4$\times$ slower (in 4-tenant setting). That is, once the tenants submit queries together, none of their SLAs could be met. This is a challenge of using the shared-process multi-tenancy model to serve analytical workloads.

Yet, Figure 1.1a reveals a consolidation opportunity. In the figure, the line 2T-SEQ shows the speedup of Q1 (with respect to 1-node) in a 2-tenant setting where one tenant first submits an instance of Q1, and after that finishes, another tenant then submits another instance of Q1; and the average query latency of Q1 is calculated. Not surprisingly, as shared-process multi-tenancy incurs little overhead [8], when two Q1 instances are executed one after the other on a MPPDB with multiple tenants, their individual query latency is close to that when running a single Q1 instance on a MPPDB with a single tenant. This observation also holds at a higher degree of multi-tenancy with four tenants (see the line 4T-SEQ in Figure 1.1a). The above shows that even for analytical workloads, if the consolidated tenants are not active together, it is possible that the tenants' individual SLA could still be met.

The second consolidation opportunity can be observed by looking at Figure 1.1b, which shows the query latency of Q1. First, assume there are four tenants, each rents a 2-node MPPDB to process a TPC-H scale factor 100 dataset. The service provider basically requires a total of $4 \times 2 = 8$ separate nodes to host them. The SLA of TPC-H Q1 for each tenant is then A seconds, as illustrated in Figure 1.1b. What if the service provider uses a 6-node MPPDB to host all four tenants (i.e., every tenant partitions their data on to those six nodes)? Let us assume that only one out of the four tenants is *active* (e.g., submit a Q1 instance) and the rest of them are *inactive*, i.e., not submitting any query. That active tenant then can obtain the result of Q1 in B seconds — the SLA is met. When two out of the four tenants are active together and they concurrently submit an instance of Q1 to the 6-node MPPDB, each tenant can obtain the result of Q1 in C seconds — their SLAs are also met. The above shows that it is a consolidation opportunity that is more specific to parallel analytic workload.

This thesis presents the design, implementation, and experimental evaluation of *Thrifty*, a system that offers MPPDB-as-a-service. Thrifty employs shared-process multi-tenancy and exploits opportunities mentioned above to consolidate tenants on to a shared cluster. Low active tenant ratios are commonly found in real multi-tenancy environments. For example, in IBM's database-as-a-service, the active tenant ratio is only 10% [21]. Such a low active tenant ratio gives Thrifty many opportunities to meet the SLAs after consolidation. Nevertheless, there are still some challenges left. First, consider a production cluster of 6000 nodes, a total of 4000 tenants, and each tenant requests a 2-node MPPDB. In this case, shall Thrifty put all 4000 tenants onto a 6000-node MPPDB instance? Second, the above discussion has not yet covered the issue of *high availability*, which is important for database-as-a-service. Third, Thrifty has to avoid SLA violations like some cases above where many tenants become active together. Fourth, notice that TPC-H Q1 is a *linear scale-out* query under our experimental setting. Thrifty also has to consider other cases such as concurrent execution of *non-linear scale-out queries*. Figure 1.1c shows the performance of TPC-H Q19 in the same experimental environment. TPC-H Q19 does not scale out linearly in our setting. For those queries, the second consolidation opportunity mentioned above cannot be easily applied.

In a nutshell, Thrifty aims to offer MPPDBaaS using few machines with the following requirements:

R1) Supporting thousands of tenants and machine nodes.

R2) Offering high availability services.

R3) Offering query-latency based performance SLA guarantees.

R4) Each tenant requests a multi-node MPPDB that serves multiple users; some user queries may linear scale-out but some may not.

R5) Tenants' query templates may be known or unknown beforehand. For report generating applications, the query templates could be found in the applications' stored procedures. For interactive analysis, however, a data analyst may craft and submit an ad-hoc query at any time.

R1 and R2 are two general cloud computing requirements. R3 is a DaaS requirement specific to analytical workloads.[1] R4 is specific to MPPDBaaS and R5 is necessary if we want to offer a real MPPDBaaS. These requirements together make the design and implementation of Thrifty very challenging. For example, sharing the database process among tenants (to enjoy higher consolidation ratio) makes R3 challenging because there is no more hard isolation. While performance prediction techniques for concurrent analytical workload [9, 1, 22, 16, 2] may help there on the surface, these techniques require the knowledge of the query sets as input, which contradicts with R5 above.

This thesis makes three principal contributions for this problem:

[1]DaaS supporting OLTP workloads defines their SLA based on the throughput before-and-after consolidation [7].

1. Tenant-Driven Design (TDD) (Chapter 4) — the core framework to implement MPPDB-as-a-Service. TDD includes three parts: (a) [Cluster Design] arrange the machine nodes in the cluster into *groups* and create an MPPDB for each group of nodes, (b) [Tenant Placement] assign each tenant to several MPPDBs (for replication), and (c) [Query Routing] route a tenant's query to a MPPDB containing that tenant's data at run-time. We can show that the use of TDD can tackle requirements R2 to R5 in a row. Furthermore, based on TDD, we can show that solving R1 could be treated as solving a *new variant of vector bin packing optimization problem.* We then present heuristics solutions that solve that problem effectively (Chapter 5).

2. Thrifty — a prototype implementation of MPPDB-as-a-service based on TDD. The architecture design of Thrifty (Chapter 3) as well as system issues like elastic scaling (Chapter 5.1) and system tuning (Chapter 6) are presented.

3. An experimental testbed to evaluate the consolidation effectiveness and the performance guarantees of any MPPDBaaS. Experiments show that in a MPPDBaaS with 5000 tenants, where each tenant requests 2 to 32 nodes MPPDB to query against 200GB to 3.2TB of data, Thrifty can serve all the tenants with a 99.9% performance SLA guarantee and a high availability replication factor of 3, using only 18.7% of the nodes requested by the tenants (Chapter 7).

After presenting the above, we discuss related work in Chapter 2 and conclude this work and outline some future work in Chapter 8.

Chapter 2

Related Work

Thrifty [26] is the first work to discuss MPPDB-as-a-service and provides a real implementation to support the stringent query-latency SLA for parallel analytical workloads.

2.1 DaaS

There are several approaches to provide database-as-a-service [13]: (1) The Virtual Machine approach, (2) the Shared Table Approach, and (3) the Shared Process Approach.

(1) The VM approach is to pack each individual tenant's database instance into a virtual machine (VM) and multiple VMs on a single physical machine [25]. The VM approach achieves hard isolation. Therefore, the performance SLA of individual tenants could be easily maintained. However, this approach incurs significant redundancy among the tenants (e.g., multiple copies of the OS and

DBMS) and thus [7] states that it is not cost-effective for DaaS.

(2) The Shared Table approach is to share a single database instance and *the same set of tables among all tenants*. This approach requires the tenants' schemas are mostly identical or similar, which is common in software-as-a-service (e.g., Salesforce.com) with database as the backend. The Shared Table approach can largely reduce the redundancy and thus thousands of small (data size) and mostly inactive tenants can be packed onto a single server [3, 23, 12]. This approach is orthogonal to Thrifty, as the tenants are autonomous and tenants' schemas are typically different.

(3) The Shared Process approach is to share a single database instance among the tenants but each tenant can own its private set of tables. This approach achieves a much higher consolidation effectiveness than the VM approach and and it supports autonomous tenants with arbitrary schemas. MIT's Relational Cloud [8] is based on this approach. It includes a consolidation engine, Kairos [7], to consolidate many *centralized OLTP databases* on to a shared hardware infrastructure. The SLA of consolidating OLTP workloads in Kairos is the database throughput before-and-after consolidation.

Thrifty uses the Shared Process approach in order to attain a higher consolidation effectiveness. Different from Relational Cloud [8], the tenants (a) request multiple-node MPPDBs and (b) issue analytical workloads. The former requires a skillful cluster design that arranges the nodes in the cluster into groups of MPPDBs and a corresponding tenant placement scheme that assigns a tenant to R different MPPDBs. That is not an issue in Relational Cloud for which tenants hosts only a single centralized database. The latter makes the consolidation concerns and methodology of Thrifty different from Relational Cloud: as mentioned, OLTP workloads generally touch a small amount of data per transaction. So the same database instance could still support a large number of concurrent query executions from different tenants after consolidation. Therefore, the consolidation concerns of Relational Cloud are (i) how to estimate the resource (e.g., CPU, RAM) of each OLTP-workload and (ii) how to compute the aggregated resource consumptions of multiple workloads. In contrast, analytical workloads are mostly I/O intensive, so any concurrent query execution on the same database instance would easily increase the query latency and cause SLA violation (see Points E and F in Figure 1.1). Therefore, the consolidation concern is how to avoid concurrent query execution when consolidating the tenants.

2.2 Parallel Databases

In parallel databases, performance speedup is brought by partitioning the data across multiple machine nodes to maximize intra-query parallelism. Recent research related to parallel database mostly focuses on finding a good data partitioning strategy to minimize the overhead of shipping data across multiple nodes [17, 6]. Temporal access skew was also used by a recent NewSQL system to aid the data partitioning design [19]. Thrifty leverages temporal access skew, i.e., low active tenant ratio, to aid the cluster design to deal with the SLA guarantee problem caused by concurrent executions of analytical queries in a shared database process.

2.3 Optimization problems in cloud computing

VM-based tenant placement is a hot topic in cloud computing [10, 15]. Yet, to the best of my knowledge, these works have not considered any MPPDBaaS-specific requirements like query-latency performance SLA (R3) or requesting multiple nodes by a single tenant (R4). In Thrifty, after using TDD, it forms a Largest Item Vector Bin Packing Problem with Fuzzy Capacity (LIVBPwFC), which is a variant of the classic vector bin packing (VBP) problem. The VBP problem has been used to model many forms of VM placement problem recently [15]: a VM is represented by a *vector of resource demands* $\langle r_1, r_2, \ldots, r_d \rangle$ (each r_i denotes the demand of a particular resource, say, RAM) and a machine is represented by a *vector of resource capacity* $\langle c_1, c_2, \ldots, c_d \rangle$ (each c_i denotes the capacity of a particular resource, say, RAM, of a node). As LIVBPwFC is newer and more general than VBP, a new heuristics is devised for it in this thesis.

2.4 Elastic resource scaling in cloud computing

Finally, substantial research in cloud computing has looked at the problem of elastic resource scaling (e.g., [24]). Thrifty follows those reactive approaches with the unique challenges that the scaling of MPPDB is a heavyweight operation, comparing with the elastic scaling of hardware resources like the memory allocated to a virtual machine.

Chapter 3

System Overview

Thrifty adopts a pricing model that charges a tenant based on the number of requested nodes (the degree of parallelism) and its active usage. It begins with a set of independent MPPDB tenants running parallel analytical workloads, where each tenant runs a MPPDB on a disjoint set of machine nodes. Figure 3.1 shows the architecture of Thrifty. The components of Thrifty are:

(a) *Tenant Activity Monitor*: The Tenant Activity Monitor automatically collects the query logs of the deployed MPPDBs, derives the tenant activities, and summarizes the query characteristics of individual tenants. For example, it continuously monitors the active tenant ratio of all tenants in the past 30 days. These information is passed to the *Deployment Advisor* for further processing, and is available to the system administrator for advanced system tuning.

(b) *Deployment Advisor*: The Deployment Advisor takes as inputs the tenant activity statistics, the individual tenant information (e.g., number of nodes requested), a replication factor R (specified by a system administrator for high availability), and a performance SLA guarantee P (also specified by the system administrator which ensures that, after consolidation, the tenants can still meet their SLA for a P%, e.g., 99.9%, of time). It returns as output a *deployment plan*. The deployment plan consists of two parts: *cluster design* and *tenant placement*. The cluster design details how the machine nodes in the cluster are arranged into groups. Each group of nodes then runs a single MPPDB instance. The tenant placement specifies which MPPDB instances a tenant should be deployed on; and a tenant is deployed on R MPPDB instances. At run-time, the Deployment Advisor may fine-tune the deployment plan according to the live-updated tenant activities supplied by the Tenant Activity Monitor. Currently, Thrifty assumes all nodes in the cluster are identical in configurations (e.g., CPU, RAM).

(c) *Deployment Master*: The Deployment Master follows the deployment plan devised by the Deployment Advisor to start the MPPDB instances and deploy the tenants onto them. It also switches off/hibernates nodes that are not listed in the deployment plan. The deployment is supposed to be static for days. A (re)-consolidation process is expected to be executed periodically, because it is expected that there are new tenants register with and existing tenants de-register with the service.

(d) *Query Router*: The Query Router accepts queries from tenants and routes the queries to the proper MPPDB instance according to the tenant placement.

The crux of Thrifty is to devise a good deployment plan to serve thousands of tenants using fewer machine nodes. Thrifty targets on tenants with temporal

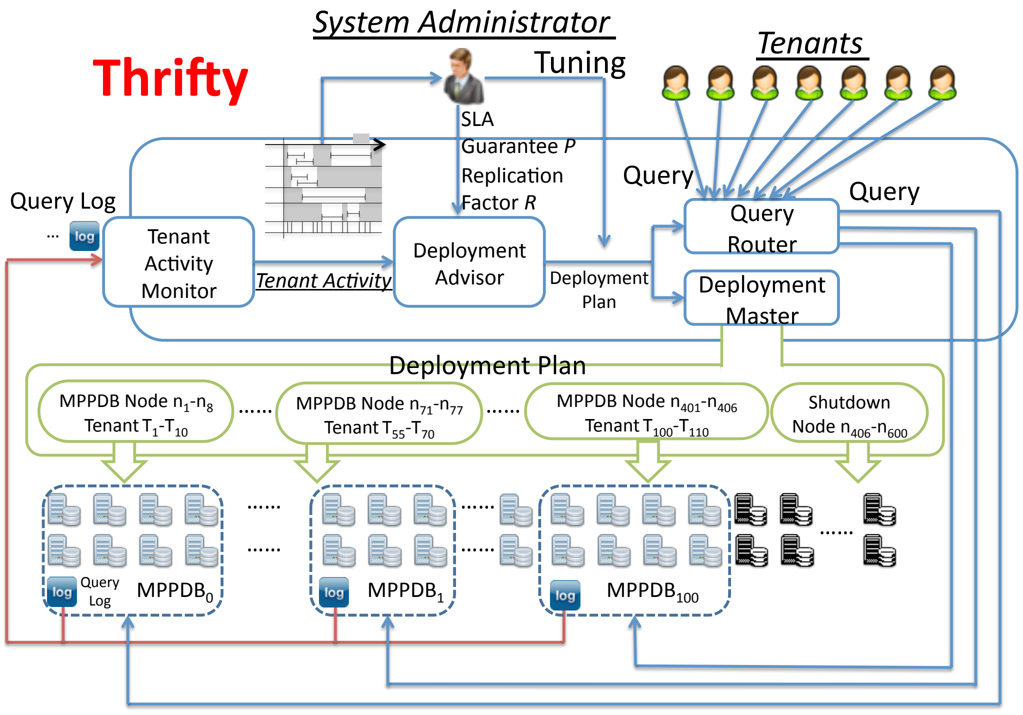


Figure 3.1. The architecture of the Thrifty system.

skew [19] analytical workloads, e.g, tenants from different time zones and mostly submit queries within a certain period (e.g., 9am to 5pm) but not in a 24×7 fashion. That type of temporal skew workload is not uncommon in today's data analytical market [5]. The target tenants are supposed to have up to only some tens terabytes of data but hope to enjoy the parallel speedup offered by the expensive MPPDB products. These are what enable consolidation. Tenants that are always active and/or with more than terabytes of data could be detected by Thrifty and they will be excluded from consolidation.[1]

[1]Those tenants offer little room for consolidation, and they are served by dedicated nodes under another service plan.

Chapter 4

Tenant-Driven Design (TDD)

The principle of Tenant-Driven Design (TDD) is to exploit the low active tenant ratio found in real multi-tenant cloud environments to design a deployment plan that lets each active tenant exclusively use an MPPDB that just fits (or, over-fits) its parallelism requirement. By using TDD, it can be shown that requirements R2 to R5 (Chapter 1.1) are all tackled in a row.

The whole Tenant-Driven Design consists of three parts: (1) Cluster Design (Chapter 4.1), (2) Tenant Placement (Chapter 4.2), and (3) Query Routing (Chapter 4.3).

4.1 Cluster Design

The cluster design of TDD is to divide a set of machine nodes into A groups (we will discuss how to set the value of A later). Then, each group of nodes form a separate MPPDB. Let T be the number of tenants, n_i be the number of

Tenant	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
n_i	6	6	5	5	5	4	4	3	2	2

(a) Tenant Characteristics (42 nodes requested)

Group	G_0	G_1	G_2
MPPDB	MPPDB$_0$	MPPDB$_1$	MPPDB$_2$
Parallelism	6-node	6-node	6-node

(b) Cluster Design (18 nodes)

MPPDB	MPPDB$_0$	MPPDB$_1$	MPPDB$_2$
Hosting	$T_1 - T_{10}$	$T_1 - T_{10}$	$T_1 - T_{10}$

(c) Tenant Placement

Figure 4.1. A Toy Example

nodes requested by tenant T_i, and N be the total number of nodes requested by all T tenants (i.e., $N = \sum_1^T n_i$). Without loss of generality, we assumes $n_i \leq n_j$ if $i > j$. Then, the tenant-driven design assigns n_1 nodes to groups G_1, G_2, ..., G_{A-1}. Group G_0 is a special group. The tenant-driven design assigns a variable of U nodes to G_0, where $n_1 \leq U \leq (N - (A-1)n_1)$. In this thesis, we call the MPPDB created from group G_0 as the "tuning MPPDB", which is used for system tuning. We will discuss more about the choice of U in Chapter 6 and now we assumes $U = n_1$.

As a toy example, assume that there are $T = 10$ tenants and the number of nodes requested by each tenant is listed in Figure 4.1a. In this toy example, the service provider requires a total of $N = 42$ nodes to host these tenants before consolidation. So, assume that $A = 3$, there are three groups and TDD results in a 18-node cluster design as listed in Figure 4.1b.

4.2 Tenant Placement

The tenant placement of TDD is very simple. Specifically, each MPPDB is assigned to host all tenants. Figure 4.1c illustrates the tenant placement for the example. From the above discussion, it observes the Property 1.

PROPERTY 1 *The tenant-driven design enforces a replication factor of A for each tenant.*

Replication is a standard way to ensure high availability and load balancing. Usually, the replication factor is around two to four[20].

4.3 Query Routing

Algorithm 1 presents the query routing algorithm based on the tenant-driven design. The idea is to "route an active tenant" (compare with the concept of "route a query") to one MPPDB and let that MPPDB exclusively process all that tenant's (concurrent) queries until that tenant becomes inactive. Here, we use a strong notion of *inactive* — as long as a tenant does not have any queries being executed by any MPPDB, that tenant is inactive at that moment.

We explain the query routing by walking through an example of tenant activities listed in Figure 4.2. In the example, the maximum number of concurrent active tenants is three. In the beginning, tenant T_4 becomes active and submits a query Q_1. As all MPPDBs in the cluster are initially free (not serving any queries), Q_1 gets routed to MPPDB$_0$ (Line 5). Next, tenant T_2 becomes active and submits a query Q_2. As MPPDB$_0$ is currently busy (serving T_4's Q_1),

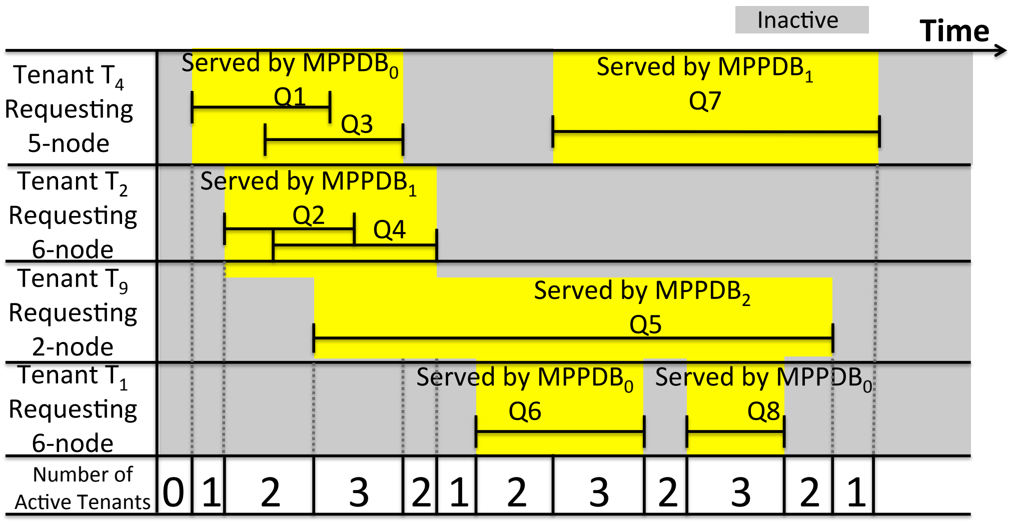


Figure 4.2. Tenant Activities and Query Routing

Q_2 is routed to MPPDB$_1$, which is free (Line 8). Next, tenant T_4 has a query Q_3 submitted while Q_1 is still running on MPPDB$_0$. So, Q_3 is also routed to MPPDB$_0$ (Line 2). Continuing the example, Q_4 is the next query arriving to the system. As Q_4 is submitted by T_2 and MPPDB$_1$ is still serving a query of tenant T_2, Q_4 is routed to MPPDB$_1$ as well (Line 2). Tenant T_9 becomes active next and submits a query Q_5. In this case, Q_5 is routed to MPPDB$_2$, which is free (Line 8). Next, tenant T_1 becomes active and submits query Q_6, since T_4 has just finished Q_1 and Q_3, Q_6 is routed to MPPDB$_0$ (Line 5). Q_7 is the next query submitted by tenant T_4 after Q_1 and Q_3. Since Q_1 and Q_3 have already finished, Q_7 of T_4 is not necessary be routed to MPPDB$_0$. It is thus routed to MPPDB$_1$ because MPPDB$_0$ is busy serving tenant T_1. Q_8 of tenant T_1 is the last query in this example. Although Q_8 is submitted almost right after Q_6 (e.g., with a short “think-time” during a user’s interactive analysis session), tenant T_1 is still regarded as inactive for a short while. Therefore, it is not necessary for Q_8 to follow Q_6 to go to MPPDB$_0$. In this example, however, it is still routed

Algorithm 1: TDD Query Routing Algorithm

```
Input: Tenant $T_i$, Query $Q$
1  if $T_i$ has queries running on $MPPDB_x$ then
2      route $Q$ to $MPPDB_x$;
3  else
4      if $MPPDB_0$ currently is free then
5          route $Q$ to $MPPDB_0$;
6      else
7          if there is any free MPPDB then
8              route $Q$ to a free $MPPDB_j$;
9          else
10             route Q to $MPPDB_0$ for concurrent processing;
```

to MPPDB_0 because all other MPPDBs are busy.

4.4 SLA Guarantee and Load Balancing

Guarantee 1 *No matter a tenant's queries are* linear-scale-out*,* non-linear-scale-out*, submitted* sequentially and possibly in an ad-hoc manner *(interactive analysis), or* submitted in a batch for concurrent execution (report generation) at any multi-programming-level (MPL)*. Tenant-Driven Design ensures the SLAs of a maximum of A active tenants are met.*

Although simple, TDD can obtain a strong performance SLA guarantee under a general and practical MPPDB-as-a-service setting. In essence, TDD has tackled requirements R2 to R5 in a row. In the example above, all tenants' SLA are met because their queries are *served by dedicated MPPDBs with exact (e.g.,* Q_2, Q_4, Q_6, Q_8*) or higher degree of parallelism (e.g.,* Q_1, Q_3, Q_5, Q_7*).*

TDD achieves load balancing among tenants implicitly. That is because

TDD uses a strong notion of inactive and it applies replication on the tenant data, therefore the tenants can be quickly served by an available MPPDB. Thrifty remarks that load balancing *within* a tenant is not TDD's but the tenant's own issue. The pricing model of Thrifty is to charge a tenant based on the number of nodes it requested. Therefore, if a tenant rents only a 2-node MPPDB instance to serve many concurrent users (e.g., MPL=100), the possibly slow down of the tenant's performance due to the high load is brought by the tenant's own node-choice — TDD is just offering what the tenant should get.

Node failure is handled directly by the MPPDB. All Major MPPDB products can still stay online even with (some) node failure. Thrifty will replace a failed node by starting a new node upon receiving node failure notification from the underlying MPPDB. Node failure, or having more than A active tenants at run-time may cause performance delay and thus SLA violations. We will address this in the next two sections.

Chapter 5

Serving thousands of tenants

With the use of TDD, Thrifty is able to handle a general type of MPPDBaaS tenants (R4 and R5) while offering high availability services (R2) with performance SLA guarantees (R3). However, the discussion so far has only considered a small number of tenants and the total cluster size (i.e., the total number of nodes involved) is small. If there are a lot of tenants, say $T = 5000$, and if the (average) active tenant ratio is 10%, the average number of active tenants at a certain point may reach 500 and the maximum number of active tenants could be even larger. According to Guarantee 1, in order to ensure all that 500+ active tenants can meet their SLAs together, it needs to set $A > 500$. That would enforce each tenant's data get replicated more than 500 times, according to Property 1. Thrifty approaches this issue (R1) by grouping the tenants into *tenant-groups* so that there are only some tens tenants in each tenant-group.

There are two considerations when grouping the tenants. First, we want to group the tenants in a way that can minimize the total number of nodes used after consolidation. Second, we want to ensure the active tenant ratio in each tenant-group is low enough to meet the replication factor R. For example, if Thrifty puts 30 tenants that are always active together into the same tenant-group, Thrifty still needs to set $A = 30$ for that tenant-group in order to meet all tenant's SLA.

Finding an optimal tenant-group formation that can address the two considerations above, unfortunately, is $\mathcal{NP}$-hard. To explain, consider a simple scenario that $R = 3$ and six tenants, whose activities are divided into sequences of d fix-width time epochs like in Figure 5.1. Based on that representation, finding the optimal tenant-group formation is equivalent to a **Largest Item Vector Bin Packing Problem with Fuzzy Capacity** (LIVBPwFC), which is a new variant of the classic vector bin packing problem. Specifically, a tenant (item) T_i is charactered by a tuple $(\vec{A}_i, n_i)$, where n_i denotes the number of nodes requested by T_i, and $\vec{A}_i = \langle a_1, a_2, \cdots, a_d \rangle$ depicts the activity of T_i: $a_k = 1$ if T_i is active in the k-th epoch or $a_k = 0$ otherwise. For example, in Figure 5.1, there are 10 epochs. Tenant T_1's activity $\vec{A}_1$ is represented as $\langle 1, 1, 1, 1, 1, 1, 0, 0, 0, 0 \rangle$, meaning that it was active from t_1 to t_6 and inactive from t_7 to t_{10}. Each tenant group (bin) TG_j is charactered by a tuple $(\vec{B}_j, P)$, where $\vec{B}_j$ is a d-dimensional vector $\vec{B}_j = \langle R, R, \cdots, R \rangle$. $\vec{B}_j$ and P together characterize the *fuzzy capacity constraint* of a bin such that it is not full as long as the set of items S that are packed in this bin have more than or equal to $P\%$ of epochs having less than R active tenants. In other words, a set S of tenants (items) fits into a TG_j (bin) if $COUNT^{\leq R}(\sum_{T_i \in S} \vec{A}_i)/d \geq P\%$, where $COUNT^{\leq R}(\vec{V}_i)$ is a function that counts the number of dimensions in a vector $\vec{V}_i$ whose values are less than or equal to R. For example, consider $S = \{T_1, T_4, T_5, T_6\}$ in Figure 5.1. $\sum_{T_i \in S} \vec{A}_i$

results in a vector $\langle 2,2,2,2,4,3,2,1,2,1\rangle$ and $COUNT^{\leq R}(\sum_{T_i \in S} \vec{A_i})$ returns 9 if $R = 3$.

Concerning the optimization goal, recall that for T tenants, the tenant-driven design uses at least $A \cdot n_1$ nodes. That is, the total number of nodes used is dominated by the largest tenant that requests the most number of nodes. For the example in Figure 4.1, T_1 is the largest tenant and it requests six nodes, so a total of $A \cdot n_1 = 3 * 6 = 18$ nodes is required. Therefore, the optimization goal is to minimize the total of number of nodes requested by the *largest item* (tenant) in each created bin.

The classic vector bin packing problem is a special case of the LIVBP-wFC problem in which only $\vec{A_i}$ and $\vec{B_j}$ are considered (i.e., n_i are ignored and $P = 100\%$). The classic vector bin packing problem is $\mathcal{NP}$-hard and recent work [18] states that First-Fit-Decreasing (FFD) is a practical heuristic to get approximate solutions. FFD suggests to sort all items according to a scalar value and inserts the items into a bin according to that order. An item is inserted into a new bin if the current bin is full. In order to get a single scalar value for a d-dimensional item, one heuristic is to take the product of the dimension values. However, FFD was not especially designed for the LIVBPwFC problem and it did not take into account the fuzzy capacity constraint and the largest item. Another direction is to phrase the problem as a mixed integer non-linear optimization problem (MINLP). Appendix 9.1 details such a formulation. However, the formulation contains non-linear constraints and many local minima. Therefore, only general-purpose global optimization algorithms/solvers (e.g., DIRECT [14]) could be used. Unfortunately, these general-purpose global optimization algorithms/solvers run extremely slow for more than 20 variables.

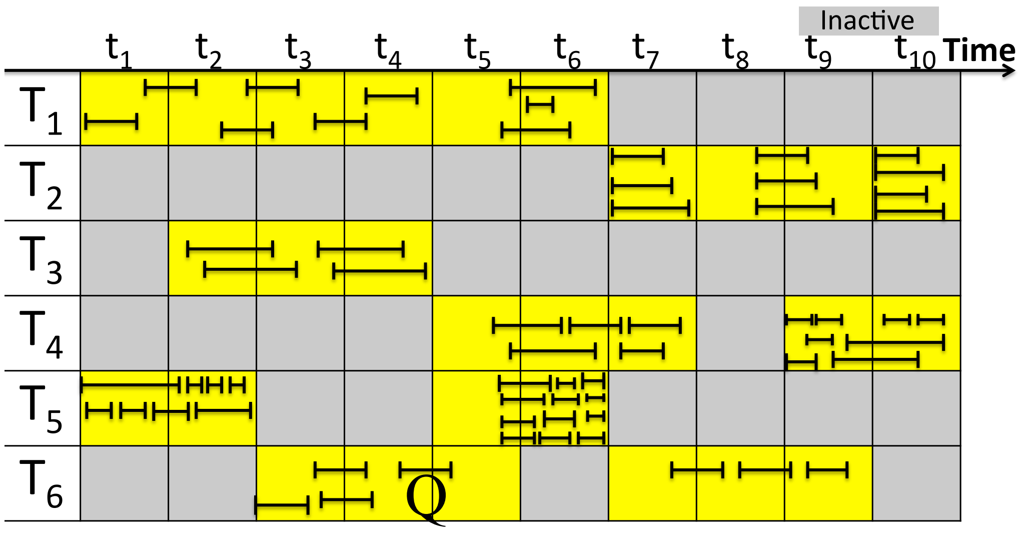


Figure 5.1. Tenant Activities with Fixed-Width Time Epochs

Figure 5.2. Example Tenant Distribution

Num. of Tenants	900	800	700	600	500	400	300	200	100
Parallelism	2	3	4	5	6	7	8	9	10

As recent work [18] suggests using heuristics to solve vector bin packing style problems is more practical, we have developed our own heuristic algorithm that exploits properties of TDD. The algorithm works by solving the two considerations separately. Specifically, it first looks at *what kind of grouping can generate a cluster design that can save more nodes?* Recall that the total number of nodes used by a cluster design is dictated by the largest tenant that requests the most number of nodes. In Figure 4.1, T_1 is the largest tenant and it requests six nodes, so that cluster design saves $42 - 3 * 6 = 24$ nodes. If it groups ten tenants of equal size, each requests six nodes, the resulting cluster design would still be the same, but that could save a total of $6 \times 10 - 18 = 42$ nodes. Therefore, the first intuition is that *it should put tenants of the same size (i.e., requesting the same number of nodes) into the same tenant-group.*

$TG_1 = \{T_3\}$	1-active	2-active
$+ T_1$?	30% → 30%	0% → 30%
$+ T_2$?	30% → 70%	0% → 0% *
$+ T_4$?	30% → 80%	0% → 0%
$+ T_5$?	30% → 50%	0% → 10%
$+ T_6$?	30% → 50%	0% → 20%

(a) T_2 is chosen

$\downarrow$

$TG_1 = \{T_3, T_2\}$	1-active	2-active	3-active
$+ T_1$?	70% → 70%	0% → 30%	0% → 0%
$+ T_4$?	70% → 60%	0% → 30%	0% → 0%
$+ T_5$?	70% → 90%	0% → 10%	0% → 0% *
$+ T_6$?	70% → 30%	0% → 50%	0% → 0%

(b) T_5 is chosen

$\downarrow$

$TG_1 = \{T_3, T_2, T_5\}$	1-active	2-active	3-active
$+ T_1$?	90% → 40%	10% → 50%	0% → 10%
$+ T_4$?	90% → 40%	10% → 60%	0% → 0% *
$+ T_6$?	90% → 30%	10% → 70%	0% → 0%

(c) T_4 is chosen

$\downarrow$

$TG_1 = \{T_2 - T_5\}$	1-active	2-active	3-active	4-active
$+ T_1$?	40% → 10% †	60% → 60%	0% → 30%	0% → 0%
$+ T_6$?	40% → 10%	60% → 60%	0% → 30%	0% → 0% *

(d) All ties; T_6 is chosen

$\downarrow$

$TG_1 = \{T_2 - T_6\}$	1-active	2-active	3-active	4-active
$+ T_1$?	10% → 0%	60% → 30%	30% → 60%	0% → 10%

(e)

TTP (for $R \leq 3$) before adding T_1: 10% + 60% + 30% = 100%

TTP (for $R \leq 3$) if T_1 is added: 0% + 30% + 60% = 90%

†: Explanation: With T_2–T_5 only, epochs t_1, t_3, t_4, and t_8 have 1 active tenant (40%).
With T_1 added, epoch t_8 has 1 active tenant (10%).

Figure 5.3. Second Step of Tenant-Grouping (* means that tenant is chosen in that iteration)

Next, we look at the second consideration, i.e., *how to ensure a low active tenant ratio within each tenant-group?* A natural intuition is that *it should analyze the tenants' activities, and put a tenant into a tenant-group that (a) minimizes the increase in the maximum number of active tenants of that tenant-group and (b) ensures that fuzzy capacity constraint of each resulting tenant-group is not violated.*

Based on the intuitions above, we have developed a two-step tenant grouping heuristic algorithm. In the first step, it puts all tenants with homogenous node-size into the same tenant-group (we call those *initial-groups*). Consider the tenant distribution in Figure 5.2, there would be nine initial-groups (e.g., all 300 tenants that request 8-node MPPDBs are in the same initial-group).

In the second step, it further splits the tenant in each initial group into tenant-groups. Specifically, for all tenants in the same initial group, it first inserts the least active tenant into a tenant-group. Next, it picks the tenant T_{best} that minimizes the increase in time percentage (measured in terms of the number of time epochs) of the maximum number of active tenants. Take the tenant activities in Figure 5.1 as an example. Assume tenant T_3 is put as the first member of the first tenant-group TG_1. So, when putting T_1 into TG_1, the total time percentage that has two active tenants, is increased from 0% to 30% (i.e., there are three epochs, t_2, t_3, t_4, out of ten, have two active tenants). If putting T_2 into TG_1, the total time percentage of having two active tenants does not increase. Figure 5.3a shows the potential time percentage changes of different numbers of active tenants when adding a tenant into the tenant-group TG_1 (with T_3 there). In the example, both T_2 and T_4 can keep the maximum number of active tenant to be one if any one of them is added to TG_1. So, we break the tie by examining the increase of the total time percentage time of having one active tenants — T_2 is then chosen and added to TG_1. Figure 5.3b shows the potential time percentage changes of different number of active tenants if adding either T_1, T_4, T_5, or T_6 to the tenant-group TG_1, which now contains T_3 and T_2. T_5 is chosen because it incurs the least increase in the total time percentage of the maximum number of active tenants. The process continues by adding T_4 (Figure 5.3c) and T_6 to TG_1 (Figure 5.3d), one-by-one. Note that after TG_1 has five tenants T_2, T_3, T_4, T_5, and T_6, the maximum number of active tenants is only three (see Figure 5.1).

Let TTP be the total time percentage that the number of active tenants is smaller than or equal to R in a tenant-group, and recall that P be a performance SLA guarantee that is specified by the system administrator, who wants to ensure that for a P% of time the tenants can meet their SLA. So, the adding of a tenant to a tenant-group stops only when that would result in $TTP < P$. In this example, assume the system administrator specifies $R = 3$ and she wants to ensure that for a 99.9% of time the tenants can meet their SLA. From Figure 5.3e, it shows that if T_1 is added into the tenant-group, the total time percentage that the number of active tenants is smaller than or equal to $R = 3$ will drop from 100% to 90%. That in contrast means there is 10% of time that there would be more than three active tenants. So, if $A = 3$, Thrifty may violate those tenants' SLA guarantee at those epochs. Therefore, T_1 would not add to tenant-group TG_1 and the tenant-grouping process starts a new iteration by creating a new tenant-group TG_2. The two-step tenant grouping algorithm is summarized in Algorithm 2. The first step takes $\mathcal{O}(T)$ time to form initial groups. For each initial group of size g_i, it searches from the g_i tenants the tenant T_{best} with the least increase in TTP and add it to a tenant group TG_j. The process repeats for the remaining $g_i - 1$ tenants, $g_i - 2$ tenants, so on and so forth and thus it requires $\mathcal{O}(g_i^2)$ searches for each initial group. Nevertheless, the algorithm running time is not a major issue because tenant-grouping is an offline process, triggered only occasionally, the number of tenants involved is usually in the order of thousands, and most importantly this experiments show that it yields deployment plans that save more than 3.6% to 11.1% nodes than standard heuristic. After tenant grouping, the resulting tenant-groups follow the tenant-driven design to generate the cluster design and tenant placement, using $A = R$.

Algorithm 2: 2-Step Tenant Grouping Algorithm

```
   Input: Tenants $\mathcal{T}$, Replication Factor $R$, SLA Guarantee $P\%$
   // First Step
1  Put tenants requesting the same number of nodes into the same initial groups
   // Second Step
2  for each initial groups do
3      $i = 1$;
4      Create tenant group $TG_i$;
5      Identify the tenant $T_{best} \in \mathcal{T}$ that will minimize the increase of time
       percentage of the maximum number of active tenants in $TG_i$;
6      if adding $T_{best}$ to $TG_i$, and $TG_i$'s TTP is still $\geq P\%$ then
7          Remove $T_{best}$ from $\mathcal{T}$ and add $T_{best}$ to $TG_i$;
8          Goto Line 5;                    ▷Try to add another tenant into this group
9      else
10         $i$++;
11         Goto Line 4;                    ▷ Create another tenant-group
```

The epoch size has an influence on the grouping quality. For example, Query Q of T_6 in Figure 5.1 actually only spans a very small portion of epoch t_5. If the epoch size is halved, T_6 can be added to the tenant group TG_1 as well. Empirically, we found that epoch sizes of 10 seconds to 30 seconds could achieve the best consolidation effectiveness.

Before we go to the next section, it remarks that the two-step tenant-grouping method relies on the assumption that "the tenant history repeats itself", which has been proven in realistic DaaS environments [8, 21]. So, when the TTP of a tenant-group at $R = 3$ is 99.9%, it has a high confidence that the number of active tenants is less than or equal to three at run-time. Of course, it is still possible that the number of active tenants exceeds three at run-time (e.g., 0.08% of time having four active tenants and 0.04% of time having five active tenants, which results in a 0.12% of time having more than three active tenants). Thrifty uses a reactive approach that elastically adds more nodes for that tenant group to handle those situations (will discuss this in the next section). This approach is pragmatic and makes good use of the elastic nature of cloud computing.

5.1 Lightweight Elastic Scaling

At run-time, tenants' activities may deviate from the history. Consider a tenant-group having a TTP of 99.95% during tenant-grouping. At run-time, some tenants in that group may become more active than what the past tenant activities have indicated, and that will result in a lower run-time TTP (RT-TTP). When the RT-TTP drops to, say 98%, that implies there is 2% of time the tenants are not exclusively served by dedicated MPPDBs and some of their queries may not meet the SLA. Thrifty handles this issue using a standard elastic scaling approach as in other cloud systems. The Tenant Activity Monitor monitors the RT-TTP of each tenant-group using a time window (e.g., 24-hour). When it detects the RT-TTP of a tenant-group of the past 24 hours has just dropped below the performance SLA guarantee P, Thrifty will call the Deployment Advisor to take action.

Tenant / Data Size	Node Starting & MPPDB Initialization	Bulk Loading
2-node / 200GB	462s	10172s
4-node / 400GB	850s	20302s
6-node / 600GB	1248s	30121s
8-node / 800GB	1504s	40853s
10-node / 1TB	1779s	50446s

Table 5.1. Starting and Bulk Loading a MPPDB

One possible action is to elastically scale up A by one, i.e., adding one more MPPDB. In MPPDBaaS, however, scaling up A is not as simple and efficient as scaling resources (e.g., RAM) in VM-based cloud environment. Specifically, it takes hours to start a new MPPDB and bulk load all tenants' data before it is ready to use. Table 5.1 shows the time of starting up a MPPDB and bulk loading tenant data into a MPPDB in this environment. It shows that the data loading time dominates the times of starting the machines and creating the MPPDB instances, although it has already achieved a reasonable loading rate (about 1.2GB/min). Take starting a 10-node MPPDB that contains 1TB of tenant data as an example. After detecting the RT-TTP of a tenant-group of the past 24 hours has dropped below the, say, 99.9% performance SLA guarantee threshold, Thrifty needs about 14.5 hours (50446s+1779s) to prepare the new MPPDB. Assume that the RT-TTP of the tenant group is consistently 98% during those 14.5 hours, that would result in about 17.4 minutes (14.5 hours $\times$ 2%) having concurrent query processing at MPPDB$_0$. Consider a reasonable time frame of, say, one month, Thrifty has about 43 minutes (1 month $\times$ 0.1%) of "grace period" for not meeting the SLA guarantee of a tenant group.[1] So, that grace period is barely enough to tolerate the scaling of A twice. Therefore, we look for a more

[1]That 43 minutes is only an optimistic estimate because some tenants should have already consumed some of that grace period before Thrifty takes action.

lightweight approach to implement the elastic scale-up operation such that it can be tolerated more scale-up operations or possibly other performance-influencing operations (e.g., node failure) in the grace period.

The lightweight approach is to identify the "over-active" tenant(s) in that tenant group and add a new MPPDB to serve only those tenant(s), instead of adding a MPPDB that serves *the whole tenant-group*. Table 5.1 shows that the data loading time scales linearly with the data size. As the data size of a tenant in a tenant-group must always be less than the data size of *all* tenants in that group, the elastic scale-up operation that starts a new MPPDB and loads data for only the over-active tenants would be more efficient. Specifically, on detecting the RT-TTP of a tenant-group of the past 24 hours has dropped below P, Thrifty will run an over-active-tenant-identification algorithm to identify the tenant(s) that are more active than the history indicated, and create a new MPPDB to serve them. The over-active-tenant-identification algorithm is similar to the tenant-grouping algorithm presented in Algorithm 2, with the only change that takes only the tenants of a particular tenant-group as inputs. It is expected some tenant(s) cannot join the same tenant group anymore, and they are identified as over-active and their data are bulk loaded[2] to a new MPPDB. When the new MPPDB is ready, the Deployment Advisor will notify the Query Router to route queries from the over-active tenant(s) to the new MPPDB.

For any tenant-group that has ever gone through the above elastic scaling process, Thrifty will add tenants in those tenant-groups to a re-consolidation list. Those tenants, together with new tenants, over-active tenants, and tenants

[2]For MPPDB products that support parallel data loading, this system would enable that option.

in tenant-groups with de-registered tenants, will get re-consolidated in the next (re)-consolidation cycle. Finally, tenants with regular bursts in tenant activity (e.g., there are usually bursts near the end of a fiscal year) could be identified by Thrifty's regular activity monitoring and they would be excluded from consolidation before the bursts arrive.

We regard its approach as a reactive approach to maintain the SLA performance guarantee. A pessimistic approach is to use $A + 1$ MPPDBs upfront to serve a tenant group, which is not cost-effective. A proactive approach is to predict at run-time whether the RT-TTP will soon drop below P and proactively trigger lightweight elastic scaling if so. That approach, however, is subjected to prediction error and spikes (e.g, sharp drop of RT-TTP followed by sharp rise) in tenant activities.

Chapter 6

Manual Tuning

The elastic scaling operation of Thrifty automatically adds a MPPDB to serve the over-active tenants when the RT-TTP of a tenant-group of the past 24 hours stays below the SLA performance guarantee P. That MPPDB will be there until the next re-consolidation cycle (to minimize the overhead of scaling up and down). A system administrator may override this action. Consider a performance SLA guarantee of 99.9%, and a tenant-group of tenants served by three 10-node MPPDBs ($A = R = 3$). If there is a minor increase of active tenant ratio of that group, resulting in a RT-TTP of, say, 99.8%, in the past 24 hours, and if a system administrator realizes from the Tenant Activity Monitor that the RT-TTP of that group is not continuously dropping but stays flat, she may think that Thrifty's elastic scaling that starts a new 10-node MPPDB for the over-active tenants is overkilling (because there is only 99.9%−99.8%=0.1% of tiny time percentage of possibly SLA violations).

In situations like this, the system administrator may instead use the parameter U to manually tune the performance SLA guarantee of that tenant-group. Recall that the tenant-driven design reserves a parameter U to control the number of nodes used by the MPPDB_0 (Chapter 4). In the example above, the system administrator may override Thrifty's elastic scaling but increase U from 10 to, say, 12. Recall that the query routing algorithm of Thrifty routes a query to MPPDB_0 if all MPPDBs are busy (Algorithm 1 Line 10). Whenever the 0.2% of time the fourth (or the fifth...) active tenant submits a query, that is routed to MPPDB_0 for concurrent processing. So, it is possible that the 99.9% SLAs can be met *empirically* by having a higher degree of parallelism and more computation power (U+2) in MPPDB_0 (like Point C in Figure 1.1b).

Chapter 7

Evaluation

Multi-tenancy is an important topic in cloud database research. However, due to privacy, tenant logs are seldom available to the public. In view of this, we begin by describing a methodology to create close-to-realistic multi-tenant DaaS tenant logs. Next, we study the consolidation effectiveness of Thrifty based on those generated tenants. Finally, we evaluate the lightweight elastic scaling feature of Thrifty.

7.1 Generation of Tenant Logs

The objective of this section is to describe a methodology to generate close-to-realistic MPPDBaaS tenant activity logs . The resulting logs are useful to validate the consolidation effectiveness of the two-step tenant grouping algorithm.

To imitate the realistic scenarios, the log generation process has two steps: (1) Real Query Log Collection and (2) Multi-Tenant Log Composition. In the first step, we imitate the activity of different kinds of tenants, submit queries to MPPDBs, and collect the corresponding real query logs from the MPPDBs. In the second step, we compose activity logs of a large number of tenants from those real query logs.

Step 1: Real Query Log Collection We use TPC-H and TPC-DS data and queries in this step. A commercial MPPDB product is used to create five MPPDB instances: 2-node, 4-node, 8-node, 16-node, 32-node. We assumes a tenant has more data if it requests more nodes. A tenant may either hold TPC-H data or TPC-DS data (with equal probability). So, the 2-node, 4-node, 8-node, 16-node, and 32-node MPPDB instances respectively hold 200GB, 400GB, 800GB, 1.6TB and 3.2TB of TPC-H/TPC-DS data (each node gets a 100GB data partition).

We then imitate a tenant and submit queries to each MPPDB by following the procedure below: Each tenant has at most S autonomous users, where S is a random integer between 1 and 5. Each user follows a probability distribution P to carry out the following: (a) either submits a random TPC-H/DS query to a MPPDB, or (b) submits a batch of M random TPC-H/DS queries to a MPPDB, where M is a random integer between 1 and 10. The user will not take any action until the single query (for (a)) or the query batch (for (b)) is complete. A batch of queries is complete only when all its submitted queries have finished. After the completion of a query/query batch, a user will pause for W seconds before the next event takes place, where W is a random integer from 3 to 600. We have repeated the above procedure 100 times. Each time the above procedure is carried out for 3 hours, on the 2-node, 4-node, 8-node, 16-node, and 32-node MPPDBs, using a uniform distribution as P. The query logs of the above executions are collected. Each query log collected is essentially a 3-hour real query log of an artificial tenant, which requests, say, a 16-node MPPDB with a maximum of 4 active users.

Step 2: Multi-Tenant Log Composition There are two inputs in this step: (i) the total number of tenants T; and (ii) the skewness of the tenant size distribution. The skewness of the tenant size is chosen by sampling from the CDF of a Zipf distribution with a parameter $0 < \theta < 1$, where a smaller θ tends to uniform whereas a larger θ tends to skew. The default θ is 0.8.[1] An example input in this step looks like Figure 5.2, where $T = 5400$ but the tenant can only request 2/4/8/16/32-node MPPDB instances because only it have been prepared in Step (1).

Given the tenant characteristics, the next step is to generate the tenant activity log of each tenant. We illustrate this by giving an example of generating a one-day activity log of a tenant that requests a 4-node MPPDB instance. In the beginning, the tenant first gets a random time offset O from the following:

Offset O	+0	+3	+5	+8	+16	+17	+19
Example	Seattle	New York	Sao Paulo	London	Beijing	Japan	Sydney

Next, the tenant randomly picks a 4-node 3-hour query log from the logs prepared in Step 1, and copies that into its activity log but with the time offset O added. That offset is used to imitate the start of the office hour of that tenant in a particular time zone. After that, the tenant randomly picks another 4-node 3-hour query log from the logs prepared in Step 1, and copies into its activity

[1]According to [11], the database sizes (measured in terms of amount of data) of different companies follow a skew distribution. Among the parallel database user community, a company tends to determine the number of nodes based on the data size. So, the tenant size also follows a skew distribution.

log but with the time offset $O + 3 + 2$ added. That offset is used to imitate the start of the afternoon office hour after three hours of morning office hour plus two hours of lunch. Finally, the tenant randomly picks yet another 3-hour query log, and copies into its activity log but with the time offset $O + 3 + 2 + 9$ added. That offset is used to imitate some report generation activities scheduled 6 hours after the office hour and some queries posed by users in remote offices (in different time zones).

We generate 30-day query activities for each tenant. In each week, each tenant is active for five (week)days and then inactive two days (for the weekend). Within the 30 days, each tenant is inactive in two weekdays to imitate two days of public holiday. That two days are randomly chosen, but they are the same for the tenants in the same time zone.

7.2 Experimental Setting

All the experiments are done on Amazon EC2 Extra Large Instance (15 GB memory and 8 EC2 Compute Units) running a commercial MPPDB on Linux. It follows the physical design suggested by the accompanied tuning advisor of the MPPDB to do the partitioning and create the indexes. The MPPDB product used in this experiment supports parallel loading, so it enables that option. Thrifty is implemented using Python and it uses $U = n_1$.

7.3 Consolidation Effectiveness under Different Tenant Characteristics

This first set of simulation experiments is to evaluate the consolidation effectiveness of the two-step tenant-grouping algorithm under different tenant characteristics. The consolidation effectiveness is measured as the percentage of nodes saved. For example, a 80% consolidation effectiveness means that if the tenants all together request 10000 machine nodes, Thrifty can serve all of them using 2000 nodes only, meaning 8000 machine nodes are saved.

Table 7.1 shows the parameters varied in this part of experiments. The default values are in bold face. The generated logs according to these parameters have average active tenant ratios range from 8.9% to 12%, which is close to the real 10% in DaaS [21]. The active tenant ratio under the default parameter value is 11.9%. An optimal tenant-grouping solution based on solving the MINLP formulation (Appendix 9.1) using DIRECT [14] is implemented but it has taken about 12 days to compute the optimal solution for only 20 tenants. Therefore, this experiments only compare with the FFD heuristic. In this experiments, the 2-step heuristic algorithm always generates better deployment plans by saving 3.6% to 11.1% more nodes than FFD.

Parameter	**Ranges**
Epoch Size (E)	0.1s, 1s, **10s**, 30s, 90s, 600s, 1800s
Number of Tenants (T)	1000, **5000**, 10000
Tenant Distribution (θ)	0.1, 0.2, 0.5 **0.8**, 0.99
Replication Factor (R)	1, 2, **3**, 4
Performance SLA (P)	95%, 99% **99.9%**, 99.99%

Table 7.1. Evaluation Parameters

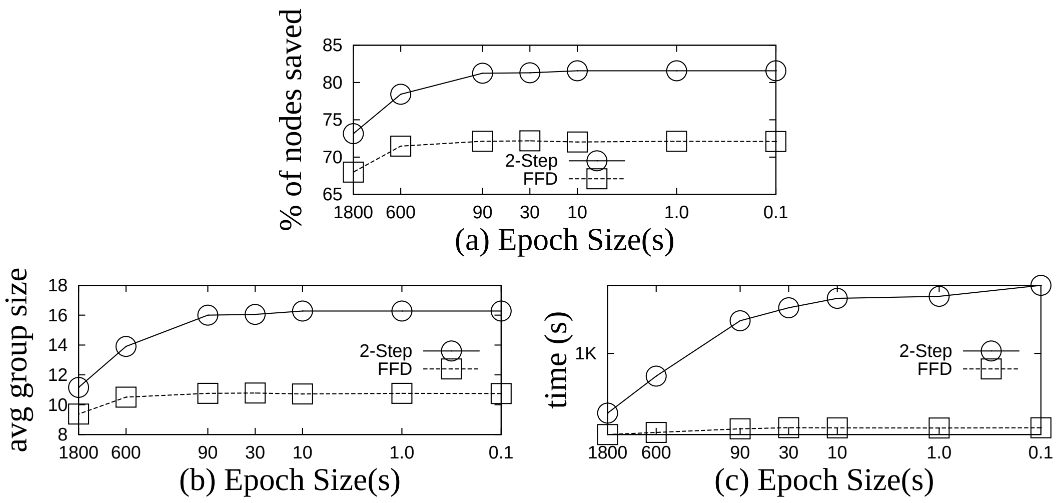


Figure 7.1. Varying Epoch Size E

Varying Epoch Size Recall from Chapter 5 that the consolidation effectiveness of Thrifty is influenced by the choice of epoch size E in the tenant-grouping algorithm. This experiment is to evaluate the influence of that parameter and let us choose a suitable one for the rest of the experiments. Figure 7.1a shows the consolidation effectiveness of Thrifty under different choices of E. When the epoch size is large, 1800s, the consolidation effectiveness of FFD and the 2-step heuristic are about 68.0% and 73.1%, respectively. The consolidation effectiveness increases with the use of smaller epoch size. For the 2-step heuristic, the consolidation effectiveness is high up to 81.5% when a 10s epoch size is used, where smaller epoch size does not yield further improvement. Figure 7.1c shows the execution time of running FFD and the 2-step heuristic algorithms. FFD is efficient because it just spends $\mathcal{O}(T \log T)$ time to sort the tenants. Although slower, the 2-step heuristic algorithm consistently generates deployment plans that save 5.1% to 9.4% more nodes than FFD over all epoch sizes and it finishes

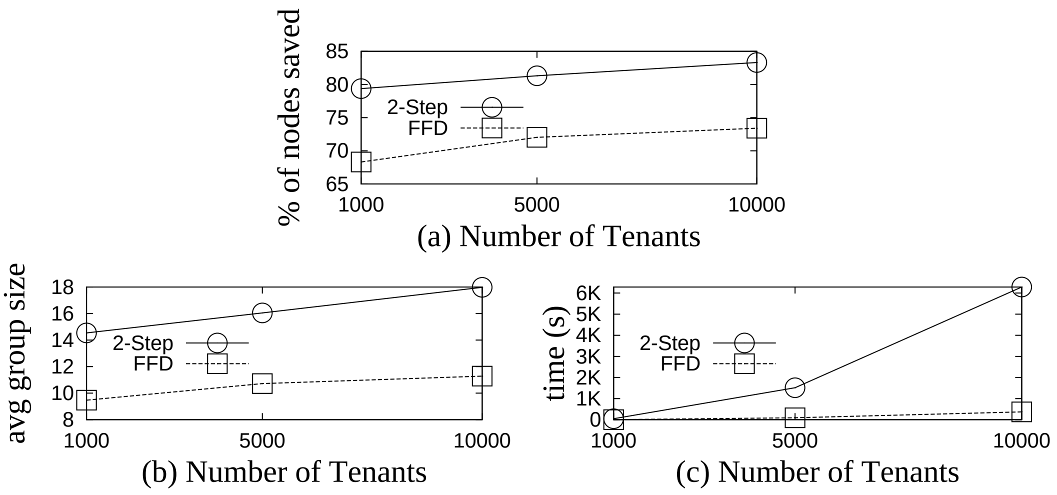


Figure 7.2. Varying Number of Tenants T

in 30 minutes even with the finest 0.1s epoch size is used. So, I adopt 10s as the default epoch size. Figure 7.1b shows the average size of the resulting tenant-groups. Tenant-groups formed by the 2-step heuristic algorithm can pack more tenants than FFD.

Varying Number of Tenants Figure 7.2a shows that Thrifty's consolidation effectiveness is not significantly influenced by the number of tenants. When the number of tenants increases from 1000 to 10000, consolidation effectiveness of the 2-step heuristic is increased from 79.3% to 83.3%. The minor increase is brought by giving more choices to the algorithm to pick when inserting a tenant into a tenant-group. Figure 7.2b supports the observation by showing that the average tenant-group size also slightly increases with the number of tenants. Figure 7.2c shows that the execution time of the 2-step heuristics algorithm. Overall, the 2-step heuristic spends about two hours to find a deployment plan that saves 83.3% of nodes for 10000 tenants. In contrast, although the FFD heuristic spends only about 6 minutes, it finds a deployment plan that can only save 73.4% of nodes

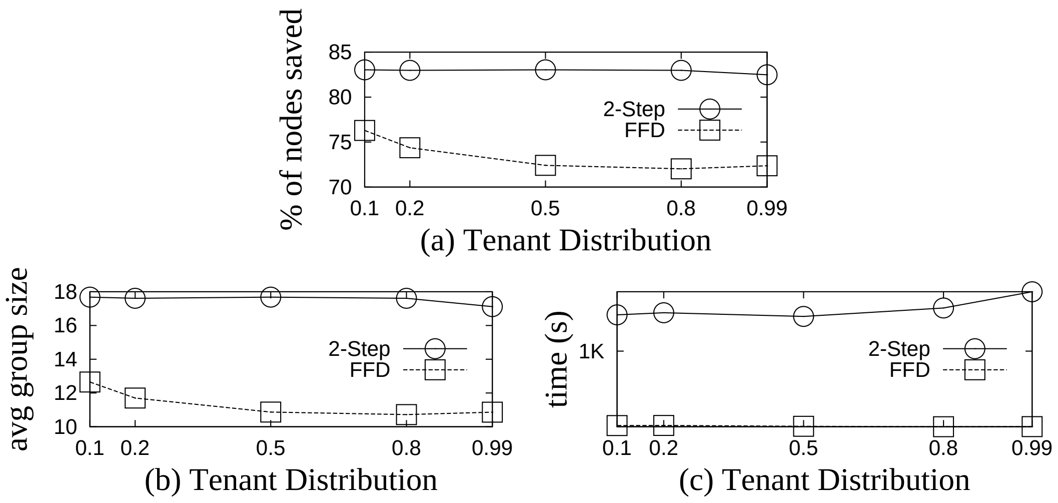


Figure 7.3. Varying Tenant Distribution θ

for 10000 tenants.

Varying Tenant Distribution Figure 7.3a shows that the consolidation effectiveness of the 2-step heuristic is not influenced by the tenant distribution as significant as FFD does. Figure 7.3b supports the observation. Figure 7.3c shows that the tenant distribution has a small influence on running time of the 2-step heuristic, because the second step of this heuristic scans the list of tenants once per insertion of a tenant into a tenant-group and thus an initial group with more tenants takes more time to finish the grouping.

Varying Replication Factor This experiment is to evaluate the consolidation effectiveness under different replication factors R. As one of the considerations of the two-step tenant-group algorithm is to ensure that the maximum number of active tenants in the resulting tenant-group does not exceed R for $P\%$ of time, a higher replication factor would put more tenants into a tenant-group. Figure 7.4b shows that the average tenant-group size of 2-step heuristic is increased from

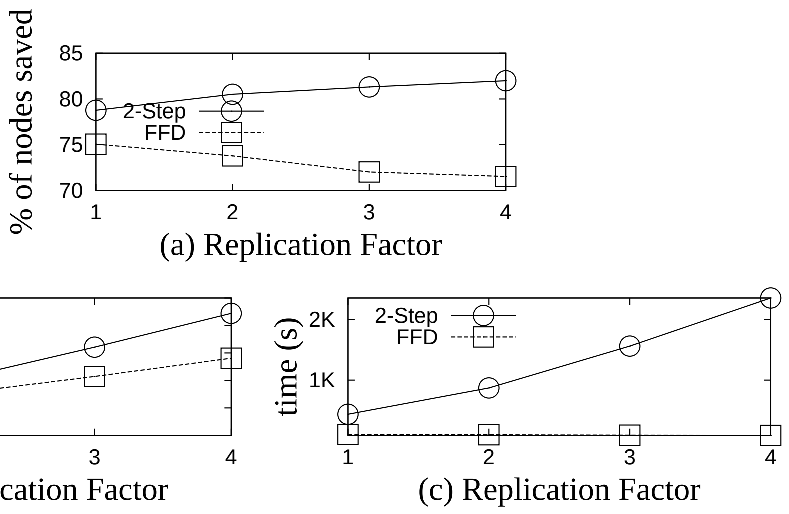
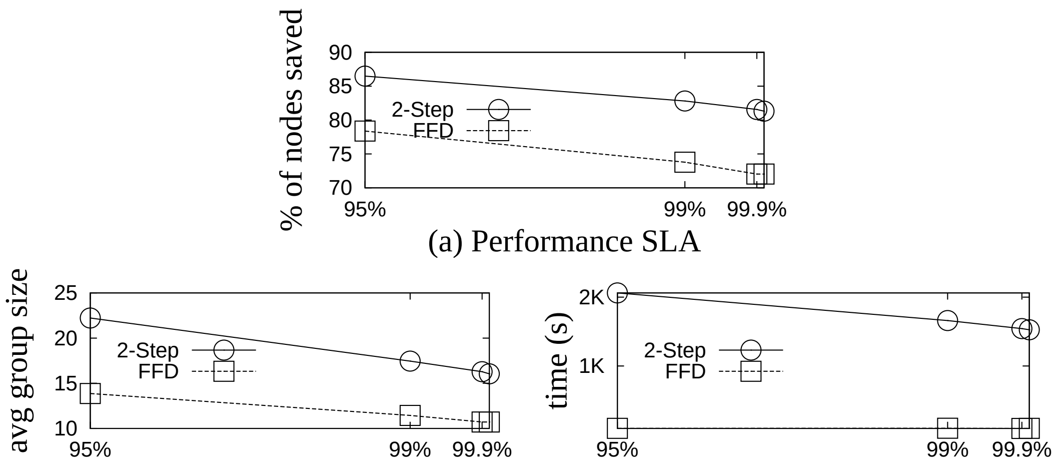


Figure 7.4. Varying Replication Factor R

4.7 to 22.2 , when R is increased from 1 to 4. Figure 7.4a shows that 2-step's consolidation effectiveness is increased from 78.8% to 82.0% accordingly. The increase of consolidation effectiveness is not as significant as the increase of the average tenant-group size because when R is 4, more machines are used, and that waters down the number of machines saved. Figure 7.4c shows that the running time of the 2-step heuristic increases with R because more tenants can be tried when inserting a tenant into a tenant-group.

Varying Performance SLA guarantee This experiment is to evaluate the consolidation effectiveness under different performance SLA guarantees. Figure 7.5a shows that the consolidation effectiveness of the 2-step heuristic can be as high as 86.5% if only a 95% performance guarantee is required, and it drops back to 81.6% and 81.3%, respectively, when more stringent performance SLA guarantees, 99.9% and 99.99%, are required. The experimental results show that 99.9% is stringent enough such that an even more stringent performance



(a) Performance SLA

(b) Performance SLA

(c) Performance SLA

Figure 7.5. Varying Performance SLA P

SLA guarantee of 99.99% does not reduce the consolidation effectiveness further. When the performance guarantee is not so stringent (95%), Figure 7.5b shows that both heuristics can pack more tenants into the same tenant group, and Figure 7.5c shows that the 2-step heuristics has a higher running time than FFD because more tenants can be tried to be inserted into a tenant-group.

7.4 Consolidation Effectiveness under Higher Active Tenant Ratio

This second set of experiments is mainly used to get a feeling of the consolidation effectiveness under some unusual scenarios where the active tenant ratio is much higher than 10%. We generate tenant logs that with higher active tenant ratios by using the default parameter values listed in Table 7.1 but with the following modifications in Step 2 (Multi-Tenant Log Composition) of the log generation process:

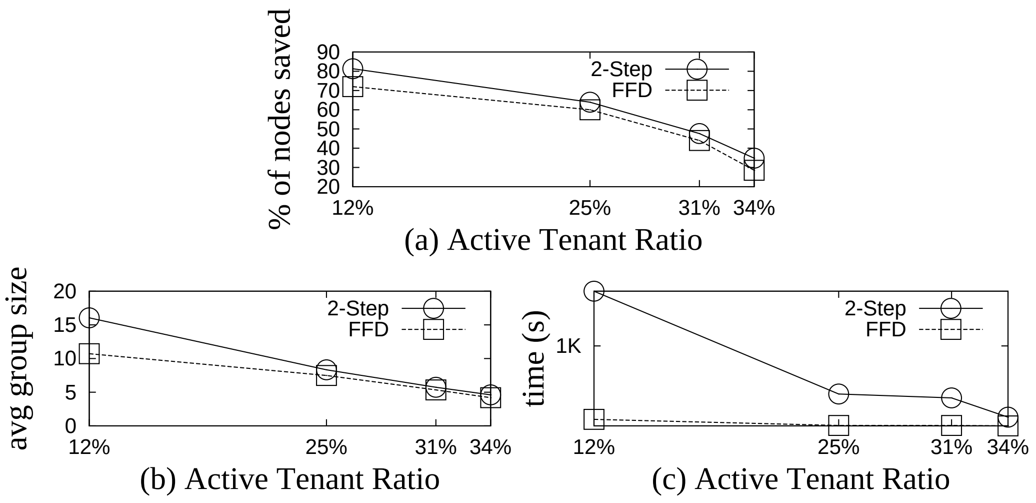
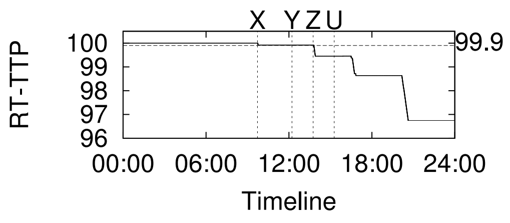


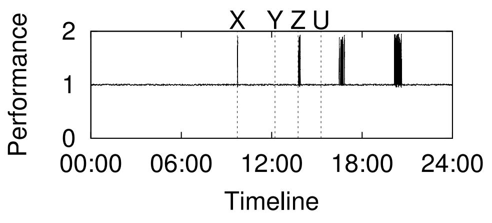
Figure 7.6. Higher Active Tenant Ratio

	Modification Made & Meaning	Resulting Active Tenant Ratio
	-	11.9%
(1)	Tenants get either +0 or +3 offsets only (imitating tenants are all from North America)	25.1%
(2)	Tenants of (1) but with no lunch hour	30.7%
(3)	All tenants get the same +0 offset (imitating tenants are all from the west coast) and with no lunch hour	34.4%

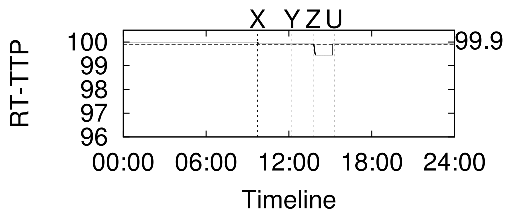
Figure 7.6a shows the consolidation effectiveness of the 2-step heuristic algorithm drops from 81.3% to 34.8% when the active tenant ratio increases from 11.9% to 34.4%. That makes sense because when the history shows that many tenants have been very active, the tenant-grouping algorithm cannot put many tenants into the same tenant group. When the active tenant ratio is high up



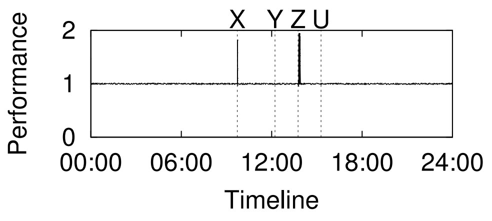
(a) RT-TTP (w/o elastic scaling)



(b) Query performance (w/o elastic scaling)



(c) RT-TTP (w elastic scaling)



(d) Query performance (w elastic scaling)

Figure 7.7. Lightweight Elastic Scaling in a Tenant Group

to 34.4%, Figure 7.6b shows that a tenant-group can contain only five tenants on average. In this experiment, $R = 3$, so it means Thrifty has to use three MPPDBs to serve five tenants and thus only nodes requested by two tenants could be saved, resulting in a 34.8% consolidation effectiveness.

Generally the active tenant ratio in a realistic DaaS has active tenant ratio as low as 10% [21]. For tenants that have high active tenant ratio, as mentioned, Thrifty could detect them easily and exclude them from consolidation. Even when the active tenant ratio is high up to 30.7% and 34.4%, respectively, the 2-step heuristic can still save 47.6% and 34.8% of the requested nodes, and not to forget that includes three replicas for each tenant as a high availability service.

7.5 Lightweight Elastic Scaling

In this section, we aim to take a look at how Thrifty's lightweight elastic scaling takes action in practice. It chooses to present some real production information about a tenant-group generated from the previous tenant-grouping experiment that uses the default values (Table 7.1). That tenant-group consists of 14 tenants that request 4-node MPPDBs ($R = 3$; deploy on EC2 clusters). Figure 7.7a shows an excerpt of run-time tenant-activities of that tenant-group with Thrifty's elastic scaling disabled. In the beginning of the excerpt, the RT-TTP of that tenant-group is 100%, meaning there were at most three active tenants in the past 24 hours. At time X, the RT-TTP of that tenant-group slightly dropped to 99.92%. We checked the query log and found that there was once a fourth tenant active and MPPDB$_0$ had concurrently processed two queries for a very short duration. Figure 7.7b shows the corresponding excerpt of the performance of queries in Thrifty. The performance is normalized, so 1.0 means a query has finished execution as quick as it should be when measured in an isolated environment. A performance, say, 1.2 means a query has finished execution 1.2 time slower than it should be when measured in an isolated environment. At time X where there were two queries concurrently processed by MPPDB$_0$, that two queries respectively obtained 50% and 80% query latency delay. However, that was the first such event in the past 24 hours and the duration of that event was short. Thus no elastic scaling took place — this showcases how Thrifty's reactive scaling approach can deal with some one-off spikes.

In order to study the effect of Thrifty's elastic scaling, we manually took over a tenant at time Y and continuously submitted queries to the system on behalf of that tenant. Figure 7.7c shows the run-time tenant-activities of that tenant-group with Thrifty's elastic scaling enabled. From time Y to time Z, only two tenants were active concurrently and thus nothing happened even though it was submitting queries continuously as being the third tenant. At time Z, three other tenants became concurrently active and this time the tenant-group had accumulated more than 0.1% of time more than three active tenants in the past 24 hours. Therefore, Thrifty carried out elastic scaling. Thrifty took about 2 seconds to identify the over-active tenant and it spent about 5000 seconds to load the data of the over-active tenant. At time U, the new MPPDB was ready. Since then, Thrifty routed all the queries to the new MPPDB and the tenant-group excluded all the activities of the removed tenant. Therefore, the RT-TTP of that tenant group returned back to 99.9% and above. Figure 7.7d shows the corresponding excerpt of the performance of queries in Thrifty with elastic scaling enabled. As multiple queries were concurrently processed by MPPDB$_0$ at time Z, they exhibited performance delay. However, after scaling up, Thrifty had been able to deal with the subsequent over-active scenarios. In contrast, Figure 7.7b shows the expected scenarios where some subsequent queries cannot meet the performance SLA as it kept on submitting queries to the system while the elastic scaling option was disabled. These together showcase how Thrifty's elastic scaling approach can deal with over-active tenants that deviate from the history.

Chapter 8

Future Work and Conclusion

This thesis presents Thrifty, a prototype that offers massively parallel processing relational database system as a service (MPPDBaaS). The technologies associated with Thrifty include the cluster design that details the arrangement of thousands of machines nodes to form MPPDBs of different degree of parallelism, the tenant-placement that details the tenant-to-MPPDB assignment, the query routing algorithm that routes tenant queries to a proper MPPDB for processing, the implementation of elastic scaling, and the system tuning methodologies. Experimental results show that Thrifty can save around 73.1% to 86.5% of nodes in a wide range of usual settings.

Thrifty is currently a research prototype and there are a lot of interesting future work. First, Thrifty currently assumes the machine nodes in the cluster are homogeneous, extending Thrifty to deal with a cluster of heterogeneous machines is thus an important yet challenging task. Second, data privacy is an important issue in DaaS. Fortunately, privacy-aware query processing techniques have no significant difference between centralized databases and parallel databases. We plan to incorporate techniques like adjustable security (e.g., [7]) into Thrifty. Third, currently Thrifty targets a very general setting where some tenants may submit ad-hoc queries. Our plan is to separate the tenants in Thrifty into two classes in the future, where tenants that would not submit ad-hoc queries (e.g., those only run reporting generation applications and agree to let us extract their query templates) are treated as a special class (they will get another service plan). For tenants in that class, it will be hosted using a specialized tenant-driven divergent design that uses $U > n_1$ nodes for MPPDB$_0$ *upfront* and use different partition schemes for different MPPDBs [6] in order to deal with the non-linear scale-out problem. That would be a special case of the tenant-driven design. Although only applicable to a restricted type of tenants, that design would incur fewer elastic scalings and with a higher consolidation effectiveness. The crux of that specialized design is to identify the minimum value of U that can afford different degrees of concurrent query processing on MPPDB$_0$ without performance SLA violations, which requires extending concurrent workload performance prediction techniques to parallel databases.

Bibliography

[1] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *EDBT*, pages 449–460, 2011.

[2] M. Akdere, U. etintemel, M. Riondato, E. Upfal, and S.B. Zdonik. Learning-based query performance modeling and prediction. In *ICDE*, pages 390 – 401, 2012.

[3] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *SIGMOD*, pages 1195–1206, 2008.

[4] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL server for cloud computing. In *ICDE*, pages 1255–1263, 2011.

[5] Yanpei Chen, Sara Alspaugh, and Randy H. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *PVLDB*, 5(12):1802–1813, 2012.

[6] Mariano P. Consens, Kleoni Ioannidou, Jeff LeFevre, and Neoklis Polyzotis. Divergent physical design tuning for replicated databases. In *SIGMOD*, pages 49–60, 2012.

[7] Carlo Curino, Evan P. C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD*, pages 313–324, 2011.

[8] Carlo Curino, Evan P. C. Jones, Raluca A. Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: a database service for the cloud. In *CIDR*, pages 235–240, 2011.

[9] Jennie Duggan, Ugur etintemel, Olga Papaemmanouil, and Eli Upfal. Performance prediction for concurrent database workloads. In *SIGMOD*, pages 337–348, 2011.

[10] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and A. Kemper. An integrated approach to resource pool management: Policies, efficiency and quality metrics. In *DSN*, pages 326 –335, 2008.

[11] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, pages 243–252, 1994.

[12] Mei Hui, Dawei Jiang, Guoliang Li, and Yuan Zhou. Supporting database applications as a service. In *ICDE*, pages 832 –843, 2009.

[13] Dean Jacobs and Stefan Aulbach. Ruminations on multi-tenant databases. In *BTW Proceedings*, pages 514–521, 2007.

[14] Donald R. Jones. Direct global optimization algorithm. *Encyclopedia of Optimization*, pages 725–735, 2009.

[15] Sangmin Lee, Rina Panigrahy, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Validating heuristics for virtual machines consolidation. In *Microsoft Research Technical Report*, 2011.

[16] Gang Luo, Jeffrey F. Naughton, and Philip S. Yu. Multi-query SQL progress indicators. In *EDBT*, pages 921–941. Springer, 2006.

[17] Rimma V. Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, pages 1137–1148, 2011.

[18] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. In *Microsoft Research Technical Report*, 2011.

[19] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.

[20] Jun Rao, Eugene J Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment*, 4(4):243–254, 2011.

[21] Berthold Reinwald. Multitenancy. *University of Washington and Microsoft Research Summer Institute*, 2010.

[22] J. Schaffner, B. Eckart, D. Jacobs, C. Schwarz, H. Plattner, and A. Zeier. Predicting in-memory database performance for automating cluster management tasks. In *ICDE*, pages 1264 –1275, 2011.

[23] Oliver Schiller, Benjamin Schiller, Andreas Brodt, and Bernhard Mitschang. Native support of multi-tenancy in RDBMS for software as a service. In *EDBT*, pages 117–128, 2011.

[24] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *SOCC*, pages 5:1–5:14, 2011.

[25] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic virtual machine configuration for database workloads. *ACM TODS*, 35(1), 2010.

[26] Petrie Wong, Zhian He, and Eric Lo. Parallel analytics as a service. In *SIGMOD*, pages 25–36, 2013.

Chapter 9

Appendix

9.1 Formulation of the LIVBPwFC

$$
\begin{aligned}
\text{Minimize} \quad & \textstyle\sum_{j=1}^{\lceil \frac{T}{R} \rceil} \max_i (R * n_i * x_{ij}) && (9.1)\\
\text{Subject to} \quad & \textstyle\sum_{k=1}^{d} H[R - \sum_{i=1}^{T} \vec{A}_i[k] * x_{ij}] \geq P\% * d, \forall j && (9.2)\\
& \textstyle\sum_{j=1}^{\lceil \frac{T}{R} \rceil} x_{ij} = 1, \forall i && (9.3)\\
& x_{ij} \in \{0, 1\}, \forall i, \forall j && (9.4)
\end{aligned}
$$

where $i \in [1, \cdots, T]$, $j \in [1, \cdots, \lceil \frac{T}{R} \rceil]$ and $H[n]$ is the discretization format of the *Heaviside step function*:

$$
H[n] = \begin{cases} 0, & \text{if } n < 0, \\ 1, & \text{otherwise.} \end{cases}
$$

Variable $x_{ij} = 1$ if tenant T_i is packed to tenant-group TG_j or $x_{ij} = 0$ otherwise.

Objective function: There are T tenants. Each tenant-group has R replicas and can support R concurrently active tenants. So, there are at most $\lceil \frac{T}{R} \rceil$ tenant-groups. Each tenant-group TG_j requires $R * n_i$ machines, where i is the largest tenant that requests the most nodes in that tenant-group.