UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

SLA-BASED PERFORMANCE TUNING TECHNIQUES FOR CLOUD DATABASES

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

LIANGZHE LI
Norman, Oklahoma
2017

SLA-BASED PERFORMANCE TUNING TECHNIQUES FOR CLOUD DATABASES

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

____
Dr. Le Gruenwald, Chair

____
Dr. Qi Cheng

____
Dr. Sudarshan Dhall

____
Dr. Changwook Kim

____
Dr. Scott Moses

© Copyright by LIANGZHE LI 2017
All Rights Reserved.

Acknowledgements

I have been working on this research for more than three years and it has been impossible to complete this task on my own. Many people provided their support, guidance and encouragement to me.

First I want to thank all the members of my dissertation committee: Dr. Le Gruenwald, Dr. Qi Cheng, Dr. Sudarshan Dhall, Dr. Changwook Kim and Dr. Scott Moses. Without their guidance I could not have successfully finished my research and had my results published. I also want to thank other members of the OU Database Group for helping me on this research. A special gratitude I owed to Dr. Gruenwald is that she spent a lot of time advising me on my work. I will always be grateful for her assistance.

However, I wish to give my greatest gratitude to my family, my parents, my wife and my daughter. I was supposed to stay with you every day but you have seen me less for the past three years because of my busy work. Especially, I really want to thank my daughter, Sunny Li. Though you are too little and had no chance of helping me on my task, you brought so much happiness to me. I hope you know that I love you so much.

A lot of friends in the School of Computer Science offered their help, too. I cannot list all their names here, I thank you all.

Table of Contents

Acknowledgements iv

List of Tables ix

List of Figures x

Abstract xii

Chapter I: Introduction 1

1. Objective 1
2. Database as a Service (DbaaS) 1
3. System Tuning for DbaaS 4
4. Issues of Database Partitioning on Tuning DbaaS 6
5. Contribution 9
6. Organization 10

Chapter II: Literature Review 12

1. Database Buffer Management Algorithms 12
2. Resource Provisioning Algorithms 15
3. Database Partitioning Algorithms 21
   - 3.1. Database Partitioning in Non-Distributed Environments 21
   - 3.2. Database Partitioning in Single-Tenant Distributed Environments 28
   - 3.3. Horizontal Database Partitioning in Multiple-Tenants Distributed Environments 32

Chapter III: A Proposed Buffer Pool Management Technique for Cloud Databases 35

1. Motivation of SLA-LRU 36
2. Overview of SLA-LRU 37

3. Buffer pool level related SLA penalty cost model 38

4. Workflow of SLA-LRU 42

5. Overheads of SLA-LRU comparing with LRU-2 46

Chapter IV A Proposed Performance Tuning Algorithm Based on Database Partitioning for Cloud Databases 50

1. Cost Forecasting for Resource Provisioning 50

1.1. CPU utilization patterns for dynamic provisioning 51

1.2. Problem modeling of cost forecasting for dynamic resource provisioning 51

2. Cost Forecasting for Database Partitioning 55

2.1. Factors impacting database partitioning 55

2.2. Artificial Neural Network on cost prediction for database partitioning . 56

3. Partition Distribution 59

3.1. Motivation of partition distribution for DbaaS 60

3.2. Computing load balance using the overload score 62

3.3. Communication cost 67

Chapter V Performance Analysis 72

1. Theoretical Analysis 72

1.1. Complexity analysis for SLA-LRU 72

1.1.1. Time complexity of SLA-LRU 73

1.1.2. Space complexity of SLA-LRU 75

1.2. Complexity analysis for AutoClustC 75

1.2.1. Time complexity of AutoClustC 76

1.2.2. Space complexity of AutoClustC 79

1.3. Summary of worst-case time and space complexity analysis results 81

2. Experimental Analysis 81

2.1. Simulation Models 82

2.1.1. Simulation model for SLA-LRU 82

2.1.2. Simulation model for AutoClustC 84

2.2. Competitive algorithms 86

2.3. Performance metrics 87

2.3.1. Performance metrics for SLA-LRU 87

2.3.2. Performance metrics for AutoClustC 88

2.4. Experimental results 91

2.4.1. Experimental results for SLA-LRU 91

2.4.1.1. Comparison of query response time of LRU-2, MT-LRU and SLA-LRU 91

2.4.1.2. Comparison of SLA penalty costs of LRU-2, MT-LRU and SLA-LRU 92

2.4.2. Experimental results for AutoClustC 98

2.4.2.1. Performance of the ANN Model for Database Partitioning Cost Forecasting 98

2.4.2.2. Performance of the AR(2) model for Resource Provisioning Cost Forecasting 100

2.4.2.3. Ratio of monetary Cost of Resource Provisioning Cost to Monetary Cost of Database Partitioning 100

2.4.2.4. Performance of the New Database Partitions 101

2.4.2.5. Query response time improvement of processing the whole TPC-H benchmark query set with partition distribution over the query response time without partition distribution 103

Chapter VI Conclusions and Future Work 108

1. Summary of Performance Evaluation Results 110

1.1. Summary of the performance results for SLA-LRU 110

1.2. Summary of performance results for AutoClustC 111

2. Future Research 112

List of Tables

Table 1. List of symbols used by SLA-LRU 38

Table 2. Variables used in the theoretical analysis 72

Table 3. An example entry in the SLA penalty cost table 73

Table 4. Summary of worst-case time and space complexity analysis results 81

Table 5. Tenant's category 83

Table 6. Buffer pool level based SLA penalty cost function for SLA-LRU 84

Table 7. Hit ratio degradation based SLA penalty cost function for MT-LRU 84

Table 8. Sample dataset used for ANN training 85

Table 9. Eight tenants' categories 93

Table 10. Promised buffer pool level for each tenant for SLA-LRU 94

Table 11. Actual buffer pool level for each tenant before running algorithms 94

Table 12. Actual buffer pool level for each tenant after running LRU-2 95

Table 13. Penalty cost for each tenant after running LRU-2 95

Table 14. Overall performance of SLA-LRU and MT-LRU 98

Table 15. Parameters used by the database partitioning algorithm in AutoClustC 104

Table 16. Specifications of the virtual environment 105

Table 17. Communication delay for each server pair 105

List of Figures

Figure 1. Cloud workload patterns 17

Figure 2. Main idea of LRU-K (K=2) 36

Figure 3. Penalty cost function examples 40

Figure 4. Penalty cost change for tenant A and tenant B with SLA violation penalty function of pattern (a) and pattern (b), respectively 42

Figure 5. Two portions of a buffer page list 44

Figure 6. Algorithm of analyzing SLA violation penalty cost 48

Figure 7. Algorithm of releasing buffer pages 49

Figure 8. Dynamic forecasting estimation 52

Figure 9. The trend sub-utilization and residual sub-utilization 54

Figure 10. The ANN model used for partitioning cost forecasting 58

Figure 11. Performance tuning cost analysis in AutoClustC 59

Figure 12. AWS cloud structure 61

Figure 13. PMs and VMs in a data center 63

Figure 14. Partition distribution algorithm 71

Figure 15. Average TPC-H benchmark query set processing time 92

Figure 16. Buffer assignment status (actual buffer pool level) before performing any buffer management algorithm and after performing LRU-2 and SLA-LRU 94

Figure 17. SLA penalty cost ratio of using LRU-2 over using SLA-LRU 96

Figure 18. SLA penalty cost ratio of using LRU-2 over using MT-LRU 97

Figure 19. ANN performance on forecasting database partitioning CPU time 99

Figure 20. Linear regression of the network on forecasting the database partitioning CPU time 99

Figure 21. Probability distribution of prediction error in resource provisioning CPU time forecasting 100

Figure 22. Query response time of 95th percentile of query for different query types before and after database partitioning 103

Figure 23. CPU utilization ratio of PM_1* 106

Figure 24. Average time of processing one TPC-H benchmark query set under the overloaded and un-overloaded status 107

Abstract

Today, cloud databases are widely used in many applications. The pay-per-use model of cloud databases enables on-demand access to reliable and configurable services (CPU, storage, networks, and software) that can be quickly provisioned and released with minimal management and cost for different categories of users (also called tenants). There is no need for users to set up the infrastructure or buy the software. Users without related technical background can easily manage the cloud database through the console provided by service providers, and they just need to pay to the cloud service provider only for the services they use through a service level agreement (SLA) that specifies the performance requirements and the pricing associated with the leased services. However, due to the resource sharing structure of the cloud, different tenants' workloads compete for computing resource. This will affect tenants' performance, especially during the workload peak time. So it is important for cloud database service providers to develop techniques that can tune the database in order to re-guarantee the SLA when a tenant's SLA is violated. In this dissertation, two algorithms are presented in order to improve the cloud database's performance in a multi-tenancy environment. The first algorithm is a memory buffer management algorithm called SLA-LRU and the second algorithm is a vertical database partitioning algorithm called AutoClustC.

SLA-LRU takes SLA, buffer page's frequency, buffer page's recency, and buffer page's value into account in order to perform buffer page replacement. The value of a buffer page represents the removal cost of this page and can be computed using the corresponding tenant's SLA penalty function. Only the buffer pages whose tenants have the least SLA penalty cost increment will be considered by the SLA-LRU algorithm when

a buffer page replacement action is taken place. AutoClustC estimates the tuning cost for resource provisioning and database partitioning, then selects the most cost saving tuning method to tune the database. If database partitioning is selected, the algorithm will use data mining to identify the database partitions accessed frequently together and will re-partition the database accordingly. The algorithm will then distribute the resulting partitions to the standby physical machines (PMs) that have the least overload score computed based on both the PMs' communication cost and overload status.

Comprehensive experiments were conducted in order to study the performance of SLA-LRU and AutoClustC using the TPC-H benchmark on both the public cloud (Amazon RDS) and private cloud. The experiment results show that SLA-LRU gives the best overall performance in terms of query response time and SLA penalty cost improvement ratio, compared to the existing memory buffer management algorithms; and AutoClustC is capable of identifying the most cost-saving cloud database tuning method with high accuracy from resource provisioning and database partitioning, and performing database re-partitioning dynamically to provide better query response time than the current partitioning configuration.

Chapter I: Introduction

1. Objective

The objective of this research is to develop a novel performance tuning technique for the cloud database that has the following abilities:

i. Ability to estimate the monetary operational cost for different tuning methods (resource provisioning and database partitioning) and selecting the one with a lower cost;

ii. Ability to monitor and manage the sharing buffer pool among multiple tenants in order to reduce the SLA penalty cost for the service provider;

iii. Ability to distribute different partitions to the proper Virtual Machine (VM) instances in the same data center in order to provide high performance for the cloud database.

In the following sections, we present first the background of cloud databases in Section 2, then existing performance tuning methods for cloud databases in Section 3, research issues that need to be addressed when applying database vertical partitioning to cloud database tuning in Section 4, the contribution of our research in Section 5, and finally the organization of the dissertation in Section 6.

2. Database as a Service (DbaaS)

Cloud database, also called Database as a Service (DbaaS), can provide subscription-oriented, enterprise-quality services with high availability, reliability and scalability [1]. It may be defined as a pay-per-use model for enabling on-demand access to reliable and configurable services that can be quickly provisioned and released with minimal management. Users/tenants need not set up the infrastructure or buy the software, but pay

to the cloud service provider only for the services they use through a performance service level agreement (SLA) that specifies the performance requirements and the pricing associated with the leased services. In recent years, DbaaS in the cloud has attracted a lot of attention. Major IT companies like Amazon, Facebook, Google, IBM, Microsoft and Yahoo! have provided large scale database management services. Some of them, such as Amazon SimpleDB [2], DynamoDB [3], Google Bigtable [4], and Yahoo! PNUTS [5], consist of large scale systems with a simplified query interface. Less scalable but fully relational approaches are also available, e.g., as Amazon RDS [6] and SQL Azure [7]. The reason why many major IT companies invest huge amount of money in the cloud database area is that DbaaS has the following benefits [8]:

i. Easiness in administering the database

   The "ready-to-use" concept in DbaaS makes it easy for users to go from project conception to deployment. All major DbaaS providers provide Management Console, Command-Line Interface, or simple API for users to access the capabilities of a production-ready relational database in minutes. There is no need for the users to perform infrastructure provisioning, install database software or maintain a database system.

ii. Easiness in scaling the database

   The users can scale their database's storage resources or computing resources within only several minutes by typing a few command lines through a Command-Line Interface. If the users think that would be hard for them, they can even scale their database's storage or computing resources by just clicking a few buttons

through the Management Console. This would allow those users who have no solid computer background to re-configure the database very easily.

iii. High availability and durability for the database

DbaaS runs on the highly reliable infrastructures located in IT companies' huge data centers. When a user provisions a database instance, DbaaS synchronously replicates the data to a standby database instance which is generally in a different availability zone or data center. DbaaS also performs backups, snapshots and host replacement automatically. All of these tasks make the database highly available and durable.

iv. Fast database access

Based on the application performance requirements, users may build their database servers with multiple virtual CPUs and may choose different storage options, such as choosing optimized SSD-backed storage for their high-performance required applications or general-purpose magnetic storage for their low-performance required applications in which data is accessed less frequently.

v. Secure database access

DbaaS allows users to control network access to their databases. Many DbaaS providers also enable users to isolate their database instances and to connect to their existing IT infrastructure through an industry-standard VPN. This would make the data transfer process much more secure.

vi. Low monetary cost for operation

Users pay very low rates and only for the resources they actually consume. In addition, DbaaS providers may offer different price options according to the data usage frequency.

3. System Tuning for DbaaS

Databases in many cloud applications, such as those that process large volumes of sales transactions, medical records and scientific data, can be very large. The usefulness of these databases highly depends on how quickly data can be retrieved. Due to the change in tenants' workload patterns, DbaaS usually serves more clients than a single machine or small cluster machine does, the performance of some tenants may degrade, and thus, the performance SLA may be violated. DbaaS providers have to handle such kind of problems in order to give their customers better user experience. Many efforts have been made on how to tune a cloud database in order to re-guarantee the minimum level of performance. Some existing solutions including (1) static and dynamic resource provisioning [9] [10], (2) queuing and scheduling [11], and (3) admission control [12] can be used to solve such a problem. But different approaches have different disadvantages as described below:

i. Static and dynamic resource provisioning

Provisioning is the process of allocating physical computing resources to VM. When the system detects that the pre-defined performance SLAs have a high chance to be violated, more resources such as CPU will be added to improve the situation. The static provisioning will provision a pre-defined amount of resource to the system, while dynamic provisioning will estimate the near future resource demand and provision the corresponding amount of resource to the system. A

major disadvantage of provisioning is that the data center operational cost will increase, especially for static provisioning. The consequence of the improper provisioning might be the negative profit to the service provider.

ii. Queuing and scheduling

Due to the high degree of tenants on cloud databases, sometimes the incoming queries are temporarily held in a queue and then scheduled based on some prioritization criteria, such as worse performance penalty cost. A major disadvantage of this queuing and scheduling method is that this solution only works for short-term load peaks, and some tenants' performance may be heavily degraded due to their queries' postponed execution.

iii. Admission control

Instead of doing queuing and scheduling, DbaaS may have new queries either stalled or rejected when performance SLAs have a high chance to be violated. The major disadvantage of this method is very similar to that of the queuing and scheduling solution, which is in order to guarantee some tenants' performance SLAs, other tenants' performance might have to be sacrificed since their queries were stalled or rejected.

Due to the disadvantages of the above discussed methods, researchers continue searching for new tuning solutions. Data storage improvement through database partitioning is one of them. By partitioning the database tables, the number of disk I/Os, and thus the query response time, can be reduced. Database partitioning is a process that the database service uses to partition the database into smaller partitions so that when a query is processed, only the partitions required by the query need to be transferred from

disk to main memory, leading to a reduction in I/Os, and thus query response time. Database partitioning can also work as an alternative way to tune the cloud database when a performance SLA violation occurs. This method does not need extra resource provisioning from physical machines (PMs) to VMs, hence this method can reduce the data center's operational cost on performance tuning. Database partitioning techniques can be classified into two major categories: horizontal partitioning and vertical partitioning [13], [14]. In horizontal partitioning, tuples are saved in the same disk block according to some specific relation. If users want to fetch all tuples based on that relation, the database service can easily locate those tuples on disk. In vertical partitioning, attributes are grouped together based on how often they are used in a query set. If we reorganize database tables in such a way that each table is partitioned vertically into sub-tables/partitions and the database system, when executing the query, will access only the relevant sub table that contains the attributes in the query, then fewer pages from disk will be accessed to process the query [15], which reduces I/O time, and thus can lead to a better query response time. In many cases, horizontal and vertical partitioning are used together in order to provide better performance, which is called mixed partitioning [14]. In this dissertation, we mainly focus on tuning the DbaaS using vertical partitioning.

4. Issues of Database Partitioning on Tuning DbaaS

Since the first vertical database partitioning algorithm [16] was developed in 1972, many vertical database partitioning algorithms have been proposed. Most of those algorithms are designed for a non-distributed environment; a few of those algorithms are designed for a distributed environment with a single tenant and none is designed for a distributed environment with multiple tenants, which is a cloud environment [1]. The

vertical partitioning algorithms for a non-distributed environment or distributed environment with single tenant, i.e., the traditional vertical partitioning algorithms, are not suitable for a cloud database environment. The reasons are due to the following three major issues that cannot be solved using the traditional vertical partitioning algorithms:

i. How to analyze and estimate the future resource demand and tuning cost for different tuning approaches so that a better tuning decision can be made in order to minimize the operational cost.

   Resource provisioning is a typical tuning method widely used in cloud computing since resource provisioning will eventually solve the performance SLA violation problem. Then why should the DbaaS providers consider a vertical partitioning tuning method? The reason is that, in practice, no provider can ignore the operational cost since that is so related to their profit, and resource provisioning can significantly increase the operational cost to the DbaaS providers. So if a cost-saving performance retuning algorithm, such as vertical partitioning, can re-guarantee the performance SLA, resource provisioning can be avoided. Then an important question is how to analyze and know the costs of vertical partitioning tuning and provisioning tuning. If the algorithm has no ability to estimate the costs for the two tuning methods, then there is no way to find out the correct cost-saving way to tune the cloud database.

ii. How to monitor and manage the sharing buffer pool among multi-tenants so that the overall SLA penalty cost can be minimized.

   Sharing physical resources (CPU, RAM, I/O, and band width) is a considerable issue in clouds [17], [18]. This is because multi-tenancy is one of the key features

of clouds, where a large number of tenants' databases with different SLA requirements are co-located in one environment and share the same physical resources. Many database operations including database partitioning process which are designed to improve a specific tenant's database performance cannot use the shared resources without consideration for other tenants. As we know, physical resource competing in a cloud database might sacrifice some tenants' database performance due to the resource limitation. In practice, different classes of tenants with different levels of SLAs might be co-located in the same VM instance, i.e., those tenants share the same computing resources. The SLA penalty costs may be different for different tenants. If there is no resource sharing monitoring mechanism, then there would be a chance that the SLA penalty cost will get very high once vertical partitioning tuning is applied to a cloud database. In a cloud database, the buffer pool shared by multiple tenants is one of the most important sharing resources, so how to monitor and manage the sharing buffer pool is a challenge to service providers.

iii. How to distribute the resulting partitions to proper nodes in order to provide high performance.

When a database table is partitioned, generally more than one partition will be generated. In a typical database partitioning process, more than one database table is partitioned, i.e., many partitions will be generated. How to deploy those partitions to different nodes is not an easy problem in a multi-tenant cloud environment. In practice, the DbaaS providers will distribute the partitions to different nodes in different availability zones in order to provide high service

availability and durability. However, what are the impacts of the partition distribution on other tenants using the same nodes? Adding a new partition to a node or removing an old partition from a node may change the resource utilization or workload on that node, which then affects the SLAs of all tenants on that node. Therefore, how to select proper nodes, which may be located in the same data center or a different data center, is a critical issue.

5. Contribution

Cloud database tuning to re-guarantee tenants' SLAs is an important process for all DbaaS providers. Because of the complex environment of cloud databases, tenants' SLAs are occasionally violated. Without an efficient database tuning technique, the DbaaS providers will incur high SLA penalty fees and therefore make the profit decreased or even negative. Unfortunately, the typical tuning method used by current DbaaS providers will lead to high data center operational cost. Database partitioning is an alternative method for cloud database performance tuning; but existing database partitioning algorithms can only partition database tables in a single tenant database environment.

In this dissertation, we propose two algorithms that together will provide an efficient algorithm to tune cloud database performance. The first algorithm, SLA-LRU, is a database buffer pool management algorithm. SLA-LRU controls the buffer page replacement by considering the pages' frequency, recency and value. The frequency represents how often a page is referenced; the recency represents how long a page resides in the buffer pool and the value represents how expensive to remove a page from the buffer pool. SLA-LRU can efficiently manage the cloud database's buffer memory and minimize the SLA violation penalty cost when performing the buffer page replacement

process in a resource sharing multi-tenancy environment. To our best knowledge, SLA-LRU is the first practical buffer pool management algorithm which considers different tenants' SLAs with a measurable metric in a cloud database.

The second algorithm, AutoClustC, has the ability to analyze and estimate the operational costs of two tuning methods, resource provisioning and database partitioning, on a cloud database, and select the lower cost tuning method to tune the cloud database in order to re-guarantee the tenants' SLAs. AutoClustC uses the second order Autoregressive (AR) model [19] and Artificial Neural Network (ANN) model [16] to forecast the operation cost of the resource provisioning tuning method and database partitioning tuning method, respectively. If the database partitioning tuning method has a lower cost, a database partitioning algorithm based on the existing partitioning algorithm, AutoClust [20], is triggered to re-partition the database. Finally, the resulting partitions are distributed to proper nodes (PMs) in the same data center based on a "vote" mechanism, which votes for the best PM by computing the weighted overload score for each PM, in order to enhance the performance under an overloaded workload environment. To the best of our knowledge, AutoClustC is the first dynamic tuning algorithm based on database partitioning that is designed for cloud databases.

6. Organization

The rest of the dissertation is organized as follows. Chapter II reviews the existing work related to cloud database performance tuning with a focus on database partitioning. Chapter III describes, SLA-LRU, our proposed technique on cloud database buffer pool management. Chapter IV describes AutoClustC, our proposed technique on cloud database tuning using database partitioning. Chapter V presents the analytical results as

well as the experimental results studying the performance of our techniques. Finally, Chapter VI provides conclusions and future research directions.

Chapter II: Literature Review

In this dissertation, two algorithms, SLA-LRU and AutoClustC, are proposed in order to solve all research issues presented in Section 4 of Chapter I. SLA-LRU is a database buffer management algorithm used to manage cloud database buffer pool based on tenants' SLAs. AutoClustC is a cloud database performance tuning algorithm, which is able to find out the lower cost tuning technique between the two techniques, database partitioning and resource provisioning, and can tune the database when a specific tenant's performance SLA is violated. AutoClustC can reduce service providers' SLA penalty cost by considering tenants' SLAs. In this Chapter, we first review the literature on database buffer pool management and then review the literature on resource provisioning and database partitioning.

1. Database Buffer Management Algorithms

Database as a service in the cloud has attracted a lot of attention during the recent years. Major IT companies have provided large-scale database management services, such as Database.com [21], Google Cloud SQL [22], Windows Azure SQL Database [7], Oracle Database Cloud Service [23] and Amazon RDS [6]. In those multi-tenancy environments, multiple tenants with different performance SLAs share computing resources. Due to the resource overbooking characteristic in a cloud environment, how to manage the resource efficiently, especially the buffer memory, has attracted more and more attention.

When data is retrieved from the database, most recently accessed data will most likely be retrieved from the buffer pool, which serves as a cache of database pages and is crucial

for good performance. We call such data retrieval process *logical I/O*. If data cannot be found in the buffer pool, it has to be retrieved from disk, we call such data retrieval process *physical I/O*. Many existing techniques focus on how to manage the buffer pool efficiently so that the ratio of the logical I/O over the total I/O can be kept at a high level. The most famous one is LRU-K [24], which is used by most cloud databases today. The LRU-K algorithm is an extension of the LRU algorithm [25], which always replaces the least recently used buffer pages. LRU-K introduces a K parameter, which represents the number of times for which a page has to be referenced. Only when K reaches a specific threshold, the corresponding buffer page will be put into the buffer page list. By doing this, LRU-K will consider both the page's recency and frequency rather than just considering the page's recency (such as LRU), hence gives better performance.

Several other existing buffer management algorithms try to partition the buffer pool into two or multiple separate regions for different purposes. DBMIN [26] was proposed to allocate a separate buffer pool for each query. ARC [27] and its clock-based approximation CAR [28] divides the buffer pool into two parts: one region contains frequent pages, the other contains recent pages. The algorithm in [29] developed a multi-buffer framework for saving energy consumption of accessing flash memory. All these algorithms focus on the goal of maximizing performance (i.e., hit ratio) for a given workload. They consider the recency and frequency of a referenced page so as to remove unnecessary pages. However, they have a common weakness when applied to the multi-tenancy cloud environment in that they ignore the value of the referenced page, i.e., those algorithms ignore the cost of the removal of a page. Generally, pages belonging to different tenants have different values. Improper page movement may increase the SLA

penalty cost to the provider. In order to solve this problem, a buffer management algorithm called MT-LRU [30] has recently been proposed. MT-LRU focuses on a multi-tenancy environment. This technique considers the buffer page hit ratio degradation (HRD) as the metric in different tenants' SLAs in order to manage the buffer pool. When a query accesses a buffer page, if it is found in the buffer pool, this access is referred as a hit, otherwise this access is a miss. The Hit Ratio (HR) is defined as $HR = \frac{h}{N}$, where N is the total number of pages accessed and h is the number of pages found in buffer pool. Then $HRD = \max\{0, HR_B - HR_A\}$, where HR_B is the hit ratio when the promised buffer pool is statically reserved for the tenant and HR_A is the hit ratio when the buffer pool is dynamically shared by multiple tenants. Though MT-LRU is the first buffer management algorithm considering the SLA violation issue for cloud databases, this algorithm has the following disadvantages: (1) the HRD is meaningless to the tenants when it works as the key metric in SLA, especially for tenants without corresponding database background, and thus they have no idea how to select the proper level of HRD; (2) in order to know the HRD, every read on a page has to be recorded and the number of page read actions has to be saved, which will add overheads to the algorithm; and (3) as discussed in [31], the biggest disadvantage of using HRD metric is that the HRD may change very little after a long period of database running, especially in the situation when database performance fluctuates significantly. Under such case, the HRD shows little variation, i.e., HRD cannot be used to detect a tenant's SLA violation. That is why other metrics, such as Page Life Expectancy, are suggested to be used with HRD metric [31]. Overall, MT-LRU considers frequency and recency of a page as it is based on LRU-K which considers pages' frequency and recency. It also considers the value of a page, which is

represented using HRD, but using HRD alone as the metric in SLA is less meaningful to tenants and may lead the failure to detect SLA violations when the performance of the database keeps fluctuating. In order to fix the weaknesses of MT-LRU, in this dissertation we propose SLA-LRU, an algorithm to efficiently manage the buffer pool in a multi-tenancy database environment. This algorithm uses tenants' buffer pool level, which is defined as the percentage of the total buffer pool size allocated to tenants, as the metric in the tenants' SLAs. This algorithm is based on LRU-K and considers all three buffer page factors: recency, frequency and value, when performing the page replacement. Value is represented by using a function to compute the penalty cost when a violation of an SLA that is based on buffer pool level occurs. Using the buffer pool level, instead of HRD, as a tenant's SLA metric is more meaningful. A low buffer pool level is a direct sign of an SLA violation. Thus, the system can correctly determine whether a tenant's SLA is violated by measuring the actual buffer pool level used by the tenant.

2. Resource Provisioning Algorithms

Resource provisioning on cloud is the process of allocating computing resource from PM to VM, i.e., it is a process of how to manage the system resources. Traditional Resource Management Systems (RMS) such as Condor [32], Load Leveler [32] and Portable Batch System [33], adopt system-centric resource allocation approaches which focus on optimizing overall system performance. They assume all users are equally important and disregard the actual SLAs of different users. Hence, they are not able to be used in a multi-tenancy environment in which different tenants' requirements are crucial and need to be fulfilled.

One of the main advantages of the cloud computing paradigm is that it simplifies the time-consuming processes of computing resource provisioning, which typically works as a way of re-guaranteeing higher Quality of Service (QoS) level to the tenants when SLAs are violated. The resource provisioning algorithm of a RMS for a multi-tenancy cloud should be able to address the following two major issues, which are not addressed in traditional RMS.

Issue 1:

First, such provisioning algorithm needs to be aware of the actual workload or resource demand patterns which the system has to deal with. Some patterns might be good for static resource provisioning; some patterns might be good for dynamic resource provisioning. In static resource provisioning, historical average resource utilization is computed and used as the amount of resource that is provisioned from PM to VM. After the initial static resource provisioning, the average resource utilization may not be recomputed for a long period of time. In contrast, dynamic resource provisioning is based on shorter timescales, and uses some Machine Learning or Statistic models to estimate the amount of resource provisioned from PM to VM. According to the description in [34], there are typically 4 cloud workloads or resource utilization patterns as shown in Figure 1. In pattern (a) the workload or resource utilization follows an on/off style, i.e., the workload or resource utilization switches from two static values according to different time period. In pattern (d) the workload or resource utilization is totally random, i.e., prediction is impossible using the historical data. So patterns (a) and (d) cannot be benefited from dynamic resource provisioning, and static resource provisioning should be used on these two patterns. In pattern (b) and (c) the workload or resource utilization

follows some style, and the workload or resource utilization has variety according to different time period. So patterns (b) and (c) may be benefited from dynamic resource provisioning, and dynamic resource provisioning could be used on these two patterns. In this section, we focus on the literature of dynamic resource provisioning since we use a dynamic method to forecast the cost of resource provisioning in this dissertation.

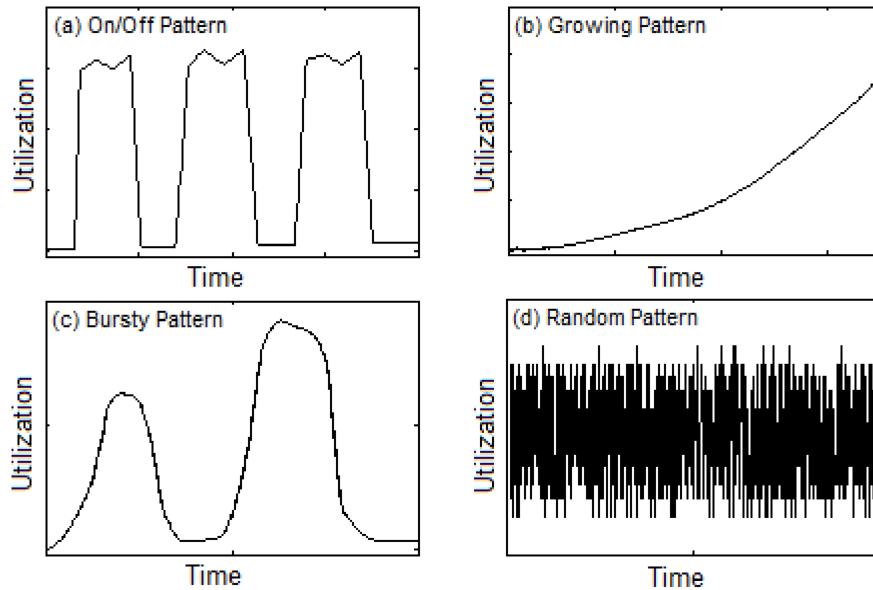


Figure 1. Cloud workload patterns

Issue 2:

Second, the resource provisioning process needs to be cost-effective. When cloud service providers try to maintain the SLAs for different tenants, certainly they can provision huge resources to the specific VM in one time, but the weakness of doing in this way is that it may significantly reduce the service providers' profit. Though the providers have to handle the performance degrading problem, which may be caused by heavy workloads or query pattern changes, they cannot charge extra fees to the users. Computing resource is a major operation cost to service providers, resource provisioning

will definitely increase the data center's operation cost. So resource provisioning algorithm for a multi-tenancy cloud must take the cost into consideration.

Several approaches have been proposed for dynamic resource provisioning. These approaches mainly focus on the perspective of cloud providers. In [35], the authors have presented an approach for dynamic resource provisioning of VM, which is called Sandpiper. The authors present a black-box strategy that is fully OS and application agnostic as well as a gray-box approach that can exploit OS and application level statistics. From those statistics, a unique metric, which is based on the consumption data of the three physical computing resources, CPU, network and memory, can be defined in order to make the provisioning decision. Sandpiper can automate the task of monitoring and detecting hotspots, determine a new resource mapping method from PM to VM, and initiate the necessary migrations in a data center.

The paper [36] proposed a new self-adaptive capacity management framework that includes three models: a two-level SLA driven pricing model which gives rewards for throughput to be within SLA limits and penalty for throughput going above, an analytical queuing based performance model which captures application specific bottlenecks and the parallelism inherent to multi-tier platforms to maximize the provider's business objective, and a complex optimization model. By combining those three models, the resource provisioning algorithm in [36] can significantly reduce the cost in terms of the provider's achieved revenues. But this approach would not be cost-effective in a situation where an application has numerous classes since different class may have different models. Also, according to the studies on the performance and cost in the DbaaS environments in [37], the authors have shown that given the range of the pricing models

and the flexibility of the allocation of resources in cloud-based environments, it is hard for users to figure out their actual monthly cost upfront. So using an inaccurate price model in resource provisioning may lead to a negative profit to the cloud providers.

The authors of paper [38] developed an adaptive resource control system that dynamically adjusts the resource shares to individual tiers in order to meet application-level QoS goals while achieving high resource utilization in the data center. The control system uses classical control theory, which carries out a black-box system modeling approach to overcome the absence of first principle models for complex enterprise applications, and relies on an approximate model which relates performance metric such as response time to the fraction of processor allocated to the VM in order to maintain high resource utilization rate.

Dolly [39] is a resource provisioning system for cloud database, which uses VM cloning technique to spawn database replicas and provision resource for shared-nothing replicated databases in the cloud. This algorithm defines a database provisioning cost model in order to adapt the provisioning policy to the cloud infrastructure specifics and application requirements.

Rogers et al. [40] proposed two approaches, white-box approach and black-box approach, for managing the resource provisioning for cloud databases. The white-box resource provisioning approach uses a finer grained estimation of the expected resource consumption of the workload, which relies on the DBMS optimizer to predict the physical resource (i.e., 1/O, memory, CPU) consumption for each query. The black-box resource provisioning uses coarse-grained profiling to characterize the workload's end-to-end performance across various cloud resources.

Kingfisher [41] is a cost-aware system that provides support for elasticity in the cloud. The algorithm works in two steps: (1) leveraging multiple mechanisms to reduce the time to transition according to new configurations, and (2) optimizing the selection of a virtual server configuration in order to minimize the cost. Kingfisher can significantly decrease the cost of resource provisioning compared to the current cost-unaware approach, but this algorithm does not address any SLA violation related issues.

Recently, several open source VM management platform solutions, such as Eucalyptus [42] and OpenStack [43], have been developed in order to build Infrastructure as a Service (IaaS) clouds. Those solutions are designed to allow third-party extensions through modular software framework. Besides those open source VM management platform solutions, many market-based systems, such as [44] [45], have been proposed to manage allocations of computing resources from PM to VM. However, none of these market-based systems has yet incorporated tenant-driven service management with automatic resource management.

Our research in this dissertation mainly focuses on how to tune the cloud database using a database partitioning technique. But in order to show database partitioning based tuning is more cost-effective than traditional provisioning tuning in some cases, we propose a time-series based cost estimation method for resource provisioning. This method can track the tenant's historical resource (we use CPU as an example) utilization, and use the auto-regressive (AR) model [19] to estimate the near future resource that is allocated from PM to VM once a SLA violation occurs. An AR model can describe a certain time-varying process in which the output variables depend linearly on this process's own previous values and on a stochastic term.

3. Database Partitioning Algorithms

In this section, the literature review is divided into three subsections. Subsection 2.1 discusses the existing work done in database vertical partitioning in non-distributed environments. Subsections 2.2 and 2.3 discuss vertical database partitioning algorithms for distributed environments with a single tenant and multiple tenants, respectively.

3.1. Database Partitioning in Non-Distributed Environments

The first well-known database partitioning algorithm was introduced in 1972 with the name of Bond Energy Algorithm (BEA) [16]. This algorithm is an attribute affinity based algorithm. It uses a two-dimension array to represent the relationship between two different kinds of variables, row variable and column variable. Each column represents one kind of variable and each row represents the other kind of variable. Each element in the array is represented by a numerical value, which usually is an integer, to show the relationship between the row and column variables corresponding to this element. This algorithm permutes rows and columns of the array in order to group elements with similar values together. At the end of the algorithm, elements with similar values are located in the same block in the array and each block can be considered as a partition. When doing permutation on the array, the algorithm needs user's subjective judgment to tell the similarity of elements; so this algorithm is hard to implement without human interpretation. Sometimes blocks may have overlaps and some elements do not belong to any block. It means the partitioning result is not always as good as what people expect.

Later, after the development of BEA, another new important algorithm emerged, which was called Navathe's Vertical Partitioning (NVP) [15]. NVP is also an attribute

affinity matrix based algorithm. This was the first time that a partitioning algorithm considers the frequency of queries and reflects the frequency in the attribute affinity matrix on which partitioning was performed. NVP is an extension and improvement of BEA. This algorithm repeatedly does binary vertical partitioning (BVP) on a larger fragment, which is obtained from the previous BVP, to form two fragments. This process will not stop until the fragment cannot be partitioned further. An evaluation function is used to determine which fragment should be selected and whether it can be partitioned further. This algorithm is only suitable for a small query set because of the $O(2^n)$ time complexity where *n* is the number of times the binary partitioning (which is proportional to the number of queries) is repeated. If fragment overlapping is allowed, the time complexity will be even bigger than that.

Later, after the NVP algorithm, a new algorithm called the Optimal Binary Partitioning algorithm was proposed in [46]. This algorithm uses the branch and bound method [47] to construct a binary tree in which each node represents a query. The left branch of a node represents the attributes being queried by the query that are included in a reasonable cut (a reasonable cut is a binary cut that partitions the attributes into two sets; in these two sets at least one of them is a obtained fragment which is the union of a set of attributes that the query accesses). The right branch of a node represents the remaining attributes. If all attributes of an unassigned query are contained in the fragment of the current node, then this query needs not be considered as the child of the current node. This algorithm focuses on a set of important queries rather than attributes themselves. It does reduce time complexity compared to the NVP algorithm but it does not consider the impact of query frequency, and also its run time still grows exponentially with the number of queries.

Some algorithms use a graph search technique when doing partitioning. The paper [48] is an example which uses a graph theory based clustering technique. The attributes usually queried together are used to form a similarity graph. Vertices of the graph are elements and edges connect elements that have similarity values higher than a predefined threshold. Partitions are the subgraphs with edge connectivity containing more than half the vertices. When this technique is implemented for vertical database partitioning, a vertex represents an attribute and an edge represents how often the two attributes connected by this edge will appear together in the same query. Then the algorithm will traverse the graph and divide the graph into several subgraphs, each of which represents a cluster. This technique considers frequent queries and infrequent queries to be the same and this may lead to inefficient partitioning results. This is because attributes that are usually accessed together in infrequent queries but are not accessed together in frequent queries may be put in the same fragment if all queries are considered to be the same.

A more recent attribute partitioning algorithm was introduced in [49], which uses the idea of performing clustering based on an attributes affinity matrix from [15]. This algorithm starts with a vertex V that satisfies the least degree of reflexivity and then finds a vertex with the maximum degree of symmetry among V's neighbors. Once such a neighbor is found, both V and its neighbor are put into a subset. The neighbor becomes the new V. The process continues to search for neighbors of the most recent V recursively until a cycle is formed or no vertex is left. After that, the fragments will be refined using a hit ratio function. The disadvantage of this technique is similar to the disadvantage of the algorithm proposed in [48] since infrequent queries are treated the same as frequent queries.

Along with the increase of processor's speed and the sophistication of software, database systems become cleverer and more powerful than ever before. Researchers then realized that a database system itself can give a lot of help on physical database design to developers. It gives the researchers a new direction when they are working on physical database design. Such an example is presented in [50], where a new idea of using the query optimizer of a database system for automated physical design was proposed. The authors introduced a cost estimation technique which uses the query optimizer of a database system for physical database design. A query optimizer can gather useful statistic information from system views and perform *what-if* calls [51] to help the database system to make a decision on selection of the best query execution plan among multiple query execution plans without running the query. Some of later database partitioning algorithms used the idea in [50].

The AutoClust algorithm [20] is an example of using a query optimizer to generate partitioning solutions. Since our proposed techniques use AutoClust as the database partitioning technique we shall describe the workflow of AutoClust step by step here. There are five steps in the AutoClust algorithm.

Step 1: An attribute usage matrix is built based on a query set indicating which query accesses which attributes.

Step 2: The closed item sets (CIS) [52] of attributes are mined. An item set is called closed if it has no superset having the same support which is the fraction of queries in a data set where the item set appears as a subset [52]. CIS can tell us which attributes are

accessed frequently by the same query. We want to keep such attributes in the same cluster (partition) together as much as possible.

Step 3: Augmentations to add the primary key of the original database table to each existing closed item set are done to form the augmented closed item set (ACIS) which is a combination of CIS and the primary key. Then duplicate ACIS are removed.

Step 4: An execution tree is generated where each leaf represents a candidate attribute clustering (or vertical database partitioning) solution.

Step 5: The solutions are submitted to the query optimizer of the database system that will process the queries for cost estimation and the solution with the best estimated query cost is chosen as the final vertical database partitioning solution.

The authors of AutoClust algorithm also proposed some ideas of how to extend AutoClust to cluster computers. According to the authors' ideas, multiple partitioning solutions are selected from the candidate partitioning solution pool. These selected partitioning solutions are the best solutions (i.e., the ones that have the best average query estimated costs) and are implemented on the computing nodes in a round robin order. Every future incoming query will be routed to the computing node containing the partition that gives the best estimated query cost for the query execution. AutoClust uses a fixed query set as the algorithm input and mines CIS from that query set to generate multiple partitioning solutions. We call such partitioning algorithm a static or semi-automatic partitioning algorithm. This algorithm runs only once; if users want to do re-partitioning they have to monitor the database performance and trigger AutoClust by themselves. The authors did not present any performance results of their algorithm.

The DYVEP vertical partitioning algorithm was proposed in [53]. DYVEP can partition the database when queries are on the fly, i.e., it is a dynamic vertical partitioning algorithm for database systems. DYVEP monitors queries in order to accumulate relevant statistics for the vertical partitioning process. It analyzes the statistics in order to determine if a new partitioning is necessary; if yes, it triggers a vertical partitioning technique (VPT) to generate a new partitioning solution. The VPT could be any existing VPT that can make use of the available statistics. The algorithm then checks to see if the new partitioning solution is better than the one in place; if yes, then the system reorganizes the database according to the new partitioning solution. This algorithm depends heavily on the VPT being used and the set of rules that it develops to decide when to trigger the VPT. The algorithm does not address how it would take advantage of distributed databases that have partial or full replication so that queries can be directed to nodes that yield the best query costs to execute them. DYVEP is not a new algorithm to be more exact as it cannot work without a VPT algorithm. It just gives a way of how to re-run an existing VPT algorithm automatically.

AutoStore is the first true dynamic vertical partitioning algorithm that is presented in [54]. AutoStore is a self-tuning data store that can free DBAs from monitoring the current workload. This algorithm has the ability to automatically collect queries and partition the data at checkpoint time intervals. When enough queries are collected, the algorithm will update the old attribute affinity matrix and do permutation on this matrix to make the matrix have the best quality (the quality can be calculated using BEA [16]). Then AutoStore will do partitioning on the new matrix and use the greedy method to find out the best way to cluster the attributes in the new matrix based on the estimated cost from

the query optimizer. Once the best partitioning solution is found, the costs of building the new partitions and the estimated benefit brought by the new partitions will be calculated separately. If the benefit is larger, the re-partitioning action will be triggered; otherwise re-partitioning will not be triggered.

AutoStore is the first solution to solve the vertical database partitioning problem with a fully automatic online approach. Unfortunately this algorithm has several problems. The first is that the authors did not give any clue on how many queries (in the article this number is called *CheckpointSize*) we need to collect so that we have enough statistics to permute and partition the attribute affinity matrix. The second problem, which is more serious, is that this algorithm will run re-partitioning checking (i.e., checking to see if re-partitioning is needed) every time the number of queries collected is equal to the *CheckpointSize* no matter what performance trend it has at that time. This means that the re-partitioning checking process will be triggered even when the performance is still good. As we know re-partitioning checking is very expensive and should not be run too often; but AutoStore does re-partitioning checking before checking the performance trend. This is not an inefficient way to do re-partitioning.

Till now we have reviewed many database partitioning algorithms, but they were designed for non-distributed environments, so none of the above database partitioning algorithms has the ability to analyze different SLAs for different tenants. That is why those algorithms cannot be used on cloud databases. In the next subsection, we will review database vertical partitioning algorithms designed for single-tenant distributed environments.

3.2. Database Partitioning in Single-Tenant Distributed Environments

Along with the development of new partitioning algorithms, some work has been done on evaluating the performance of distributed databases on cluster computers. The results show that distributed databases can greatly improve the performance and satisfy business requirements [55]. Because of this, distributed databases have become widely used and important for many applications, which call for more research to find ways to improve their physical database design. In this section, we review existing database partitioning algorithms for single-tenant distributed databases.

A database partitioning algorithm on cluster computers, ElasTras, was introduced in [56]. This algorithm is a database schema level partitioning algorithm. The key idea of database schema level partitioning is that for a large number of database schemas and applications, transactions only access a small number of related rows which can be potentially spread across a number of database tables. ElasTraS takes the root database table of a tree structure database schema as the primary partitioning database table and the other nodes of the tree as the secondary partitioning database table. The primary partitioning database table is partitioned independently of the other database tables using its primary key. Because the primary database table's key is part of the keys of all the secondary database tables, the secondary partitioning database tables are partitioned based on the primary database table's partition key. Then all partitions will be spread across several Owning Transaction Managers, which own one or more partitions and provide transactional guarantees on them. Analyzing a database schema is much more difficult than analyzing a database table and this algorithm is generally configured for static partitioning purposes. Though the authors point out that this algorithm is for cloud

databases, the paper does not address any issues for a multi-tenancy environment. So we still regard this algorithm as a database partitioning algorithm in a single-tenant distributed environment.

In [57], the authors proposed an algorithm called FINDER that aims to find the optimal distribution policy for a set of database tables given a target workload. The assumption of this algorithm is that the workload is given and the future workload should be very similar to the one used by the algorithm. So it is a static algorithm. For a given database table set $T = \{T_1, \ldots, T_t\}$, this algorithm can find the distribution policy $D = \{X_1, \ldots, X_t\}$ where X_i is a set of attributes and Ti is distributed based on X_i. The tuples of a database table will be assigned to different segments according to the hash value of X_i. We can see that this algorithm is used to statically partition the database tables on cluster computers.

The Amossen algorithm [58] is a partitioning algorithm used only on OLTP applications. Generally an OLAP application contains lots of many-row aggregates and likely benefit from parallelizing its queries on multiple sites and exchanging small sub results between the sites after the aggregations. It means that the queries happening on such system are usually very complex. In an OLTP application, on the other hand, there are many short queries with no many-row aggregates or few-row aggregates and the queries only gather all attributes from the same site. It means that the queries happening on such system are usually very simple. In [58] the authors presented a cost model and then used simulated annealing to find the close-to-optimal vertical partitioning with respect to the cost model. In this algorithm, the queries must be very simple and have to avoid breaking single-sitedness. So we can regard this algorithm as a static algorithm.

HACA algorithm was presented in [59]. This algorithm is based on an existing object clustering technique called ant clustering technique, which was first proposed in [60]. As the name described, this existing object clustering algorithm is an unsupervised learning technique based on ants' behaviors. It is used to simulate the ant movement in nature to pick or drop objects so as to cluster objects with similar patterns together. In HACA, a hybrid ant clustering technique is used to partition attributes in a database table according to the transaction patterns. After the partitioning process, attributes with similar transaction patterns will be grouped together and the sum of irrelevant local attribute access cost and relevant remote attribute access cost will be minimized. Irrelevant local attribute access cost represents the cost spent on reading irrelevant attributes from a local partitioning fragment; relevant remote attribute access cost represents the cost on reading relevant attributes from a non-local partitioning fragment. Since the vertical partitioning problem has very high complexity in terms of being NP-Hard [61], the performance of HACA heavily relies on the iterations of the ant clustering technique used in the whole partitioning process. The more iteration the higher performance the partitioning solution will be. But the times of iteration is a user defined parameter; if this number is set to a small value, the partitioning result will lose accuracy and if this number is set to a very large value, the algorithm will take a very long time to run.

Genetic algorithm-based clustering (GAC) was proposed in [62]. This algorithm formulates a database partitioning algorithm into the travelling salesman problem. This algorithm works in 5 steps. In Step 1, some initial solutions are randomly generated. They are considered as the first generation or the parents of next generation solutions. In Step 2, a code is proposed to represent each solution. This code can be either binary or non-

binary. In Step 3, every pair of individuals from the current generation exchange their genetic composition. The offspring inherits some genes from parents during the crossover operation. For each offspring, it still has the same code format as its parents. In step 4, an offspring alters its gene with a very small probability, thus the algorithm can provide a small amount of random search on the offspring. Step 4 is optional in GAC. In Step 5, a fitness function is used to evaluate the fitness of each offspring in order to select the best offspring which will be used as the parent for the next generation. The above 5 steps are performed repeatedly until a certain criterion is met. GAC assumes the transactions or workload profile is known in advance, i.e., GAC uses an existing transaction set to perform vertical partitioning. If the users want to re-partition the database tables they have to re-collect the most recent transactions as the input of the algorithm, which is time-consuming and needs DBA with solid physical database design knowledge to find out useful transactions, which are those transactions with high physical read ratio and high frequency. This algorithm does not separate the logical I/O transactions from the physical I/O transactions. Also this algorithm does not mention anything about how to distribute partitioning results to different nodes and how to perform query routing.

The partitioning algorithms reviewed in this subsection are designed for a distributed database running on cluster computers with a single tenant. The same as the algorithms discussed in Section 3.1, the algorithms discussed in this subsection have no ability to make SLA based partitioning decisions. In recent years, cloud database becomes very popular because of its pay-per-use model. In a cloud database, more than one tenants are competing resources of the same PM. The database partitioning algorithms designed for cluster databases cannot work well any more. That is why some new database partitioning

algorithms are developed for cloud databases. In the next subsection, we review those algorithms.

3.3. Horizontal Database Partitioning in Multiple-Tenants Distributed Environments

Horizontal partitioning has been used on existing commercial cloud based database systems, like SQL Server in Microsoft Azure [63], [64]. Cloud SQL Server composes of five layers: (1) Infrastructure and Deployment Services layer, (2) Distributed Fabric layer, (3) Database Engine layer, (4) Distributed Query Services layer, and (5) Protocol Gateway layer. Partitioning activities and partition management are done by the Distributed Fabric layer. The Distributed Query Services layer is responsible for routing queries to the appropriate partition for single-partition queries and to coordinate queries across partitions for multi-partition queries.

In [65], a relational cloud model is presented, in which a dynamic horizontal partitioning technique is used to scale a single large database to multiple nodes using a workload-aware strategy presented in [66]. The dynamic horizontal partitioning process is done by analyzing query execution traces and identifying tuple groups that are accessed together by individual transactions periodically. The execution trace can be represented as a graph. Each graph node represents a group of tuples. The weight on an edge between two graph nodes whose tuples are accessed by a single transaction reflects the frequency of such pair-wise accesses in a workload. Balanced logical partitions can be found by minimizing the total weight of the cut edges. The output of the partitioning algorithm is a mapping of individual tuples to logical partitions. Then query routing is done by finding a set of predicates on the tuple attributes. The algorithm does not consider the requirements of other tenants when partitioning data for one tenant.

In [67], the authors propose several methods to scale databases from a small number of nodes (p) to a large number of nodes (q). The simplest method does a brute force matching of the current p data partitions to q data partitions under certain partitioning constraints. An advanced method is to pre-partition the data into fine-grain chunks and manage data partitions that are aligned and mapped to the chunks. Data movement is improved by using a hierarchy of pre-computed partitions; however the algorithm is static.

In [68], an elastic and scalable data management system (ElasTras) for the cloud is presented. Unlike other partitioning techniques that are table-level partitioning (either horizontal or vertical), the partitioning technique used by ElasTras is schema-level partitioning. The algorithm proposed in [68] is an improved version of the algorithm presented in [56]. The new algorithm addresses issues for a multi-tenancy environment. It uses a partitioned database design based on the principle of key-value stores, in which an associative array is used as the fundamental data model. In this data array model, data is represented as a collection of key-value pairs where each possible key appears at most once in the collection. This data storage structure allows related data fragments of different database tables to be stored in the same partition. ElasTras has the ability to automatically consolidate the database for small tenants and partition and scale out the database for big tenants, so it can be considered a dynamic schema level partitioning technique.

In [69], the authors modeled the workload as a hypergraph and tried to minimize the number of average query spans after partitioning the data. This algorithm can monitor the change of workload and incrementally repartition the database and place the data to

different nodes in small steps without having to complete the whole repartitioning process. This algorithm can deal with the partition relocation problem when workload changes. Though this algorithm is dynamic, it does not address how resource limitation will impact the partition relocation results and how the results for one tenant will impact other tenants' SLAs.

The algorithms discussed in this subsection are designed for a cloud environment, but they are used for scale out purposes. None of them is vertical partitioning. Besides horizontal partitioning for relational database clouds, there are partitioning algorithms for non-relational databases, such as [70] which performs horizontal partitioning based on data mining for NoSQL databases. While all algorithms except the one in [67] are dynamic, only [65] provides a method for routing queries to appropriate nodes, and none of them handles queries requiring physical I/Os separately from those requiring logical I/Os. Concerning the aspects of distribution and parallel in a cluster computer environment, none of these algorithms provides multiple partitioning solutions or includes a method to distribute partitions to different nodes. In addition, these algorithms do not address the research issues specific to multi-tenancy in clouds, which are discussed in Section 4 of Chapter I, as they do not monitor, model and analyze buffer pool sharing among tenants and their resource utilization to decide on re-partitioning for a specific tenant, and do not consider the impact of new partitions over the SLAs of other tenants. In this dissertation we propose algorithms which aim to address all the research issues associated with vertical database partitioning for multi-tenant cloud databases.

Chapter III: A Proposed Buffer Pool Management Technique for Cloud Databases

When data is retrieved from the database, most recently accessed data will most likely be retrieved from the buffer pool. Generally, such data retrieval is called logical I/O. If data cannot be found in the buffer pool, they will be read from disks first instead of from the buffer pool. Such data retrieval is called physical I/O. In DbaaS, the buffer pool is usually shared by multiple tenants, and the management of the buffer pool directly impacts the query response time. This is because tenants will compete for using the buffer pool, especially as the size of the buffer pool is limited. The operation of one tenant on the buffer pool may impact other tenants, so improper management of the buffer pool will directly increase the query response time for most tenants, which finally leads to a cost increment to DbaaS providers due to the SLA penalty cost.

In order to reduce the SLA penalty cost to DbaaS providers, in this chapter we present an algorithm called SLA-LRU. SLA-LRU is a buffer pool management algorithm which can manage the buffer pool level for different tenants in an efficient way by considering the SLA penalty cost for different tenant. SLA-LRU first computes the possible SLA penalty cost increment for different tenants based on a pre-defined SLA penalty cost function when there is no free memory buffer page left in the buffer pool. Then the algorithm will free the buffer pages with the least removal cost from those least recent used pages in the buffer pool.

Next, we will present SLA-LRU in the following sections of this chapter. In Section 1, we briefly discuss the motivation of our new buffer management algorithm, SLA-LRU. In Section 2 an overview of SLA-LRU will be presented. Then in Section 3 and Section

4 we present our SLA penalty cost function model and the workflow of SLA-LRU respectively. In Section 5 we discuss the overheads of SLA-LRU comparing to the LRU-2 algorithm.

1. Motivation of SLA-LRU

Many modern database systems, such as Oracle, MySQL and Microsoft database products [30], are using LRU-2 or a variant of LRU-2 as their buffer management policy. LRU-2 is a specific version of the LRU-K algorithm (K=2), which was proposed in [24]. The main idea of LRU-2 is to keep track of two lists, the referenced page list and the buffer page list, as shown in Figure 2.

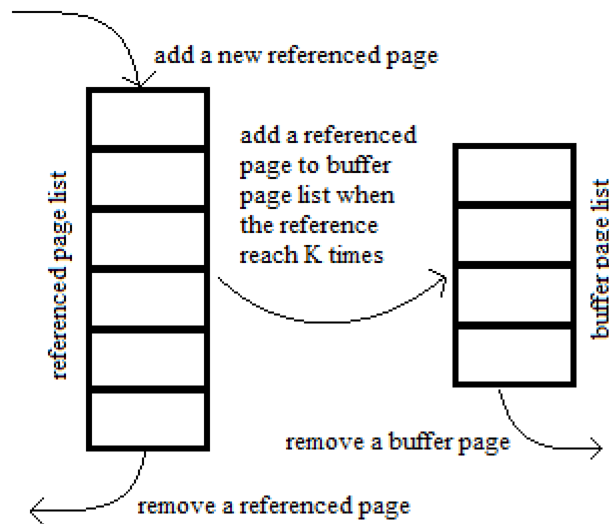


Figure 2. Main idea of LRU-K (K=2)

When a page is referenced for the first time, it will be added to the referenced page list. This page's reference time will be increased by 1 when it is referenced again. When this page's reference time is 2, i.e., the page has been referenced two times, it will be moved to the buffer page list, which is ranked based on the pages' timestamps in the

decreasing order (the oldest page is at the top of the list). When a page replacement occurs in the buffer pool, the page with the oldest timestamp in the buffer page list will be removed from the buffer pool and the buffer page list. We can see thatLRU-2 considers two factors when performing page replacement: page recency and page frequency. By defining the K parameter, LRU-2 considers the page frequency factor; and by ranking the pages in the buffer page list based on the pages' timestamps and removing the oldest timestamp page from the buffer pool, LRU-2 considers the page recency factor. Though LRU-2 performs well in a normal database system, it cannot satisfy the needs in a multi-tenancy database system since not all tenants' page values are equal. A tenant's page value represents the page removal cost. If tenant A's buffer pages' value is bigger than tenant B's buffer pages' value, the page removal for tenant A may have a higher cost than the page removal for tenant B. From the description of LRU-2, we can see that LRU-2 has no ability to handle different buffer pages' values, i.e., LRU-2 cannot guarantee low SLA penalty cost. In order to remove this weakness of LRU-2 in a multi-tenancy environment, we propose our SLA-LRU buffer pool management algorithm.

2. Overview of SLA-LRU

SLA-LRU is based on the classic LRU-2 algorithm, i.e., SLA-LRU considers the page frequency using the K parameter ($K = 2$), and considers the page recency using the page timestamps. The buffer pages in the buffer page list are ranked based on the page timestamp in a decreasing order, so the most recent page will be at the bottom of the list, and the least recent page will be at the top of the list. Then the buffer page list is divided into two portions by a page with timestamp t^* at the α percentile position of the buffer

page list. Only the buffer pages located in the portion in which the page timestamp is earlier than t* will be considered as the least recent pages.

Besides frequency and recency, SLA-LRU also considers the value of a page when releasing a page from the buffer page list. We use the term value to represent the importance of a page, and the value of a page is computed using the SLA penalty cost function which will be discussed in detail in Section 3. SLA-LRU only releases the pages that have the lowest penalty cost increment from the least recent pages. Table 1 presents the list of symbols used by SLA-LRU in this dissertation and the rest of this section discusses SLA-LRU in detail.

Table 1. List of symbols used by SLA-LRU

Symbol	Interpretation
K	The referenced frequency of a page (K equals to 2 by default)
t*	The timestamp of the page at the α percentile position in the buffer page list
α	User defined percentile of the buffer page list
$f(x)$	SLA penalty cost function
$fi(x)$	SLA penalty cost function of tenant i
Δx	The change of the buffer pool level
$\Delta penalty$	The change of the penalty cost

3. Buffer pool level related SLA penalty cost model

Due to the characteristic of multi-tenancy of the cloud database, resources used by the database will be shared among multiple tenants. We take the deployment of Amazon RDS as an example to illustrate the resource sharing architecture of a cloud database. Amazon RDS implements its multi-tenancy architecture by provisioning different EC2 instances to different AWS users. Each user can create multiple, highly varied database

instances on this user's EC2 instance. Those database instances can vary on the storage size and compute resources. Amazon provides different EC2 instance purchase options to its customers. Those options include "On-Demand", "Reserved", "Spot" and "Dedicated" [71]. When an EC2 instance is launched, the user may determine the hardware of the host computer used for this instance type. Each instance type has different CPU, memory, and storage capabilities [72]. So different Amazon RDS database instances may reside either on the same Amazon EC2 instance or on different Amazon EC2 instances. For a specific Amazon RDS instance type, it may have various purchase options or priorities. The pre-defined SLAs with different instance priorities should be different, and DbaaS providers may promise different buffer pool levels in SLA to different tenants with different instance priorities.

The buffer pool in a DBMS is a cache of database pages and plays an important role for good workload performance [30]. When a tenant purchases a database instance, the service provider provisions a specific memory for this database instance. In practice, the provisioned memory may not always be used as the buffer pool for this tenant since the memory may be shared by other tenants. So we propose to use the average buffer pool level in a specific time period of a tenant as the metric in the tenant's SLA, where the buffer pool level is the percentage of the total buffer pool size. For a different tenant priority, this average buffer pool level may be different. A tenant with a higher priority may have a higher average buffer pool level.

The penalty cost function defines how the service provider will be charged when the buffer pool level assigned to a tenant could not meet the promised level. The penalty cost function is a part of the SLA. Often major DbaaS service providers do not expose the

SLA metrics which are directly related to buffer performance, such as buffer hit ratio, to their customers since those metrics may not be meaningful to many customers. However, the metrics, such as the average buffer pool level which is based on the total buffer pool size, are easier to be understood by customers. When tenants sign the SLAs with DbaaS providers, they can choose their desired buffer pool level, just like what they do in choosing the level of the CPU or memory for their database instances.

For a tenant, the penalty cost function quantifies a possible compensation for the performance degradation. For a service provider, the penalty cost function shows the priority of resource allocation across different tenants. A penalty cost function can be either linear or non-linear. If a penalty cost function is non-linear, it can be either step-based or non-step-based. Three possible patterns of the penalty cost function are shown in Figure 3. In pattern (a), the penalty cost increases exponentially when the tenant's actual buffer pool level decreases. In pattern (b), the penalty cost has a linear relationship to the tenant's actual buffer pool level. In pattern (c), the penalty cost has a step-based relationship to the tenant's actual buffer pool level. No matter which penalty cost function is used, the service provider has to refund some portion of the service fee to the tenant if the actual buffer pool level cannot reach the promised level.

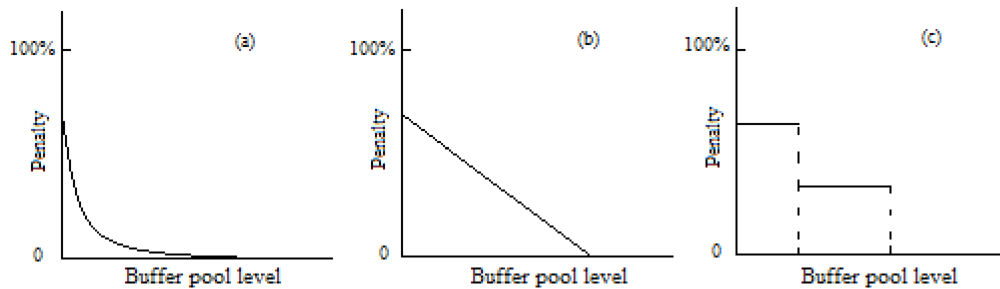


Figure 3. Penalty cost function examples

When two tenants are competing for a computing resource, the service provider can decide the priority of resource allocation by estimating their penalty cost changes. A penalty cost change, $f(x_1) - f(x_2)$, represents the penalty cost difference when a tenant's buffer pool level changes from x_1 to x_2, where $f(x)$ is the penalty cost function. Let us consider an example in which tenant A is using the penalty cost function pattern (a), tenant B is using the penalty cost function pattern (b), and the promised buffer pool level defined in the SLA of each tenant is k. Also, assume that for the current system, there is no free buffer page available, and the actual buffer pool level used by each tenant is k. Now the system is requesting r free buffer pages for either tenant A or tenant B due to the increasing workload. If the system plans to release r buffer pages from tenant A's buffer pool portion, then the penalty cost of releasing r buffer pages from tenant A's buffer pool portion is $f_{aA}(k-r)-f_{aA}(k)$; otherwise, if the system plans to release r buffer pages from tenant B's buffer pool portion then the penalty cost is $f_{bB}(k-r)-f_{bB}(k)$. Both the penalty costs are caused by missing the guarantee of the promised buffer pool levels in the SLAs of the two tenants, and are shown in Figure 4.

If the maximum amount of the penalty fee for buffer SLA violation is the same for both tenant A and tenant B, then from Figure 4 we can see that the penalty cost of releasing the buffer pages from tenant A's buffer pool portion is much smaller than the penalty cost of releasing the buffer pages from tenant B's buffer pool portion. So, the service providers should release the buffer pages from tenant A's buffer pool portion to make free buffer pages for the incoming queries in order to reduce the penalty cost caused by missing the guarantee of the promised buffer pool level.

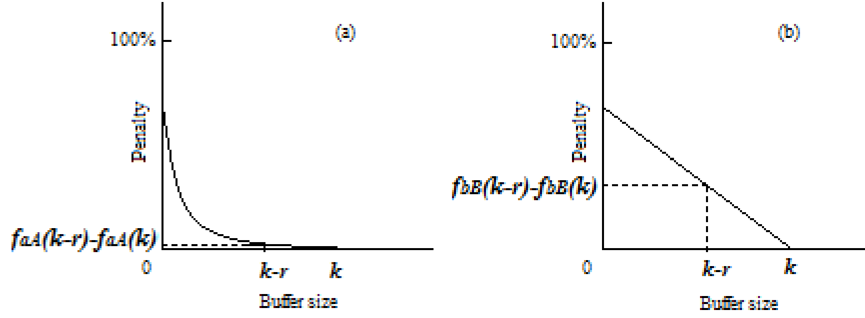


Figure 4. Penalty cost change for tenant A and tenant B with SLA violation penalty function of pattern (a) and pattern (b), respectively

From the above example, we can see that the penalty cost change trends can be used to determine different tenants' buffer allocation priorities when buffer reallocation is required.

Our algorithm can use any pattern of the penalty cost function. In our experiments, we use the step-based pattern (pattern (c) in Figure 3). The reason is that the step-based penalty functions are used widely in both practice and theory. For example, in practice, a step-based vCPU penalty cost function is used for Amazon T2 instance [73]; in theory, a step-based resource penalty cost function is used in [74]. The penalty cost function used for our experiments will be presented in detail in the performance analysis chapter, Chapter V.

4. Workflow of SLA-LRU

In a single-tenancy environment, the system does not have to consider the penalty issue caused by page replacement since all pages belong to a single tenant. However, in a multi-tenancy environment, the penalty cost function for each tenant may be different; so, considering the penalty cost change is necessary for a cloud database when a page replacement occurs. If $fi(x)$ is used to represent the penalty cost function of tenant i, then $fi(x + \Delta x) - fi(x)$ can represent the penalty cost change, $\Delta penalty$, according to the

tenant's buffer pool level change, Δx. In a cloud database, if there is enough virtual memory, $\Delta penalty$ will always be zero because the promised buffer pool level can always be guaranteed for each tenant. However, in practice, in order to save the operational cost of a data center, DbaaS providers usually overbook the resources including the virtual memory. So, the actual buffer pool level used by a tenant is generally below the promised level in the corresponding SLA. In such a case, the $\Delta penalty$ will be bigger than zero. So we have equation (1) when the actual buffer pool level of a tenant i changes by an amount of Δx:

$$\Delta penalty = fi(x + \Delta x) - fi(x) \qquad (1)$$

From equation (1) we can get equation (2) when Δx is not zero:

$$\frac{\Delta penalty}{\Delta x} = \frac{fi(x + \Delta x) - fi(x)}{\Delta x} \qquad (2)$$

If Δx is one unit of buffer pool level change, i.e., we want to measure the SLA penalty cost change trend, then equation (2) can be written as equation (3)

$$\Delta penalty = \frac{fi(x + \Delta x) - fi(x)}{\Delta x} = fi'(x) \qquad (3)$$

Equation (3) shows how we can compute the increment trend of $\Delta penalty$. This Equation means that the derivative of the SLA penalty cost function can be used to measure the penalty change trend. If the penalty cost change trend, $fi'(x)$, is bigger, the penalty cost paid by the service provider due to the SLA violation is lower.

In practice, the buffer pages will not be released one by one, instead, they are released on a "block" or "portion" basis. The buffer page list is divided into two portions by the page with timestamp t* at the α percentile position, which is shown in Figure 5.

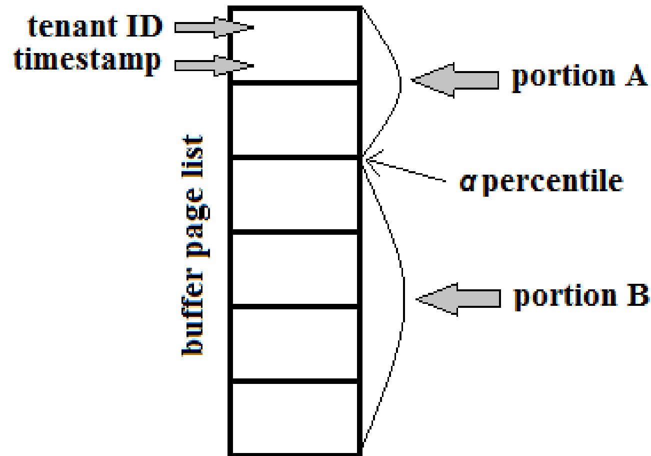


Figure 5. Two portions of a buffer page list

In Figure 5, we can see each page in the buffer page list is associated with a tenant ID and a timestamp. All buffer pages are ranked based on their timestamps. A page with timestamp t* at the α percentile position of the buffer page list is tracked. Then the buffer page list is divided into two portions, A and B, by t*. The timestamps of the pages in portion A are earlier than t* and the timestamps of the pages in portion B are older than t*. When a page replacement using LRU-2 algorithm occurs, all pages in portion A will be released. Generally a buffer page list contains the buffer pages of different tenants. For the pages of the same tenant, they have the same value since those pages belong to the same tenant who has a fixed SLA penalty cost function. When SLA-LRU analyzes the SLA penalty cost function for each tenant, the tenant with the biggest $fi'(x)$ will be marked (if there are more than one such tenant found, all of them will be marked). Comparing with other tenants, removing the pages of such tenant will give the service provider less penalty cost increment. Then among those pages that belong to the marked tenant, only the pages whose timestamps are older than t* will be removed.

By considering both the page age and page removal cost, the buffer pool sharing problem can be solved. Comparing with LRU-2, SLA-LRU has an additional process, which is the SLA penalty cost analyzing process, when performing buffer page replacements. Adding a new process to the algorithm certainly introduces overheads; so, we propose a moving-forward scanning method to reduce such overhead. This method will increase the next scanning length of the buffer page list when the current scanning cannot free enough buffer space, i.e., the position of α will be moved forward for the next scanning. This scanning method works in the following steps:

Step 1: During the current buffer page list scanning process, the pages whose corresponding tenants have the biggest $fi'(x)$and the pages whose corresponding tenants ' actual buffer pool levels are bigger than the promised ones will be removed from the buffer page list and buffer pool.

Step 2: If the buffer pages released by the current buffer page list scanning process are not enough for the incoming queries, the next buffer page list scanning process will double the scanning length, i.e., the size of portion A will be doubled.

Step 3: If the new scanning length is bigger than the buffer page list size, the scanning length will be set as the buffer page list size.

Step 4: Restart Step 1 until enough buffer pages are released.

The moving-forward scanning method can speed up the page replacement process by increasing the scanning length during different scanning phases. This will significantly reduce the total buffer page list scanning times and reduce the overhead incurred by the SLA-LRU algorithm. In the next subsection, we provide more detailed discussions comparing the overheads of SLA-LRU and LRU-2.

5. Overheads of SLA-LRU comparing with LRU-2

We discuss the overheads from two major factors: CPU and memory. First we investigate how the SLA-LRU algorithm impacts the memory. In order to analyze the SLA penalty costs for different tenants, a penalty cost function table has to be stored in the main memory. Every time when a new tenant is signed up, this tenant will be added to the corresponding category in the penalty cost function table. This table, of course, will occupy additional memory space when the database system is running. However, comparing with the buffer pool, which is also saved in the main memory, the penalty cost function table is much smaller. If a tenant's page size is 8 KB, then a 512 MB promised portion of the buffer pool can hold around 65,000 buffer pages for this tenant. But in the penalty cost function table, only the tenant ID and the corresponding promised buffer pool level are saved in the table. So SLA-LRU will not add much more memory overhead comparing with LRU-2.

Second, we investigate how the SLA-LRU algorithm impacts the CPU. In LRU-2, the algorithm will remove all buffer pages in portion A, which is decided by the parameter α, of the buffer page list. Different database systems may use different α (for example, in MySQL, α is 0.1 [75]). In SLA-LRU, the algorithm will remove the pages with the biggest $fi'(x)$, i.e., fewer buffer pages will be released during a full scan of the buffer page list comparing with LRU-2. So in order to get enough free buffer space, SLA-LRU may need more scanning times on the buffer page list than LRU-2, which will incur a CPU overhead for the algorithm. The buffer page list is saved in the main memory, by scanning the buffer page list in a moving-forward way, the overall time complexity of SLA-LRU is

still O(n), where n is the size of the buffer page list, which is the same as that in LRU-2. So SLA-LRU will not add much more CPU overhead compared with LRU-2.

Till now we can see that SLA-LRU includes two major phases: 1) analyzing the SLA violation penalty cost changes for different tenants; and 2) removing the corresponding buffer pages from the buffer pool to provide free buffer pages. The algorithms of the first and second phases are named as Algorithm 1 and Algorithm 2, respectively, and discussed below.

Algorithm 1 (shown in Figure 6): First, in order to let the storage engine know which tenants' buffer pages will be released, two temporary parameters, T and *penalty_change*, are initialized (Lines 1-2). T is a set used to save the tenant ID whose buffer pages will be removed from the buffer pool. At the beginning of the algorithm, no tenants are selected so T is set to be empty at start. Parameter *penalty_change* is used to save the penalty change trend for the tenant i, which is the derivative of a tenant's penalty cost function. We always want to find out the tenant with the least penalty cost increment (such tenant has the biggest derivative of the penalty cost function), so at the beginning of the algorithm, *penalty_change* is set to be the minimal integer value. Then the tenant with the maximal derivative of the SLA penalty cost function for the current buffer pool level will be identified (Lines 2-8). If more than one tenants are found, all of them will be added to T (Lines 9-12), then all tenants that have the maximal derivative of the SLA penalty cost function for the current buffer pool level will be returned (Line 13).

Algorithm 1	Analyze SLA penalty cost
Input:	SLA penalty cost function of each tenant ($penalty_i$)
Output:	a set containing tenants with the biggest derivative of $penalty_i$ (T)
1:	initialize an empty set T //T saves the tenants ID with the least penalty cost increment

2:	penalty_change = MIN_VALUE // MIN_VALUE represents the smallest
	// integer value a 32-bit integer can hold

3:	for each tenant i
4:	$d_penalty_i$ = derivative of $penalty_i$
5:	if $d_penalty_i$ > penalty_change // find out the smallest penalty cost increment
6:	penalty_change = $d_penalty_i$
7:	empty T
8:	add tenant i to T
9:	else if $d_penalty_i$ = penalty_change //more than one tenant has the smallest // penalty cost increment
10:	add tenant i to T
11:	end if
12:	end for
13:	return T

Figure 6. Algorithm of analyzing SLA violation penalty cost

Algorithm 2 (shown in Figure 7): First, two temporary parameters, iteration and *scan_distance*, are initialized (Lines 1-2). Iteration is used to save the current scanning times and *scan_distance* is used to save the current scanning length of the buffer page list. At the beginning of the algorithm, iteration is set to 1 and *scan_distance* is set to 1/10 of the length of the buffer list (the default value of *scan_distance* for MySQL is 1/10 of the length of the buffer list; we adopt this value for our algorithm). Next, the pointer is moved to the top of the buffer page list (Line 4).and the buffer page list is locked from being simultaneously accessed by other queries (Line 5). The buffer page is then moved from the buffer page list and the buffer pool if this buffer page is not dirty and satisfies either of the two conditions: 1) this page belongs to the tenants resulted from Algorithm 1; and 2) the buffer pool level of the corresponding tenant of this buffer page is still bigger than the buffer pool level promised in the SLA (Lines 6-10). After this, the pointer is moved to the next buffer page in the buffer page list (Lines 11-13). The buffer page list is then unlocked so that other queries can access the list (Line 14). The scanning length

of the buffer page list is increased if more free buffer pages are needed after the previous buffer page list scanning process. In order to increase the scanning speed, the scanning length on the buffer page list will increase with a speed of the exponential of 2, i.e., the next scanning length is the double size of the previous scanning length until the whole buffer page list can be scanned. This will reduce the program running time greatly, and thus will improve the query response time (Lines 15-19).

```
Algorithm 2 Release buffer pages
Input:      Buffer page list (buffer_list)
1:          iteration = 1
2:          scan_distance = buffer_list/10 // 1/10 of the length of the buffer page list
3:          while not enough free buffer space
4:            bpage = oldest page of buffer_list //scan from the top of the buffer page list
5:            set mutex // lock the buffer page list from being simultaneously accessed
6:              while scan_distance > 0
7:                if bpage is in T OR buffer utilization of bpage.tenant > promised buffer
            utilization
8:                  if bpage is not dirty // data of the page was not modified
8:                    release bpage
9:                  end if
10:               end if
11:               bpage = bpage.next_page
12:               scan_distance = scan_distance - 1
13:             end while
14:           exit mutex
15:           scan_distance = scan_distance * 2 // move a forward
16:           if scan_distance > buffer_list.size
17              scan_distance = buffer_list.size
18:           end if
19:         end  while
```

Figure 7. Algorithm of releasing buffer pages

Chapter IV A Proposed Performance Tuning Algorithm Based on Database Partitioning for Cloud Databases

In this chapter, we present our AutoClustC algorithm. AutoClustC is a performance tuning algorithm based on database partitioning for DbaaS. This algorithm considers the costs of resource provisioning and database partitioning to select the lower cost method to tune the DbaaS. If database partitioning is selected. AutoClustC will use data mining to partition the database and distribute the resulting partitions to proper PMs located in the same data center. Proper PMs are those that have the least overload scores which are computed using both the PMs' communication cost and resource overload status.

This chapter is divided into three sections. In Section 1, the technique of how to forecast the cost for resource provisioning is presented. In Section 2, the technique of how to forecast the cost for database partitioning is presented. In Section 3, the technique of how to distribute the resulted partitions to proper PMs is presented.

1. Cost Forecasting for Resource Provisioning

Resource provisioning is a typical solution to fulfill the tenants' requirements and guarantee the QoS in a virtual environment [18] [19]. Resource provisioning can eventually guarantee all tenants' performance but it will cause the extra operation cost to the service provider. So, an alternative tuning method, such as database re-partitioning, should be used if it incurs a lower cost than resource provisioning. To make this decision, we will need to know the costs of resource partitioning and database partitioning. In this section, we present a method to estimate the cost of resource provisioning using time series analysis. This section includes two subsections: Subsection 1.1 presents the characteristics of CPU utilization patterns that can be benefited from dynamic resource

provisioning and Subsection 1.2 discusses how to use statistic models to forecast the resource provisioning cost.

1.1. CPU utilization patterns for dynamic provisioning

In Section 2 of Chapter II, 4 typical resource utilization patterns are discussed. In pattern (a), the amount of resource utilization mainly switch between two fixed values, i.e., there is no strong utilization variability in pattern (a). In pattern (d), the amount of resource utilization changes extremely randomly, i.e., pattern (d) features a poor autocorrelation, which is a mathematical representation of the degree of similarity between a given time series and a lagged version of itself over successive time intervals [76]. That is why dynamic forecasting cannot be used on neither pattern (a) nor pattern (d). Instead, for those two patterns, static forecasting should be used. In [18] the authors conclude that only the utilization behavior that is characterized by strong utilization variability and good autocorrelation associated with this periodic behavior, can be benefited greatly by using the dynamic resource provisioning. According to this conclusion, we can see the pattern (b) and (c) discussed in Section 2 of chapter 2 are good choices for dynamic forecasting. We therefore assume the CPU utilization behavior in our research should satisfy this conclusion, i.e., the provisioning cost forecasting algorithm developed in this dissertation can be used on both pattern (b) and pattern (c) if the two patterns follow a periodical behavior.

1.2. Problem modeling of cost forecasting for dynamic resource provisioning

In order to understand the forecasting process more clearly, consider an example shown in Figure 8, which shows a snapshot of 24 hours CPU utilization demand historical data (U) with the demand probability density function (PDF) $u(x)$. If the current time

point is t_0, and the prediction time interval is t, the forecasting process is to compute the average CPU demand, which is denoted by U_{t_0+t} in the next prediction time interval. If the prediction error is E_t, the forecasting result will be $U_{t_0+t} + E_t$. In [18], the authors use an AR model to compute the gain, which is the ratio of the estimated future CPU demand of dynamic resource provisioning to the estimated future CPU demand of static resource provisioning. In our algorithm, we also use an AR model, but the exact estimated future CPU demand is computed so that we can compare the costs of resource provisioning and database partitioning.

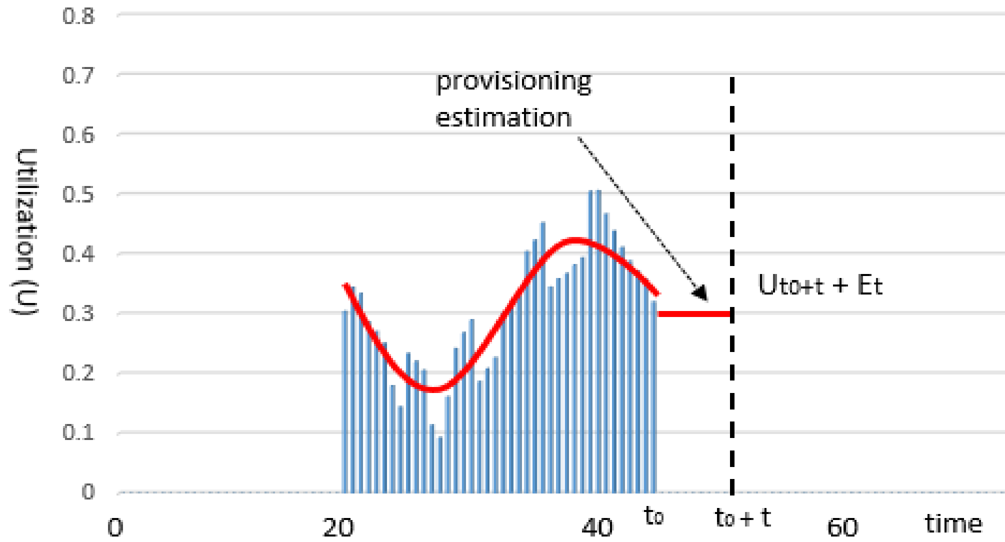


Figure 8. Dynamic forecasting estimation

If we use $U_t(x)$ to represent the demand probability density function of the predicted time series, the exact average future demand forecast can be represented as

$$U_{t_0+t} + E_t = \int_0^{\infty} (x + E_t) \times u_t(x)dx \qquad (4)$$

The expression $(x + E_t)$ represents a given CPU resource allocation, which will be weighted with $u_t(x)$, the probability of a particular provisioned CPU amount x.

Equation 4 can be rewritten as

$$U_{t_0+t} + E_t = \int_0^{\infty} x \times u_t(x)dx + E_t \qquad (5)$$

Equation 5 can be approximated using the following formula:

$$U_{t_0+t} + E_t \approx \int_0^{\infty} x \times u(x)dx + E_t = E[U] + E_t \ (6)$$

where *E[U]* is the statistical mean of the measured historical CPU demand. Now the only undetermined parameter in Equation 6 is E_t, which will be computed based on [19] using Equations (7), (8) and (9).

Based on the assumption of our algorithm, the CPU demand is characterized by periodic behaviors. So for any time series T, the CPU demand in T can be represented as

$$U_T = D_T + U_T^R \qquad (7)$$

where D_T is the trend sub-utilization of the CPU demand and U_T^R is the residual sub-utilization of the CPU demand as shown in Figure 9. Figure 9 shows a very small portion of CPU demand from Figure 8, and this small partition has been zoomed in in order to see the details. The trend sub-utilization is highlighted in blue and the residual sub-utilization is highlighted in red. The trend sub-utilization is deterministic due to the known pattern, i.e., the CPU utilization function is known; and the residual sub-utilization is random and needs to be estimated using some statistical model.

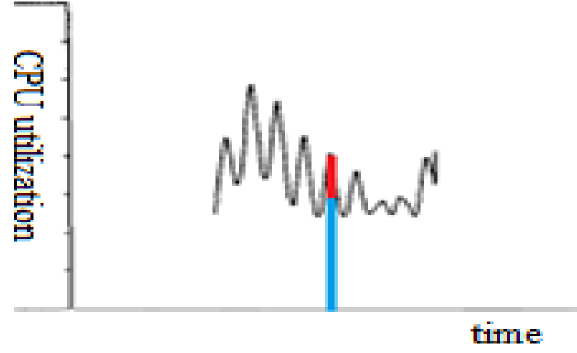


Figure 9. The trend sub-utilization and residual sub-utilization

As stated in [19], the second order AR model (AR(2)) is a simple and effective method in time series modelling, so we use the AR(2) to estimate the residual sub-utilization for our case. The AR(2) model can be represented as

$$U_T^R = \alpha_1 U_{T-1}^R + \alpha_2 U_{T-2}^R + \epsilon_T \qquad (8)$$

where α_1 and α_2 are two AR(2) parameters that are estimated from the historical data, and ϵ_T is the error term, which is assumed to be an independently and identically distributed Gaussian random variable with a mean of zero and a variance of σ_ϵ^2. In order to study the accuracy of an n step prediction, a characteristic function of the AR model is defined as

$$G(j) = \frac{\gamma_1^{j+1} - \gamma_2^{j+1}}{\gamma_1 - \gamma_2} \qquad (9)$$

where γ_1 and γ_2 are the roots of the equation $1 - \alpha_1 B - \alpha_2 B^2 = 0$. Then the n step prediction error is represented using the Gaussian variable having mean zero and variance $\sigma_e^2(n) = \sum_{j=0}^{n-1} G^2(j)\sigma_\epsilon^2$. Here, σ_ϵ^2 is the error variance of one-step prediction.

Once the future CPU demand, *CPU_Demand*, is estimated, it can be used as the argument in the cost function C(*CPU_Demand*) to get the exact operational cost for

resource provisioning, where the C function describes the relationship between CPU time and money spent. In the next step, we need to forecast the cost of database partitioning. In Section 2, a novel forecasting technique will be proposed to predict the cost of database partitioning.

2. Cost Forecasting for Database Partitioning

As we discussed earlier in Section 1 of this chapter, resource provisioning is the simplest way to fulfill the QoS for different customers; but this way generally needs the cloud service providers to reallocate more computing resources. In order to maximize the profit for the service providers, we compare resource provisioning with another performance tuning method, database partitioning. The tuning method with a lower cost always works as the practical performance tuning method on cloud databases. In this section, we first discuss the factors that can impact the cost of database partitioning in Subsection 2.1; then in Subsection 2.2 we present how to use the ANN model [77] to solve the forecasting problem for database partitioning.

2.1. Factors impacting database partitioning

There are many factors that can impact the CPU time spent on figuring out a suitable partitioning solution for a particular database table. The partitioning method used in our algorithm is based on the Closed Item Sets (CIS) mining [52]. The original algorithm, called AutoClust, was first published in [20]. We can conclude 3 major factors that may heavily impact the CPU utilization spent on the partitioning process. The first factor is the size of the database, S. In AutoClust, a query optimizer is used to estimate the cost of each partitioning solution, i.e., partitions will be temporarily physically created in order to let the query optimizer compute the cost. If the database size is large, the partition

creation process will require a high CPU cost to be finished. The second factor is the number of attributes in a database table, *NA* (if the partitioning process covers more than one table, the maximum number of attributes among those tables will be used). When mining the CIS from the query set, frequent CIS have to be generated. If there are *NA* attributes, the possible number of attribute sets would be 2NA. Then each attribute set has to be compared with the attribute set accessed by each query in order to find out the CIS. There are many algorithms such as [78] [79] to prune the number of possible attribute sets, *NA*; but *NA* is still another factor that will impact the CPU cost on partitioning. Finally, the third factor is the number of query types, *NQ*. From the second factor, we already know each query type will be scanned in order to tell whether an attribute set is CIS. So *NQ* is a factor that has to be considered. Besides the above three factors, one more factor has to be added from the aspect of multi-tenancy, which is the number of users of the DbaaS, *NU*. In DbaaS, common physical resources are shared by multiple tenants. A major consequence of such an environment is that the multiple tenants will compete for the resource of the same VM, which will delay the partition creation process. Hence the degree of multi-tenancy becomes an additional factor.

2.2. Artificial Neural Network on cost prediction for database partitioning

A cloud database is a complex system and the relationship between the partitioning cost and *S, NA, NQ* and *NU* is highly non-linear. Because of these characteristics, we propose to use ANN [77] to forecast the partitioning process cost since ANN performs well on complex systems that are intrinsically non-linear in nature [80]. In our ANN model, the inputs are *S, NA, NQ* and *NU*. One hidden layer with 8 neural nodes (twice the size of the input) is used between the input and output layers. The control architecture is

a feed forward back propagation network. The activation function used is the sigmoid function, which is a transfer function used to calculate a layer's output from its net input, for all the inner nodes. This function can give the neural network the ability to learn and generate an output for which it is not trained. However, in order to make the ANN work properly, a well-defined training dataset is necessary. The whole ANN works in two phases: (1) the network is trained using the data provided by users; and (2) the new input is fed to the network and the network produces a desired output that is most appropriate for the given input. Since it is important to choose a proper training function and learning function, the TRAINGDX [81] and LEARNGDM [82] functions can be used in the network. The TRAINGDX function is a network training function that updates weight and bias values according to gradient descent momentum and an adaptive learning rate. The LEARNGDM function calculates the weight change for a given neuron from the neuron's input and error, the weight, learning rate, and momentum constant, according to gradient descent with momentum. The structure of the whole network is shown in Figure 10 where W represents the weight of a node and B represents the bias of a node. There are four different types of input (*S, NA, NQ* and *NU*) and one type of output (CPU time), so we have 4 nodes in input and 1 node in output. We use twice the size of the input types as the number of nodes in the hidden layer, so we have 8 nodes in the hidden layer.

Once the ANN model is constructed using the training dataset, the CPU time spent on the partitioning process can be predicted by sending the current values of the system parameters, which are *S, NA, NQ* and *NU*, to the ANN model. Then the money spent on performing attribute partitioning tuning and resource provisioning tuning can be calculated by using the C(*CPU_Demand*) function where the CPU time estimated from

each of the two forecasting partitioning and resource provisioning processes will be the argument of the C function. The method that yields the lower monetary cost will be selected to improve the system performance. The whole performance tuning cost analysis algorithm is shown in Figure 11.

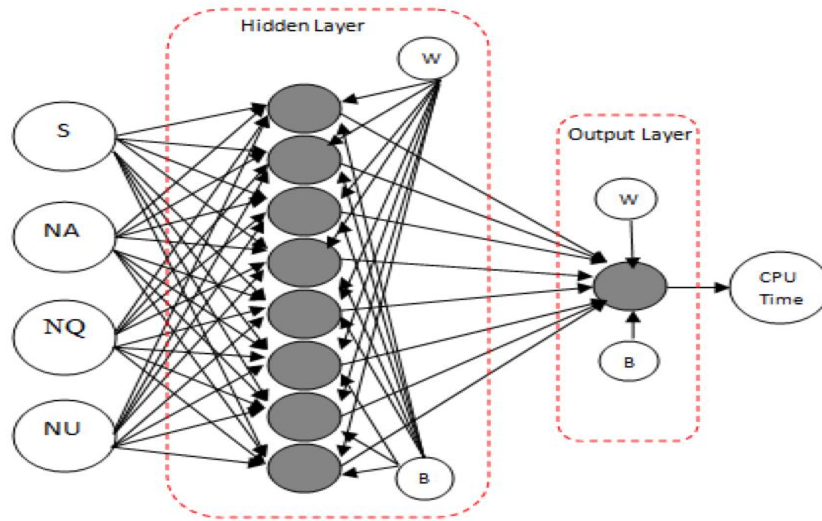
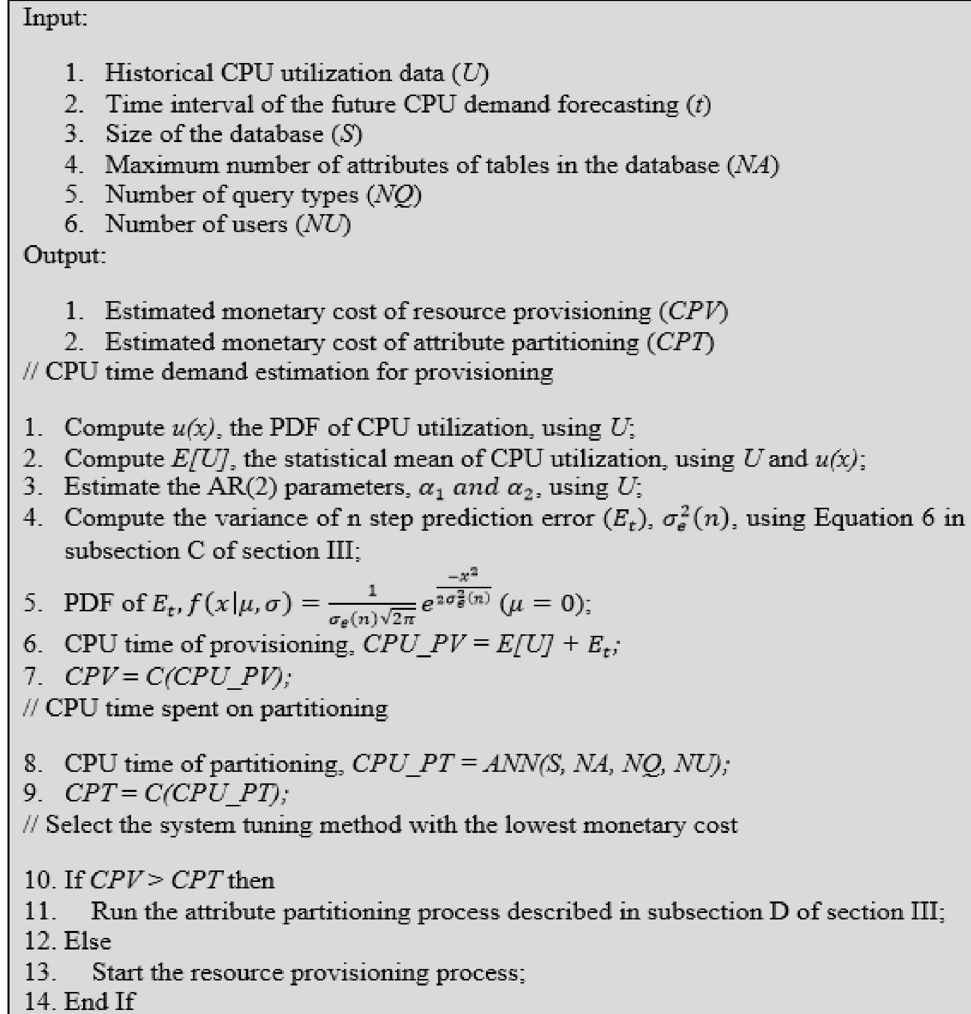


Figure 10. The ANN model used for partitioning cost forecasting



Input:

1. Historical CPU utilization data (*U*)
2. Time interval of the future CPU demand forecasting (*t*)
3. Size of the database (*S*)
4. Maximum number of attributes of tables in the database (*NA*)
5. Number of query types (*NQ*)
6. Number of users (*NU*)

Output:

1. Estimated monetary cost of resource provisioning (*CPV*)
2. Estimated monetary cost of attribute partitioning (*CPT*)

// CPU time demand estimation for provisioning

1. Compute *u(x)*, the PDF of CPU utilization, using *U*;
2. Compute *E[U]*, the statistical mean of CPU utilization, using *U* and *u(x)*;
3. Estimate the AR(2) parameters, α_1 *and* α_2, using *U*;
4. Compute the variance of n step prediction error (E_t), $\sigma_e^2(n)$, using Equation 6 in subsection C of section III;
5. PDF of $E_t, f(x|\mu,\sigma) = \frac{1}{\sigma_e(n)\sqrt{2\pi}} e^{\frac{-x^2}{2\sigma_e^2(n)}}$ ($\mu = 0$);
6. CPU time of provisioning, *CPU_PV = E[U]* + E_t;
7. *CPV = C(CPU_PV);*

// CPU time spent on partitioning

8. CPU time of partitioning, *CPU_PT = ANN(S, NA, NQ, NU);*
9. *CPT = C(CPU_PT);*

// Select the system tuning method with the lowest monetary cost

10. If *CPV > CPT* then
11. Run the attribute partitioning process described in subsection D of section III;
12. Else
13. Start the resource provisioning process;
14. End If

Figure 11. Performance tuning cost analysis in AutoClustC

3. Partition Distribution

After a database table is partitioned, generally more than one partition will be generated. In a typical database partitioning process, more than one database table is partitioned, and thus many partitions will be generated. How to deploy those partitions to different PM is not an easy problem. In this section we present a partition distribution algorithm to distribute the resulting partitions to PM. This algorithm considers both the communication cost between different PM groups and the resource overload status of each PM in order to find out the best PM to distribute partitions. We assume for each

standby database instance, it has a full copy of the data of the master database instance. In the Subsection 3.1, we first review the structure of a typical data center and discuss the motivation of why partition distribution is important for DbaaS. In the Subsection 3.2, we present how the distribution algorithm compute the resource overload score for each PM group, and how it compute the SLA violation probability for each PM in the PM group with the least overload score in order to find out the best PM to distribute partitions. Then, in the Subsection 3.3, we present how the partition distribution algorithm use the communication cost as the weight to adjust the overload score of each PM group if the communication cost is too high to be ignored.

3.1. Motivation of partition distribution for DbaaS

Before we discuss the motivation, first we will take a close look at the physical structure of a typical cloud. We take AWS RDS [6] as our example since it is the major cloud service provider. For AWS RDS, it can hold tenants' database instances in multiple locations world-wide. These locations are composed of regions and availability zones (AZ). Each region is a separate geographic area. Each region has multiple, isolated locations known as availability zones. Each availability zone has multiple data centers. The overall physical structure is shown in Figure 12 [83], where we can see that AWS has many regions across the nation. If we take the North Virginia region as an example, we can see there are 5 AZs in this region. Those 5 AZs are connected to transit centers using redundant paths. A transit center is used to connect different regions together. The communication delay between different AZs in the same region is less than 1 millisecond. In an AZ, there are multiple data centers which are connected by a high speed network, for which the delay between two data centers is less than 0.25 millisecond [83].

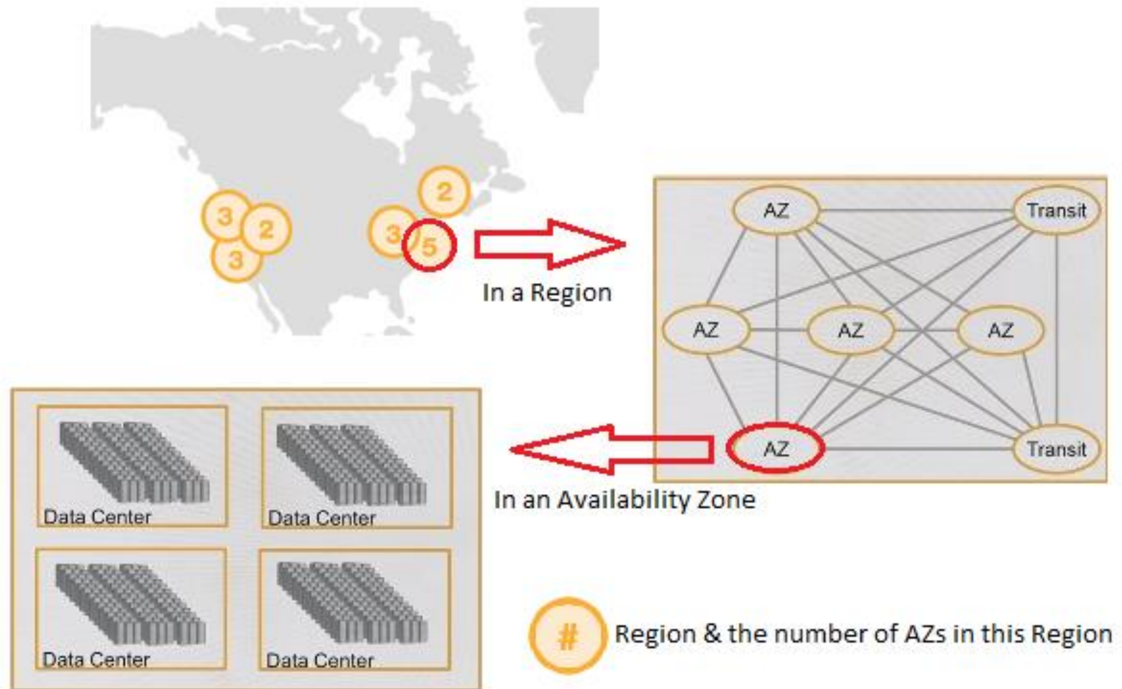


Figure 12. AWS cloud structure

For a cloud database, high availability of data is a very important feature. Generally speaking, high availability refers to a system that is continuously operational for a desirably long length of time. This availability can be measured nearly to 100% operational or never failing. A public cloud provider often puts their availability metric in their SLAs. In order to guarantee high availability, tenants' data must be duplicated in the same data center or across multiple data centers. If the PM which holds a tenant's data fails, other PMs that contain the backup data can take over the failed PM's job and keep the tenant's application running. Those PMs are generally called standby PMs. For our case, when the partitioning process is done, multiple partitioning solutions may be generated, each of which contains multiple partitions. We may distribute different partitioning solutions to different standby PMs so that each standby PM will contain a full copy of tenants' data and each copy is partitioned in a different way. When the PM

containing the master database instance fails, other standby PMs in which the backup database instances exist can continue processing the incoming queries. So the high availability of the cloud database can be guaranteed. This is the first motivation of our partition distribution algorithm.

In a cloud database, computing resources are shared by multiple tenants. Therefore DbaaS providers are facing more unpredictable workloads. This makes the SLAs guarantee work more difficult, especially during the workload peak time. When the pre-defined performance SLAs are violated under some specific circumstances, DbaaS providers need to either refund some portion of the service fee to the tenants or find out a way to re-guarantee the performance SLAs as soon as possible. If the performance SLA violation is caused by the overloaded workload, one of the easiest way of performing the SLA re-guarantee work is to use the database instance that contains the duplicated data located on the standby PMs to process some portion of the queries, so that the workload on the master database instance can be reduced, hence the performance SLA can be re-guaranteed. So this is the second motivation of our partition distribution algorithm.

Partition distribution can occur on different levels: the data center level, availability zone level, and region level. In this dissertation we only consider the case for which partitions are distributed in the same data center, i.e., partitions are distributed on the data center level.

3.2. Computing load balance using the overload score

In modern data centers, virtualization technologies can enable application running and data saving to be hosted inside VMs. Those VMs utilize the physical resource of the PMs in which they are hosted. An important characteristic of a well-managed data center is

whether the data center has the ability to avoid overloaded PMs. Overloaded PMs often lead to performance degrading and are easy to fail. When database partitions are distributed in the same data center, the PM that accepts the distributed partitions has to provision the physical resource for a new database instance. If this PM is already almost overloaded, the performance SLAs of the tenants that are located on this PM will be violated. This may even cause the PM to fail. This is why intelligent partition distribution is so important. However, deciding which PM to be selected to host the distributed partitions is a challenging task. In this section, we present our PM selection method that is based on PM's overload score. Before we discuss our algorithm, first we need to understand how PMs are connected in a data center.

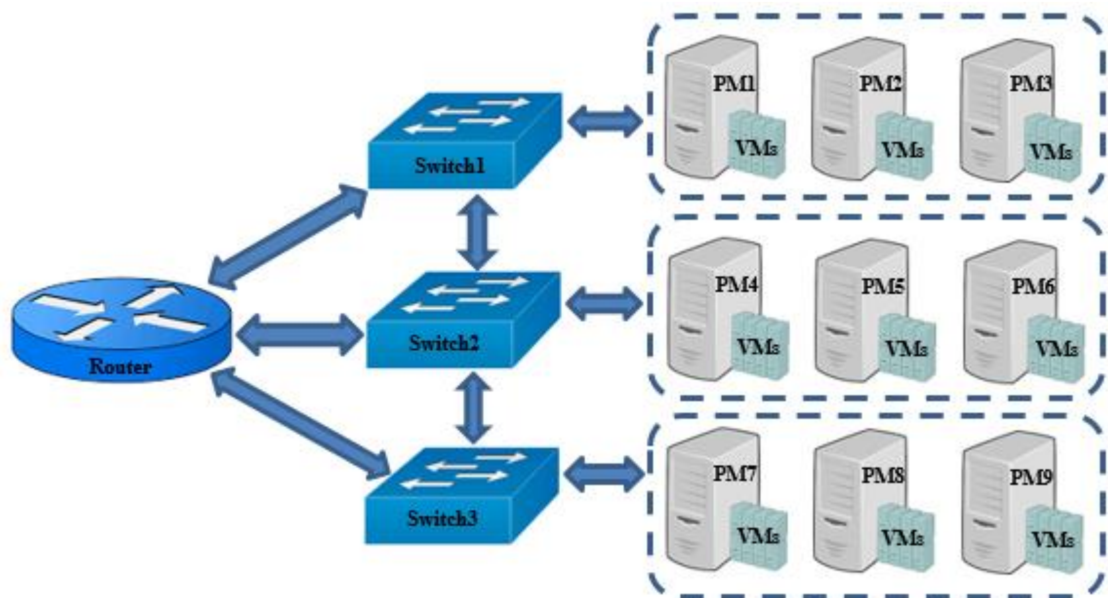


Figure 13. PMs and VMs in a data center

A small part of a typical data center is represented in Figure 13. A data center may consist thousands of PMs which are connected by switches and routers. PMs connected to the same switch form a PM group. For example, PM1, PM2 and PM3 are connected to

switch1, so PM1, PM2 and PM3 form a PM group connected to switch1. We can see that in order to find the PM to host the distributed partitions, we first should find a proper PM group, and we can then select the best PM in this group for our answer. So for our partition distribution algorithm, there are two major steps: 1) locate the proper PM group, and 2) find out the best PM in the PM group resulted from Step 1. These steps are described below.

Step 1: locate the proper PM group:

Creating a VM in a PM requires the provisioning of different physical resources, such as CPU, memory, and I/O bandwidth. For different time points, the amount for each resource used by a VM may be different. In this dissertation, we focus on the CPU and memory provisioned from a PM to a VM. We can use a vector, *VecURes*, to represent the CPU and memory resources used by VMs for a particular PM at the time point *t* as follows:

$$VecURes = < URes_{CPU}, URes_{RAM} > \qquad (10)$$

For each resource, there is always a threshold defined by the user to limit the amount of resource that can be provisioned from a PM to its VMs. If the amount of some resource provisioned from PM to its VMs exceeds the pre-defined threshold, this PM is said to have an overload status. The performance of the VMs in an overloaded PM is very poor, and the PM has a higher chance to fail. We can use a vector, *VecTRes*, to represent the threshold of the CPU and memory resources that can be maximally provisioned from a PM to VM at any time point as follows:

$$VecTRes = < TRes_{CPU}, TRes_{RAM} > \qquad (11)$$

In order to tell how busy a PM is, we can compute the resource utilization fraction of a PM. The resource utilization fraction of CPU equals to the ratio of $URes_{CPU}$ over $TRes_{CPU}$, and the resource utilization fraction of memory equals to the ratio of $URes_{RAM}$ over $TRes_{RAM}$. We can use a vector, *VecFRes*, to represent the CPU and memory resources utilization fraction in a PM at time point *t* as follows:

$$VecFRes = < \frac{URes_{CPU}}{TRes_{CPU}}, \frac{URes_{RAM}}{TRes_{RAM}} > \quad (12)$$

Then we want to measure the degree of overload of resource *i*, which is either CPU or memory, for a PM. This degree is called the overload score of resource *i* for a PM. We use the $overload_score_i$ to represent it. The function used to compute the overload score of resource *i* is from [84]:

$$overload_score_i = \begin{Bmatrix} 0 & VecFRes_i < VecTRes_i \\ e^{\frac{VecFRes_i - VecTRes_i}{VecTRes_i}} & VecFRes_i \geq VecTRes_i \end{Bmatrix} \quad (13)$$

Till now we have presented the degree of overload of resource *i* for a PM. The degree of overload of a PM can be obtained by summing all the overload scores of CPU and memory on this PM. We can use $overload_score_PM_j$ to represent the degree of overload of the *jth* PM as follows:

$$overload_score_PM_j = overload_score_{CPU} + overload_score_{RAM} \quad (14)$$

If there are *N* PMs in the *kth* PM group, then the degree of overload of this PM group ($overload_score_PMG_k$) can be obtained by summing the overload scores of all PMs in this PM group as follows:

$$overload_score_PMG_k = \sum_{j=1}^{N} overload_score_PM_j \quad (15)$$

Now the overload score of each PM group has been computed, if this score is higher for a particular PM group, it means this PM group has a higher chance to fail due to the resource overload problem. So we need to select the PM group whose overload score is the smallest to identify the best PM to distribute the partitions to. Next we will go to Step 2 to select the best PM from this selected PM group.

Step 2: Locate the best PM in the selected PM group.

The best PM to distribute the partitions to should be the one that has the lowest probability to violate the existing tenants' SLAs, i.e., we need to compute the SLA violation probability for each PM in the selected PM group when a new standby database instance is created on this PM. We know that if a PM has a high CPU and RAM, and a low number of tenants (tenant degree), then creating a new standby database instance on such PM may have a low probability of causing a possible SLA violation to other existing tenants.

We define two vectors, *VecCPU* and *VecRAM*, to represent the CPU and memory available to be provisioned for the new standby database instance on each PM, respectively. If the selected PM group contains N PMs then *VecCPU* equals to:

$$VecCPU = \langle CPU_1, CPU_2, \dots, CPU_N \rangle \qquad (16)$$

and *VecRAM* equals to:

$$VecRAM = \langle RAM_1, RAM_2, \dots, RAM_N \rangle \qquad (17)$$

Then the probability of choosing the *jth* PM by considering the possible SLA violation due to CPU overload is $\frac{VecCPU_j}{\sum_{i=1}^{N} VecCPU_i}$. The probability of choosing the *jth* PM by considering possible SLA violation due to memory overload is $\frac{VecCPU_j}{\sum_{i=1}^{N} VecCPU_i}$.

We can see that the probability of choosing a PM as the target to distribute the partitions to is directly proportional to the available CPU and memory of this PM. Unlike the CPU and memory, the probability of choosing a PM is inversely proportional to this PM's tenant degree. We define a vector, $VecTD$, to represent the tenant degree of each PM of the selected PM group as follows:

$$VecTD = \langle TD_1, TD_2, \ldots, TD_N \rangle \qquad (18)$$

Then the probability of choosing the *jth* PM by considering its possible SLA violation due to the tenant degree is $\frac{\frac{1}{VecTD_j}}{\sum_{i=1}^{N}\frac{1}{VecTD_i}}$. So the overall probability of choosing the *jth* PM by considering the possible SLA violations caused by CPU, RAM and tenant degree is:

$$\frac{VecCPU_j}{\sum_{i=1}^{N} VecCPU_i} \times \frac{VecRAM_j}{\sum_{i=1}^{N} VecRAM_i} \times \frac{\frac{1}{VecTD_j}}{\sum_{i=1}^{N}\frac{1}{VecTD_i}} \qquad (19)$$

After estimating the probabilities of choosing a PM considering its possible SLA violations due to CPU, RAM and tenant degree for all PMs in the selected PM group, we will select the PM with the highest probability resulted from equation (19) and distribute the partitions to that PM.

3.3. Communication cost

In the Subsection 3.2 we discussed how to use the overload score to compute the degree of overload for each PM group, and then find the best PM in the selected PM group by considering the probability of SLA violation for each PM. Besides the resource overload factor, there is another factor that can impact the process of choosing the target standby PM to distribute the partitions to. This factor is the communication cost. If the communication time between two PM groups is too high to be ignored, we have to

consider the impact of communication cost, which can work as a weight to the overload score when choosing the proper PM group. If the communication time between two PM groups is small compared to the average query response time, for example the average query response time is measured in seconds and the communication time is measured in milliseconds or even microseconds, then we may ignore the communication delay or scale the weight using a scaler ρ, which equals to the average communication delay over average query response time.

According to [85], multi-path TCP protocols are being used in today's data center in order to improve the whole data center's performance by performing very short timescale distributed load balancing. Multi-path TCP protocols make effective use of parallel paths in modern data center's network topologies. When data is transferred from source to target, more than one path will be used. For example, in Figure 13, if the router wants to route data to the PM group consisting of PM7, PM8 and PM9, the path "router-switch3" and the path "router-switch2-switch3" may be used simultaneously. So when we consider the communication cost between a PM group and other switches in the network, we need to measure the overall communication cost from the switch that is directly connected to that PM group to all other switches. The Closeness Centrality [86] of network can help us solve this problem.

In a network, a node is said to be closer to all other nodes, if the sum of the distances from this node to all other nodes is smaller than that of other candidate nodes [87]. In a data center we can consider a PM group as a candidate node. Formally, the Closeness Centrality Degree (CCD) of a node v is defined as $C_c(v)$, which equals to:

$$C_c(v) = \frac{M-1}{\sum_{a \neq v} d(v,a)} \qquad (20)$$

where M is the total number of nodes in a network and $d(v, a)$ represents the distance between node v and node a. Equation (20) tells us that if a node has the maximum CCD in a network then this node has the minimal overall distance to other nodes.

When we apply the CCD in our case, M equals to the total number of PM groups; *v* represents the a PM group; *a* represents a PM group other than *v*; and $d(v, a)$ represents the communication delay between the *v* and *a.* The CCD of each PM group can be represented using a vector, *VecCCD*, as follows:

$$VecCCD = \langle CCD_1, CCD_2, \ldots, CCD_N \rangle \qquad (21)$$

The bigger CCD for a PM group means this PM group connected has less overall communication delay to other PM groups.

Next we compute the weight using CCD. First we need to compute the inverse of CCD and save it in the vector *VecICCD* as follows:

$$VecICCD = \langle ICCD_1, ICCD_2, \ldots, ICCD_M \rangle \qquad (22)$$

where $ICCD_i = \frac{1}{CCD_i}$. Then we normalize the *VecICCD* vector using Min-Max normalization in order to use this vector as the weight to the overload score vector.

$$VecNICCD = \langle \frac{ICCD_1 - Min(VecICCD)}{Max(VecICCD) - Min(VecICCD)}, \frac{ICCD_2 - Min(VecICCD)}{Max(VecICCD) - Min(VecICCD)}, \ldots, \frac{ICCD_M - Min(VecICCD)}{Max(VecICCD) - Min(VecICCD)} \rangle \qquad (23)$$

For a PM group, when we compute the overload score of this PM group, a lower overload score for a PM group means this PM group has higher chance to be selected as the target PM group. If we want to use communication delay as the weight to penalize the overload score, we should penalize the overload score to make it a bigger value than the

original one. The reason is that we always want to select the PM group with the smallest overload score. If the giving scaler ρ, which equals to $\frac{average\ communication\ delay}{average\ query\ response\ time}$, is not zero then the weighted overload score of the PM group k, $overload_score_PMG_k$, equals to $overload_score_PMG_k \times (1 + VecNICCD_k \times \rho)$. By doing so we consider the communication cost issue of the network. If the communication cost can be ignored when the communication delay is small enough, ρ will be zero and there is no penalty for each PM group's overload score.

The whole partition distribution algorithm (communication delay is considered) is shown as Figure 14.

From Figure 14 we can see the algorithm first computes the overload score for each PM in a PM group (lines 2-5). Then the overload score of the corresponding PM group is calculated by summing each PM's overload score in this group (line 6). After that the algorithm calculates the inverse of this PM group's closeness centrality degree, and adds this degree to *VecICCD* vector (lines 7-8). When *VecICCD* is generated it will be normalized using Min-Max normalization to get *VecNICCD* (line 10). If the communication cost is considered, the algorithm will use a weight to penalize the overload score for each PM group (lines 11-13). Then the PM group *x* with the least overload score will be selected (line 14). For the selected PM group, each PM's selection probability will be computed (lines 15-18). Finally the PM with the biggest selection probability will be returned (line 19).

Input:
1. Number of PM groups (*M*)
2. Number of PMs in the *kth* PM group (N_k)
3. Vector of CPU and memory provisioned to VMs for the *jth* PM in the *kth* PM group (*VecURes*)
4. Vector of CPU and memory provision threshold for the *jth* PM in the *kth* PM group (*VecTRes*)
5. Vector of the CPU provisioned to VMs for each PM in the *kth* PM group (*VecCPU*)
6. Vector of the memory provisioned to VMs for each PM in the *kth* PM group (*VecRAM*)
7. Vector of the tenant degree for each PM in the *kth* PM group (*VecTD*)
8. Weight scaler (ρ)

Output:

The ID of the PM to distribute partitions

Parameters initialized:
1. Vector of workload fraction for each resource of the *jth* PM in the *kth* PM group (*VecFRes*)
2. Vector of the inverse of closeness centrality degree for each PM group (*VecICCD*)
3. Vector of normalized inversed closeness centrality degree for each PM group (*VecNICCD*)

Steps:
1. For each PM group k
2. For each PM j
3. $VecFRes_i = VecURes_i / VecTRes_i$
4. Compute $overload_score_PM_j$ for *jth* PM using *VecFRes*
5. End for
6. Compute $overload_score_PMG_k$ by summing $overload_score_PM_j$
7. Compute closeness centrality degree, CCD_k, for the *kth* PM group
8. Add the inverse of CCD_k to *VecICCD*
9. End for
10. *VecNICCD* = Min-Max normalization of *VecICCD*
11. For each PM group k
12. $overload_score_PMG_k = overload_score_PMG_k \times (1 + VecNICCD_k \times \rho)$
13. End for
14. Find out the PM group, x, with the least overload score.
15. For each PM j in PM group x
16. Compute the selection probability using
17. $\frac{VecCPU_j}{\sum_{i=1}^{N} VecCPU_i} \times \frac{VecRAM_j}{\sum_{i=1}^{N} VecRAM_i} \times \frac{\frac{1}{VecTD_j}}{\sum_{i=1}^{N} \frac{1}{VecTD_i}}$
18. End for
19. Return the ID of the PM with the biggest selection probability

Figure 14. Partition distribution algorithm

Chapter V Performance Analysis

This chapter presents the theoretical and empirical analysis of our algorithms, SLA-LRU and AutoClustC. We will evaluate their performance from different metrics. The empirical analysis is conducted using the TPC-H benchmark [88]. We present the theoretical analysis first followed by the description of our experimental model and the experimental results.

1. Theoretical Analysis

In this section we discuss the time and space complexity of SLA-LRU and AutoClustC. The variables used in the analysis are listed in Table 2.

Table 2. Variables used in the theoretical analysis

Name	Meaning
n_t	# of tenants
n_b	# of entries in the buffer page list
n_c	# of entries in the historical CPU utilization data set
n_i	# of inputs of ANN
n_o	# of outputs of ANN
n_n	# of nodes in hidden layer of ANN
n_{tr}	# of records of the training data set for ANN
n_{ti}	# of training iterations for ANN
n_{PMG}	# of PM groups
m_{PM}	Maximum # of PMs in one PM group

1.1. Complexity analysis for SLA-LRU

SLA-LRU is divided into two parts: 1) analyzing the SLA penalty cost for each tenant, and 2) releasing memory buffer pages when the memory buffer pool is full. All tenants' SLA penalty cost functions are saved in an SLA penalty cost table with a format as shown in Table 3. From Table 3 we can see that each record of the SLA penalty cost table contains the ID, category and SLA penalty cost, which is based on different buffer pool

levels, for the corresponding tenant. For instance, the penalty cost function for Tenant 1 in Table 3 is defined such that the penalty costs are 0, *c*, 2*c* and 4*c,* respectively according to different buffer pool utilization level ranges, which are defined by the cloud service providers.

Table 3. An example entry in the SLA penalty cost table

Tenant ID	Tenant category	Penalty cost (Micro)	Actual buffer pool level range
1	Micro	0	(95%p, 100%p]
		c	(25%p, 95%p]
		2c	(5%p, 25%p]
		4c	[0%p, 5%p]

The SLA penalty cost table is updated when there is a change to this table, i.e., when a new tenant is added/removed from the SLA penalty cost table or the SLA penalty cost function is changed for some tenant in the SLA penalty cost table. The first part of SLA-LRU runs when the algorithm wants to know the page removal cost for different tenants. The second part of SLA-LRU runs when there is no free memory buffer page available for incoming queries. In this section, we present the complexity of each part individually and then the total complexity of SLA-LRU, which is the sum of the complexity of both parts. The time complexity of SLA-LRU is analyzed based on the amount of time SLA-LRU would take to perform a full scan on the memory buffer page list in the worst case.

1.1.1. Time complexity of SLA-LRU

We first present the time complexity of analyzing the SLA penalty cost for each tenant. Before SLA-LRU releases memory buffer pages from the buffer pool, the algorithm has to know the derivative of the SLA penalty cost function for each tenant that is using the buffer pool. The computation of the derivative of the SLA penalty cost function occurs

only once unless a change happened to the SLA penalty cost table. When the algorithm analyzes the SLA penalty cost function for each tenant, a full scan will be performed on the SLA penalty cost table. The length of the table equals to the number of tenants located on the VM, so the time of scanning the SLA penalty cost table will be $O(n_t)$, where n_t is the number of tenants on the VM.

Then we present the time complexity of releasing memory buffer pages when the buffer pool is full. The time complexity of this part is analyzed based on the amount of time SLA-LRU would take to perform a full scan on the memory buffer page list. As we discussed in Section 2 of Chapter III, during the first iteration of the memory buffer page releasing process, the scanning percentage on the memory buffer page list is α. The scanning percentage on the memory buffer page list for the next iteration will always be doubled according to the previous iteration. So we have equation (24)

$$2^{p-1}\alpha = 1 \qquad (24)$$

where p is the total number of scanning times. So p can be represented by using the following equation:

$$p = 1 + \lceil \log_2 \frac{1}{\alpha} \rceil \qquad (25)$$

Then the total number of memory buffer pages scanned can be represented by using the following equation:

$$total_number_of_pages_scaned = (\alpha + 2\alpha + 4\alpha + \cdots + 2^{\lceil \log_2 \frac{1}{\alpha} \rceil}\alpha) \times n_b \qquad (26)$$

where n_b is the number of the entries in the memory buffer page list. Equation (26) is a sum of geometric sequence, and we know that $1 - q^x = (1 - q)(1 + q + q^2 + \cdots + q^{x-1})$. For our case q is 2, so it can be re-written as:

$$total_number_of_pages_scaned = \alpha \times n_b \times (2^{\lceil \log_2 \frac{1}{\alpha} \rceil + 1} - 1) \qquad (27)$$

So the time complexity for releasing memory buffer pages is O($\alpha \times n_b \times (2^{\lceil \log_2 \frac{1}{\alpha} \rceil + 1} - 1)$). Generally, we define α as a meaningful value, so O($\alpha \times n_b \times (2^{\lceil \log_2 \frac{1}{\alpha} \rceil + 1} - 1)$) can be considered as O(n_b). Then the total time complexity of SLA-LRU is O(n_b) + O(n_t). In practice, the number of tenants located on a VM is far less than the number of entries of the memory buffer page list. So we can consider the final time complexity of SLA-LRU to be O(n_b).

1.1.2. Space complexity of SLA-LRU

SLA-LRU stores the derivative of each tenant's SLA penalty cost function with the tenant's ID and class in a hash table in the memory. Moreover, a set is used to save the tenants that have the maximal derivatives of the SLA penalty cost function. Hence the total amount of storage required is O($2n_t$), we can consider the final space complexity to be O(n_t).

1.2. Complexity analysis for AutoClustC

AutoClustC is divided into three parts: 1) cost forecasting for resource provisioning, 2) cost forecasting for database partitioning, and 3) partition distribution. For each tuning process, parts 1) and 2) must be run in order to find out the most cost saving approach, which is either resource provisioning or database partitioning. Part 3) is triggered only when database partitioning is selected. In this section, we present the complexity of each part individually and then the total complexity of AutoClustC, which is the sum of the complexity of all three parts. In order to analyze the complexity for the worst case, we

assume all three parts will be executed, i.e., database partitioning is selected as the tuning method.

1.2.1. Time complexity of AutoClustC

In this section we first present the time complexity of forecasting the cost for resource provisioning, then present the time complexity of forecasting the cost for database partitioning, and finally present the time complexity of partition distribution.

Time complexity of forecasting the cost for resource provisioning:

From Section 1.2 of Chapter IV, we know that the resource provisioning from PM to VM can be estimated using the equation: $E[U] + E_t$, where $E[U]$ is the statistical mean of the measured historical CPU demand and E_t is the error term. So the time complexity of forecasting the cost for resource provisioning is the sum of the time complexity of computing $E[U]$ and the time complexity of computing E_t. In order to compute $E[U]$, historical CPU demand data must be collected. If the number of entries in the historical CPU demand data set is n_c then the time complexity of computing $E[U]$ is $O(n_c)$.

E_t is computed using the AR(2) model. From Section 1.2 of Chapter IV, we know the 2 step prediction error can be represented using the Gaussian variable having mean zero and variance $\sigma_e^2(n) = \sum_{j=0}^{1} G^2(j)\sigma_\epsilon^2$. $G(j)$ is the characteristic function of the AR(2) model, which can be presented using the equation $G(j) = \frac{\gamma_1^{j+1} - \gamma_2^{j+1}}{\gamma_1 - \gamma_2}$. From Section 1.2 of Chapter IV we know that γ_1 and γ_2 are the roots of the equation $1 - \alpha_1 B - \alpha_2 B^2 = 0$, where α_1 and α_2 are two AR(2) parameters. So the time complexity of computing $G(j)$ is based on the time complexity of finding out the value of α_1 and α_2. The Yule–Walker algorithm [89] can be used to calculate α_1 and α_2 using the historical CPU demand data

set. According to [89], the time complexity of finding out α_1 and α_2 for the AR(2) model is a constant. So the time complexity of computing E_t is a constant, O(1).

Overall, the time complexity of forecasting the cost for resource provisioning is O(n_c), where n_c is the number of entries of the historical CPU demand data set.

Time complexity of forecasting the cost for database partitioning:

The cost forecasting process for database partitioning uses ANN, so the time complexity includes two parts: time complexity for training and time complexity for predicting. We first discuss the time complexity for training, then discuss the time complexity for predicting, then sum the two time complexity together to get the final time complexity.

In order to train an ANN (assuming only one hidden layer with n_n nodes existing) with n_i inputs, n_o outputs using a training data set with n_{tr} number of records, each node of the network has to receive the input and take the weight to adjust the input in order to generate a new output. The predicted output will be compared with the corresponding output in the training data set in order to modify the weights. So the forward and reverse propagations' time complexity will always be O($n_i n_n n_o$) for each data tuple in the training data set in one training iteration. If the whole training data set is used for training and n_{ti} training iterations are taken place, the overall time complexity of training process will be O($n_i n_n n_o n_{tr} n_{ti}$).

When ANN is used for prediction, the way of calculating the time complexity for prediction is similar to the way of calculating the time complexity for training. The only difference is that the prediction runs on one data tuple and has no reverse propagation. So the time complexity of the predicting process is O($n_i n_n n_i$).

Now we can sum the two time complexity together, which is O($n_i n_n n_o n_{tr} n_{ti}$) + O($n_i n_n n_i$). In our ANN model n_i equals to 4, n_n equals to 8 and n_o equals to 1, so the final time complexity of forecasting the cost for database partitioning is O($32 n_{tr} n_{ti}$) + O(32) = O($n_{tr} n_{ti}$). We can see that the time complexity of using ANN in our application is mainly determined by the number of records of the training data set and the number of training iterations.

Time complexity of partition distribution:

Based on the discussion in Sections 3.2 and 3.3 of Chapter IV, we know the time complexity of partition distribution equals to the sum of the time complexity of computing the overall weighted overload score for each PM group and the time complexity of finding out the PM with highest selection probability in that PM group.

When partition distribution algorithm is searching the proper PM group to distribute resulted partitions, the algorithm first compute the overall overload score for each PM, which always has constant time complexity, O(1), since we only consider the overload status for CPU and memory. Then the algorithm will compute the overall overload score for a PM group by summing the overload scores of all PMs located in this PM group. If the maximum number of PMs in one PM group is m_{PM}, the time complexity of finding out the overall overload score for one PM group is O(m_{PM}). If totally there are n_{PMG} PM groups, the time complexity of finding out the overall overload score for all PM groups is O($m_{PM} n_{PMG}$). Once the overall overload score is computed for each PM group, the closeness centrality degree will be computed for each PM group in order to analyze the communication delay. The time complexity of computing the closeness centrality degree

is O($n_{PMG}{}^2$). So the total time complexity of search the proper PM group is O($m_{PM}n_{PMG}$) + O($n_{PMG}{}^2$).

Next, the algorithm will compute the selection probability for each PM in the selected PM group in order to find out the proper PM to distribute partitions. Since we only consider the impact of CPU, memory and tenant degree for each PM the time complexity of computing the selection probability for each PM will be constant. We know there are m_{PM} PMs in the PM group, so the time complexity of finding out the proper PM to distribute partitions is O(m_{PM}).

We can sum the above time complexities to get the total time complexity for partition distribution, which is O($m_{PM}n_{PMG}$) + O($n_{PMG}{}^2$)+ O(m_{PM}) = O($m_{PM}n_{PMG}$) + O($n_{PMG}{}^2$).

1.2.2. Space complexity of AutoClustC

In order to compute the space complexity of AutoClustC, we have to find out the space complexity for forecasting the cost for resource provisioning, the space complexity for forecasting the cost for database partitioning and the space complexity for partition distribution. Then the sum of the three space complexity is AutoClustC's space complexity.

Space complexity for forecasting the cost for resource provisioning:

When AutoClustC forecasts the cost of resource provisioning, the historical CPU demand data set has to be saved in the memory, so the space complexity for forecasting the cost for resource provisioning equals to O(n_c), where n_c is the number of entries in the historical CPU demand data set.

Space complexity for forecasting the cost for database partitioning:

When AutoClustC forecasts the cost of database partitioning using ANN, the training data set has to be saved in memory with O(n_{tr}) space complexity, where n_{tr} is the number of records in the training data set. In order to make ANN running, the memory for each node's weight has to be allocated. Since there are only 4 inputs and 1 hidden layer with 8 nodes, we can consider the space complexity of allocating memory to nodes' weight as a constant. So the overall space complexity for forecasting the cost of database partitioning is O(n_{tr}).

Space complexity for partition distribution:

There are 3 major steps when AutoClustC performs the partition distribution. In the first step it computes the overload score for each PM of a PM group. In the second step it computes the weight for each PM group and find out the overall weighted overload score for each PM group. In the third step AutoClustC selects the PM with the highest selection probability from the candidate PM group.

When AutoClustC performs the first step, two resources, CPU and memory, are considered. So the space complexity of computing the overload score for one PM is constant, which is O(1).

When AutoClustC performs the second step, it has to allocate memory to save the distance of each PM group pair. If there are n_{PMG} PM groups, the maximum number of PM group pairs equals to $\frac{{n_{PMG}}^2 - n_{PMG}}{2}$ and this will use O(${n_{PMG}}^2$) space complexity. AutoClustC also needs to save the overload score of all PMs in the data center, and this will use O($n_{PMG} m_{PM}$) space complexity, where m_{PM} is the maximum number of PMs of

one PM group. So the total space complexity of the second step is O($n_{PMG}m_{PM}$) + O($n_{PMG}{}^{2}$).

When AutoClustC performs the third step, the selection probability of each PM in the candidate PM group has to be saved, so the space complexity of step 3 is O(m_{PM}).

Then the space complexity for partition distribution equals to O($n_{PMG}m_{PM}$) + O($n_{PMG}{}^{2}$) + O(m_{PM}) = O($n_{PMG}m_{PM}$) + O($n_{PMG}{}^{2}$).

Now we can sum the three space complexity of AutoClustC together, which is O(n_c) + O(n_{tr}) + O($n_{PMG}m_{PM}$) + O($n_{PMG}{}^{2}$), to get the final space complexity for AutoClustC.

1.3. Summary of worst-case time and space complexity analysis results

The summary of the time and space complexity analysis results is shown in Table 4.

Table 4. Summary of worst-case time and space complexity analysis results

	Time complexity	Space complexity
SLA-LRU	O(n_b)	O(n_t)
AutoClustC	O(n_c) + O($n_{tr}n_{ti}$) + O($n_{PMG}m_{PM}$) + O($n_{PMG}{}^{2}$)	O(n_c) + O(n_{tr}) + O($n_{PMG}m_{PM}$) + O($n_{PMG}{}^{2}$)

2. Experimental Analysis

We have conducted an extensive set of experiments to study the performance of our algorithms SLA-LRU and AutoClustC. We also compare SLA-LRU with two existing buffer pool management technique, LRU-2 [24] and MT-LRU [30] which we have reviewed in Chapter II. For AutoClustC, to the best of our knowledge it is the first cloud database tuning algorithm that is based on vertical database partitioning. So we have no similar algorithm to compare with. However, we compare the performance of the resulting partitions when using distribution with that of not using distribution. We present the simulation models in Section 2.1 and the experimental results in Section 2.2.

2.1. Simulation Models

The goal of the simulation model is to demonstrate the effectiveness of our algorithms. We build two experiment models to test the performance of our SLA-LRU and AutoClustC algorithms, respectively. The details of our simulation models are discussed in the following subsections.

2.1.1. Simulation model for SLA-LRU

In order to implement our algorithm, we use the open source MySQL Server 5.5 [75] as our database platform. MySQL Server 5.5 uses the classic LRU-2 algorithm to manage the memory buffer pool. We modify this component of the original code to include our SLA-LRU. We conduct experiments comparing SLA-LRU with the two existing buffer pool management algorithms, LRU-2 [24] and MT-LRU [30] using the TPC-H benchmark [88] database as our dataset and the TPC-H bench mark queries as the query set. Totally there are 8 database tables: ORDERS, SUPPLIER, LINEITEM, PARTSUPP, CUSTOMER, PART, NATION and REGION, and 22 query types. All the code modifications are done using C and all test scripts are written using tcl scripts. The experiments are performed on a machine with a CPU of i5-2400 3.10 GHz and 4 GB memory.

In order to run SLA-LRU and MT-LRU, the SLA penalty cost functions have to be pre-defined. For SLA-LRU, we classify tenants into 4 categories: Micro, Small, Medium and Large. We adopt the class categories used by Amazon T2 instance [73], which are shown in Table 5. The SLA promised buffer pool levels are 5%, 10%, 20% and 40% of the corresponding VM's memory for Micro, Small, Medium, and Large tenants, respectively. According to the vCPU penalty cost function used by Amazon [73], the

penalty costs for different categories of tenants should follow a pattern of exponential of 2. For example, Amazon defines the vCPU penalty cost for Micro, Medium and Large tenant categories as $2^1 \times 3=6$, $2^2 \times 3=12$ and $2^3 \times 3=24$, respectively. So we will also use the pattern of exponential of 2 in our experiments. Based on the discussions in [30] and [74], step-based resource utilization is widely used in practice to decide the penalty cost. So we also use the similar step-based actual buffer pool level to decide the penalty cost for each category of tenants in our experiments as shown in Table 6, where c is a weight to compute the penalty cost, and p is the SLA promised buffer pool level for the corresponding tenant category. From this penalty cost function we can see that the service provider does not have any penalty cost when the tenant's buffer pool level just starts to decrease from the promised level until the buffer pool level falls below 95% of the promised level. Once the tenant's buffer pool level falls below 25%, the penalty cost to the service provider becomes very high. For MT-LRU, we use the penalty cost function shown in Table 7. The same as the way we defined Table 6, the penalty cost pattern in Table 7 also follows [73], and the HRD pattern follows [30] and [74].

Table 5. Tenant's category

Tenant category	Promised buffer pool level
Large	40%
Medium	20%
Small	10%
Micro	5%

Table 6. Buffer pool level based SLA penalty cost function for SLA-LRU

Penalty cost (Large)	Penalty cost (Medium)	Penalty cost (Small)	Penalty cost (Micro)	Actual buffer pool level range
0	0	0	0	(95%p, 100%p]
8c	4c	2c	c	(25%p, 95%p]
16c	8c	4c	2c	(5%p, 25%p]
32c	16c	8c	4c	[0%p, 5%p]

Table 7. Hit ratio degradation based SLA penalty cost function for MT-LRU

Penalty cost (Category I)	Penalty cost (Category II)	HRD range
0	0	[0%p, 5%]
c	4c	(5%p, 25%]
2c	8c	(25%p, 95%]
4c	16c	(95%p, 100%]

2.1.2. Simulation model for AutoClustC

AutoClustC is triggered when the performance related SLA of a tenant is violated. We use percentile query response time, which is discussed in detail in Section 2.3.2 of this chapter, as the measurement in the performance related SLA. AutoClustC first forecasts the costs for resource provisioning and database partitioning, then selects the lower cost method to tune the database. If database partitioning is chosen, a partitioning algorithm based on AutoClust [20] is run to partition the corresponding tenant's database, and distribute the resulting partitions to proper PM in the same data center. Sometimes, such PM is called standby PM since the database instance on this PM is not the master database instance. The number of PMs that can be used to work as the standby PMs is defined in the tenants' SLAs. In our experiments, we perform two round tests. In the first round, we conduct experiments to test the accuracy of each prediction model in our algorithm, and then conduct experiments to test the performance of the resulting partitions if database

partitioning is selected as the tuning method on the master database instance without running the partition distribution process. In the second round test, we assume database partitioning is selected, and directly conduct experiments to test the performance of the resulting partitions performance when running the partition distribution process.

The first round experiments are done on an AMAZON RDS cloud [6] with the database instance class of db.m1.medium and database engine of SQL Server SE 11.00.5058.0.v1. The second round experiments are done in a virtual environment based on three physical servers, PM1, PM2 and PM3. PM1 and PM3 are Dell PowerEdge R510 servers with Intel Xeon CPU E5645@2.4 GHz and 16 GB RAM. PM2 is a Dell PowerEdge 2900 server with Intel Xeon CPU E4310@1.6 GHz and 8 GB RAM. All machines run Microsoft SQL Server 2008 edition. The experiment program is coded in Java and tcl script. The TPC-H [88] database is used as our dataset. The TPC-H queries are used as the query set. Totally there are 8 database tables: ORDERS, SUPPLIER, LINEITEM, PARTSUPP, CUSTOMER, PART, NATION and REGION, and 22 query types in the TPC-H benchmark.

In order to train the ANN, the following training data set as shown in Table 8 is used.

Table 8. Sample dataset used for ANN training

ID	Database size (MB)	# query types	# users	# attributes	CPU time (cycles)
1	42.6	8	12	16	1.88
2	42.6	15	12	16	5.77
3	145.5	8	12	16	4.19
4	145.5	15	12	16	13.13
5	267	22	8	16	29.7
6	317	8	12	16	8.19
7	237.5	15	8	16	26.84
8	237.5	22	8	16	38.6
9	395	22	12	16	44.91
10	571.4	15	8	16	35.24

2.2. Competitive algorithms

We compare SLA-LRU with two existing algorithms that we have reviewed in Chapter II: LRU-2 [24] and MT-LRU [30]. We restate the keys ideas of these two algorithms here for ease of reference.

LRU-2 keeps tracking two lists: the referenced page list and the buffer page list. A page will be added to the referenced page list when it is referenced for the first time, and this page's reference time is increased by 1. When this page is referenced again its reference time becomes 2, i.e. the page has been referenced two times, it will be moved to the buffer page list, which is ranked based on the pages' timestamps in a decreasing order (the oldest page is at the top of the list). When a page replacement occurs in the memory buffer, the page with the oldest timestamp in the buffered page list will be removed from the buffer and the buffer page list.

MT-LRU focuses on a multi-tenancy environment. This technique considers the buffer page hit ratio degradation (HRD) as the metric in different tenants' SLAs in order to manage the buffer memory. According to the description in [30], when a query accesses a memory buffer page, if it is found in the buffer pool, this access is referred as a hit; otherwise this access is a miss. The Hit Ratio (HR) is defined as: $HR = \frac{h}{N}$, where N is the total number of page accessed and h is the number of page accesses found in the buffer pool. Then $HRD = \max\{0, HR_B - HR_A\}$, where HR_B is the hit ratio when the promised memory buffer is statically reserved for the tenant; HR_A is the hit ratio when memory buffer is dynamically shared by multiple tenants. When MT-LRU releases memory buffer pages from buffer pool, besides the reference times and timestamp of buffer pages, it also

consider the *HRD* for each buffer page, hence the SLAs of different tenants are considered.

2.3. Performance metrics

In this section we present the performance metrics for our SLA-LRU and AutoClustC algorithms, respectively.

2.3.1. Performance metrics for SLA-LRU

We measure the performance of the SLA-LRU algorithm based on two performance metrics: (1) the average query response time of processing one TPC-H query set, and (2) the SLA violation penalty cost improvement ratio.

Average query response time of processing one TPC-H query set:

In the TPC-H benchmark query set, there are 22 query types. A different query type has different complexity. Some query types may not contain any nested query, while some query types may contain several nested queries. So for those 22 query types, they have different query response time. A query of a simple query type may take less than 1 second to be processed, while a query of a complex query type may take more than 10 seconds to be processed. That is why we use the average query response time of processing one TPC-H query set, instead of one TPC-H query, as our performance metric. We run the TPC-H benchmark query set 10 times and calculate the total query response time. Then we use the total query response time to compute the average benchmark query set processing time which is the average time to finish processing one TPC-H benchmark query set.

SLA violation penalty cost improvement ratio:

Once we know the average query response time of processing one TPC-H benchmark query set for two algorithms that we want to compare against each other, we also want to know how much better one algorithm works compared to the other algorithm in terms of the SLA violation penalty cost. We can use the SLA violation penalty cost improvement ratio metric to measure the improvement. The SLA violation penalty cost improvement of an algorithm A over an algorithm B is calculated using the formula, $\frac{100\%.(SLAPenaltyCostB - SLAPenaltyCostA)}{SLAPenaltyCostB}$ where $SLAPenaltyCostB$ is the SLA violation penalty cost after running a query set when algorithm B is used as the memory buffer page replacement algorithm and *SLAPenaltyCostA* is the SLA violation penalty cost after running the same query set when algorithm A is used as the memory buffer page replacement algorithm. If the result is bigger than zero, then we say that algorithm A can reduce the SLA violation penalty cost for the service provider compared to algorithm B, i.e., algorithm A performs better than algorithm B in terms of SLA violation penalty cost.

2.3.2. Performance metrics for AutoClustC

For the first round experiment, we measure the performance of the AutoClustC algorithm based on three performance metrics: (1) the prediction accuracy of the AR(2) and ANN models, (2) the monetary cost ratio of resource provisioning to database partitioning, and (3) the query response time of the resulting partitions when applied database partitioning to repartition the database for performance tuning.

Prediction accuracy of the AR(2) and ANN models:

When the AR(2) model is used for estimating the CPU time for resource provisioning, according to the discussion in Section 1.2 of Chapter IV, we know the estimated CPU

utilization can be represented by the sum of the prediction value and prediction error. So the error ratio of using the AR(2) model to forecast the resource provisioning cost can be calculated as $\frac{prediction\ error}{prediction\ error + prediction\ value}$.

When the ANN model is used for estimating the CPU time for database partitioning, we need to train the ANN model in order to use this model to do prediction. For each iteration of the training process, we have an error ratio which represents the difference between the predicted value and the real value. We use the Mean Square Error (MSE) of all iteration errors to calculate the final accuracy of the ANN model.

Ratio of monetary cost of resource provisioning to monetary cost of database partitioning:

In order to see which tuning method, resource provisioning or database partitioning, is the better cost saving method, , we can calculate the ratio the monetary cost of resource provisioning to the monetary cost of database partitioning as $\frac{C(CPUCostOfResourceProvisiong)}{C(CPUCostOfDatabasePartitioning)}$, where C is a function that computes the monetary cost given the CPU cost. Generally, DbaaS providers define the C function by themselves when they started to provide any services. In our experiment, we use a linear C function.

Query response time of new resulting partitions:

When database partitioning is selected as the method to tune the database performance, some new partitions will be generated. In order to measure how well the new partitions perform, we can calculate the percentile query response time for each query type. The percentile query response time of a particular query type is defined as the 95th percentile query response time of all queries which belong to the same query type.

For the second round experiment, we measure the performance of the AutoClustC algorithm based on the following performance metrics: query response time improvement of processing the whole TPC-H query benchmark using partition distribution over processing it without using partition distribution.

Query response time improvement of processing the whole TPC-H query benchmark with partition distribution over the query response time without partition distribution:

The main reason of distributing partitions to other standby machines is that the workload can be split by different database instances when the master database instance is heavily overloaded. By implementing the partition distribution process, the probability of performance SLA violation can be reduced. In order to measure how the distribution of the new partitions perform, we can calculate the query response time improvement, which is the ratio of the time difference between the time of processing the whole TPC-H benchmark query set without partition distribution and the time of processing the same query set with partition distribution over the time without partition distribution. It is computed as $\frac{TimeWithoutDistribution - TimeWithDistribution}{TimeWithoutDistribution}$, where *TimeWithDistribution* represents the overall query response time of processing the whole TPC-H benchmark query set with partition distribution and *TimeWithoutDistribution* represents the overall query response time of processing the whole TPC-H benchmark query set without partition distribution.

2.4. Experimental results

In this section we show the experimental results of the SLA-LRU algorithm in the first subsection, and the experimental results of the AutoClustC algorithm in the second subsection.

2.4.1. Experimental results for SLA-LRU

In this subsection, we compare LRU-2, MT-LRU and SLA-LRU in terms of query response time and SLA penalty cost.

2.4.1.1. Comparison of query response time of LRU-2, MT-LRU and SLA-LRU

As we discussed in Section 5 of Chapter III, the increase of the scanning time on the buffer page list is the main overhead of SLA-LRU, and the additional operation of saving a hit or miss of a buffer page access for each buffer read request is the main overhead of MT-LRU. From Figure 15 we can see that the average query response time from processing the whole benchmark query set using LRU-2 is a little bit less than that when using SLA-LRU, but much less than that when using MT-LRU. The reason is that, for MT-LRU, each buffer page access request has to be analyzed so that the algorithm can know whether this access is a hit or miss, and this result has to be saved in memory. When there is no free space in the buffer pool, MT-LRU has to scan the corresponding memory in order to know the hit ratio degradation for each tenant. Since the number of buffer page requests is large, especially when the number of tenants is big, the memory scanning process will take a long time to be done.

From Figure 15 we can see the maximum difference between LRU-2 and SLA-LRU occurs when the number of tenants equals to 10. When there are 10 tenants on one instance, the average benchmark processing time using LRU-2 is 329 seconds and the

average benchmark processing time using SLA-LRU is 341 seconds. We can see that SLA-LRU is slower than LRU-2 by (341-329)/329=3.6%, i.e., on average, using SLA-LRU to process a query will take (329×3.6%)/22=0.54 second more than that of using LRU-2 averagely. If we compare the query response time between MT-LRU and SLA-LRU, we will see that averagely MT-LRU has to spend around 58 seconds more to process the whole benchmark query set. Also we can see that the average benchmark processing time increases when the number of tenants increases. More tenants means more workloads, and the queries have to be queued for the processor to process them. This is why the average benchmark processing time increases when the number of tenants increases.

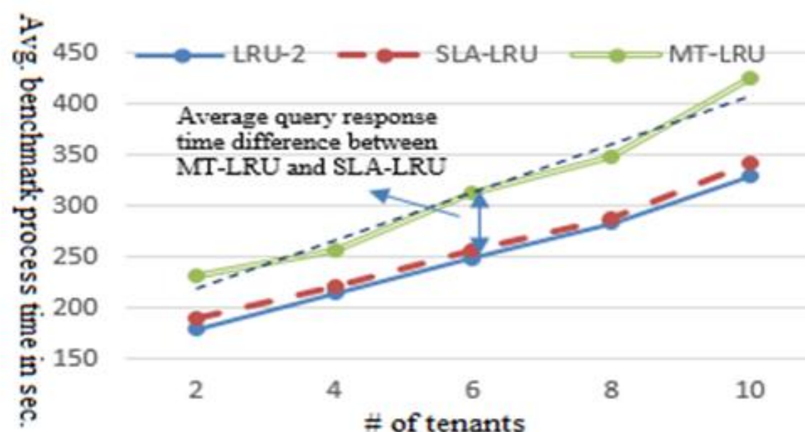


Figure 15. Average TPC-H benchmark query set processing time

2.4.1.2. Comparison of SLA penalty costs of LRU-2, MT-LRU and SLA-LRU

We measure the SLA penalty cost incurred by the LRU-2, MT-LRU and SLA-LRU algorithms when the buffer pool is shared by multiple tenants. In this test, for SLA-LRU we use the penalty cost function presented in Table 6; and for MT-LRU we use the penalty cost function shown in Table 7. We randomly assign different tenants to different tenants'

categories which are defined in Table 5. We use 8 tenants as an example to illustrate how to measure the penalty cost. The tenants' categories used in the example are shown in Table 9. We first compare SLA-LRU with LRU-2, and then compare MT-LRU with LRU-2.

Table 9. Eight tenants' categories

Tenant ID	Tenant categories	Promised buffer pool level
1	Small	10%
2	Medium	20%
3	Medium	20%
4	Small	10%
5	Micro	5%
6	Micro	5%
7	Micro	5%
8	Large	40%

Comparison between LRU-2and SLA-LRU:

The actual buffer pool levels assigned to each tenant before performing any buffer management algorithm and after performing LRU-2 and SLA-LRU are shown in the columns labelled "Preload," "LRU-2," and "SLA-LRU," respectively in Figure 16. The SLA penalty cost is computed based on the buffer assignment status after using each algorithm. For example, from Table 9 we know that tenants 5, 6 and 7 are micro tenants; tenants 1 and 4 are small tenants; tenants 2 and 3 are medium tenants; and tenant 8 is a large tenant. So the promised buffer pool level for each tenant is shown in Table 10.

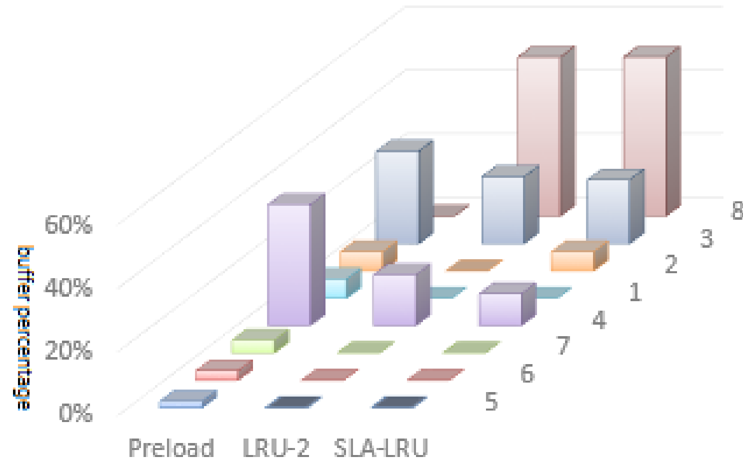


Figure 16. Buffer assignment status (actual buffer pool level) before performing any buffer management algorithm and after performing LRU-2 and SLA-LRU

Table 10. Promised buffer pool level for each tenant for SLA-LRU

Tenant ID	5	6	7	4	1	2	3	8
p / total buffer size	5%	5%	5%	10%	10%	20%	20%	40%

Before running any page replacement algorithm (preload status), from Figure 16 we can see that the actual buffer pool level for each tenant is shown in Table 11.

Table 11. Actual buffer pool level for each tenant before running algorithms

Tenant ID	5	6	7	4	1	2	3	8
Actual buffer utilization / total buffer size	2%	3%	4%	38%	6%	6%	29%	1%
Actual buffer utilization / p	40%	60%	80%	380%	60%	30%	145%	<1%

After running the LRU-2 algorithm, from Figure 16 we can see that the new actual buffer pool level for each tenant is shown in Table 12.

Table 12. Actual buffer pool level for each tenant after running LRU-2

Tenant ID	5	6	7	4	1	2	3	8
Actual buffer utilization / total buffer size	1.5%	1.5%	1.5%	15%	<1%	1%	20%	50%
Actual buffer utilization / p	30%	30%	30%	150%	4%	5%	100%	125%

Using the data given in Table 6 and Table 12, we can calculate the penalty cost for each tenant. For example, for tenant 1 (Small tenant), from Table 12 we can see the actual buffer pool level is 4% of p. Based on Table 6, when the actual buffer level range is [0%, 5%], for a Small tenant the SLA penalty cost is 8c. Similarly we can derive the penalty costs for the rest of the tenants which are shown in Table 13.

Table 13. Penalty cost for each tenant after running LRU-2

Tenant ID	5	6	7	4	1	2	3	8
Penalty cost	c	c	c	0	8c	16c	0	0

So after using the LRU-2 algorithm, the SLA penalty cost is $\mathrm{c} + \mathrm{c} + \mathrm{c} + 0 + 8\mathrm{c} + 16\mathrm{c} + 0 + 0 = 27\mathrm{c}$. Similarly, the SLA penalty cost after running SLA-LRU can be computed as $\mathrm{c} + \mathrm{c} + \mathrm{c} + 0 + 8\mathrm{c} + 8\mathrm{c} + 0 + 0 = 19\mathrm{c}$. We can see that the major difference of memory buffer reassignment between the two algorithms occurs on tenant 2 and tenant 4. Tenant 2 is a medium class tenant, and tenant 4 is a small class tenant. The SLA-LRU algorithm analyzes the "values" of the two tenants, which are the SLA violation penalty costs, and decide to free more buffer pages of tenant 4 instead of tenant 2. By doing this the SLA penalty cost could be reduced by $(27\mathrm{c} - 19\mathrm{c})/27\mathrm{c} = 30\%$ according to the penalty function used in Table 6.

The performance of SLA-LRU over LRU-2 based on the penalty cost ratio for different numbers of tenants is shown in Figure 17, where we can see that the service

provider could reduce the penalty cost when using the SLA-LRU algorithm to manage the buffer pool for most cases. From Figure 17, we conclude that the average SLA violation penalty cost improvement of using SLA-LRU over LRU-2 is about $\frac{(1-1)+(1.3-1)+(1.2-1)+(1.51-1)+(1.42-1)}{5} \approx 29\%$. Also we can see that when there are more than two tenants, SLA-LRU generally can provide good penalty cost savings. But the performance of LRU-2 and SLA-LRU looks similar when there are only two tenants (the penalty cost ratio is 1 in Figure 17). The reasons for the similar performance could be the following: 1) the buffer pool is big enough to hold two tenants' databases; and 2) the incoming queries of tenants do not fluctuate too much, i.e., few buffer page replacements are needed to process all queries.

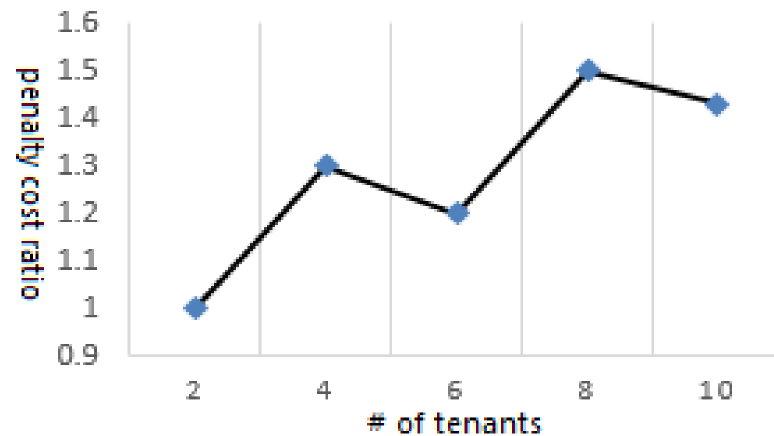


Figure 17. SLA penalty cost ratio of using LRU-2 over using SLA-LRU

Comparison between LRU-2 and MT-LRU:

For MT-LRU, the SLA penalty cost function is shown in Table 7 of Subsection 2.1.1. We assume there are two categories of tenants as shown in Table 7. Tenants belonging to category II are more important than those tenants belonging to category I. So the penalty cost for category II tenants is higher.

The performance of MT-LRU over LRU-2 based on the penalty cost ratio for different numbers of tenants is shown in Figure 18, where we can see that the service provider can reduce the penalty cost when using the MT-LRU algorithm to manage the buffer pool for most cases. The average penalty cost improvement is about $\frac{(1-1)+(1.1-1)+(1.7-1)+(2.5-1)+(2.3-1)}{5} \approx 72\%$. If the query response time can be ignored, MT-LRU can save more money than SLA-LRU for service providers. However, from Figure 15 we can see that MT-LRU cannot guarantee a fast query response time compared to LRU-2 and SLA-LRU, i.e., it sacrifices query response time too much in order to maintain a low SLA penalty cost.

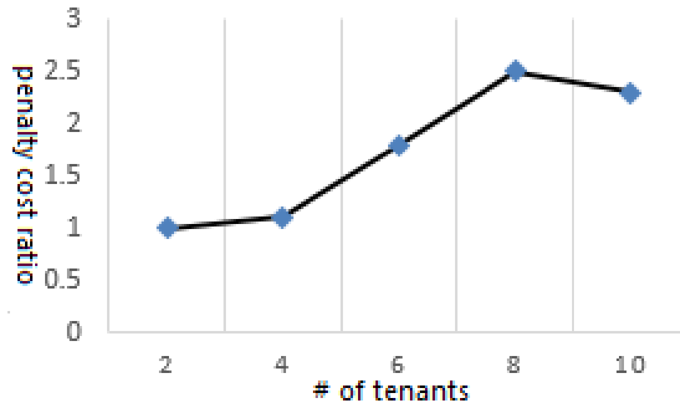


Figure 18. SLA penalty cost ratio of using LRU-2 over using MT-LRU

We can conclude the performance of SLA-LRU and MT-LRU compared to LRU-2 in Table 14 using the average query response time results shown in Figure 15 and the average penalty cost improvements that we have calculated from Figures 17 and 18. From Table 14 we can see that MT-LRU gives the best SLA violation penalty cost but the worst query response time; and SLA-LRU and LRU-2 have compatible query response time, but SLA-LRU has much better SLA violation penalty cost than LRU-2 does. Overall, SLA-LRU gives the best performance considering both metrics together.

Table 14. Overall performance of SLA-LRU and MT-LRU

	SLA-LRU	MT-LRU
Average penalty cost improvement over LRU-2	29%	72%
Averagely query response time increase over LRU-2	3.6%	21.3%

2.4.2. Experimental results for AutoClustC

In this subsection, we first present our experiments results for the first round test: the performance of the ANN model for database partitioning cost forecasting, the performance of the AR(2) model for resource provisioning cost forecasting, the monetary cost ratio of resource provisioning cost to database partitioning cost, and the performance of the new resulting partitions. Secondly, we present our experiments results for the second round test: performance improvement of using partition distribution.

2.4.2.1. Performance of the ANN Model for Database Partitioning Cost Forecasting

The training dataset of the ANN model is from the monitor system called Cloud Watch Service provided by Amazon RDS. 20% of the dataset is used as the validation set and 20% of the data set is used as the test set. The performance measured in MSE is shown in Figure 19. From this figure we can see that the best performance MSE is about 2.12 happened at 720 epochs, which is a measure of the number of times all of the training vectors are used once to update the weights. If more epochs are performed, the network will be over trained since the MSE of validation increases after 720 epochs. Then the linear regression graph is shown in Figure 20. If the training were perfect, the outputs of the network would be exactly equal to the targets of the network, i.e. the dash line and color line in the graph should be 100% overlapped (R=1); but the relationship is rarely

perfect in practice. From Figure 20 we can see that the training, validation and test are all fitting the network targets well.

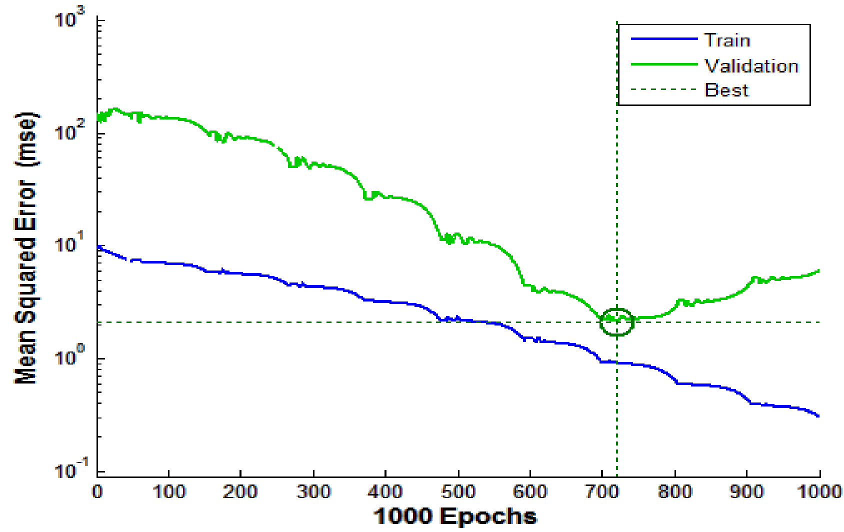


Figure 19. ANN performance on forecasting database partitioning CPU time

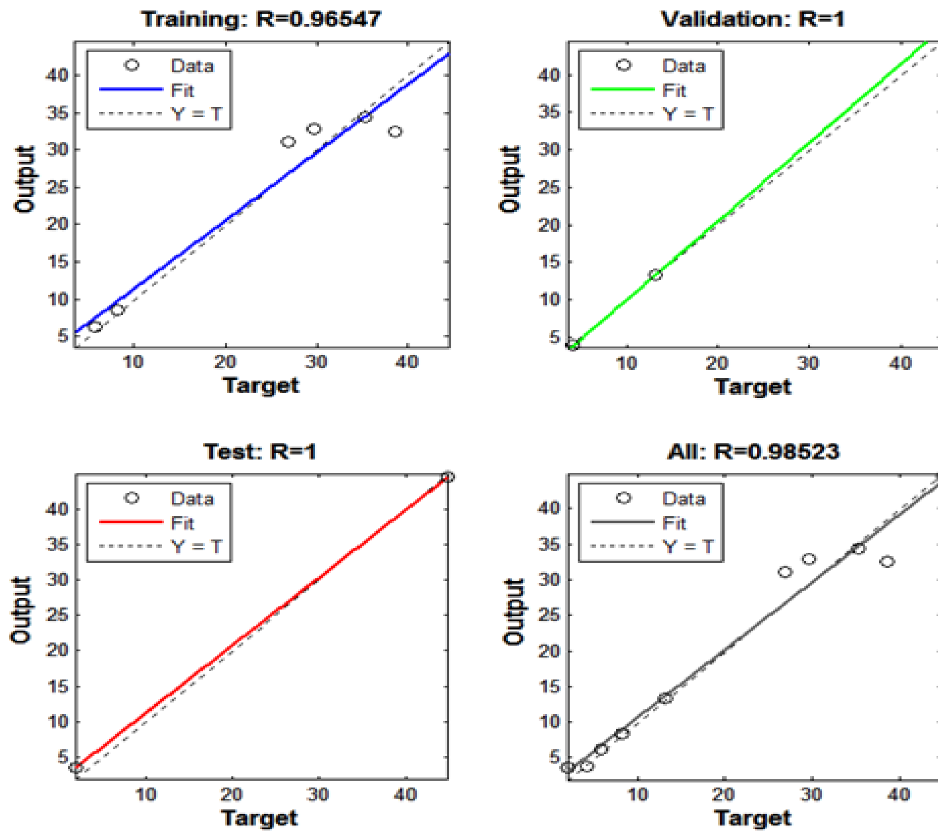


Figure 20. Linear regression of the network on forecasting the database partitioning CPU time

2.4.2.2. *Performance of the AR(2) model for Resource Provisioning Cost Forecasting*

As discussed in Section 1.2 of Chapter IV, the accuracy of the cost forecasting for resource provisioning is the accuracy of estimating the prediction error whose probability distribution follows a normal distribution with zero mean. Figure 21 shows the probability distribution of the prediction error. We can see that 95% of the prediction errors range from -0.015 to 0.015. Comparing with the statistic mean of the historical data, which is 0.31, the error rate of forecasting is about $\frac{0.015}{0.015+0.31} = 4.6\%$.

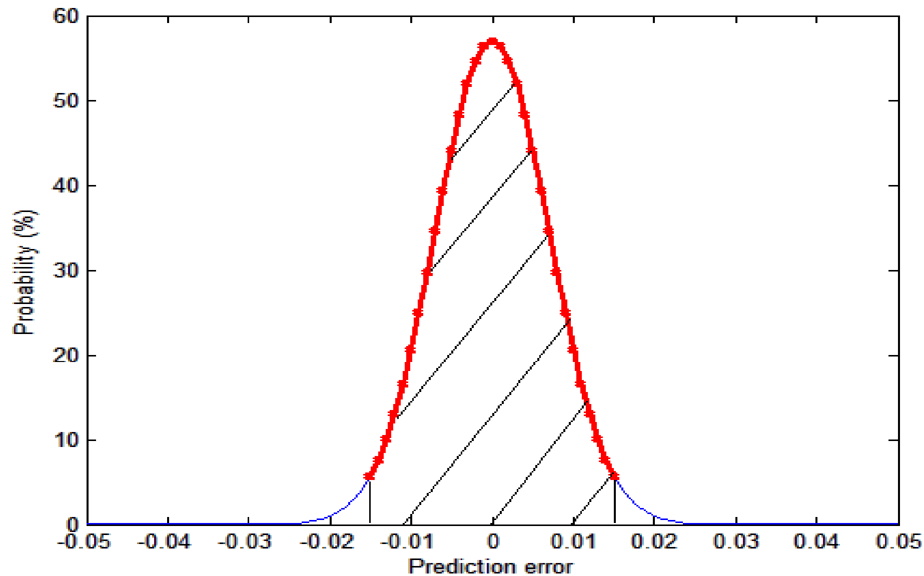


Figure 21. Probability distribution of prediction error in resource provisioning CPU time forecasting

2.4.2.3. *Ratio of monetary Cost of Resource Provisioning Cost to Monetary Cost of Database Partitioning*

Typically, after resource provisioning, the new assigned resources on the VM may take up to serval minutes for the acquired VM to be ready to use. This time is dependent

on the image size (the size of the data mounted from a PM to a VM), VM type, data center location, etc. [90]. So the new assigned resources will not be released in minutes. We assume the dynamic provisioned resources will be kept at least for 30 minutes, which is also the time interval of two sample neighbor data points as discussed in Section 1.2 of Chapter IV. From Section 2.4.2.1 of this chapter, we use the training data to estimate the CPU time cost of partitioning using the ANN model. The estimation is 42.55 CPU time units. From the section 2.4.2.2 of this chapter we estimate the average CPU time provisioned to VM. The estimation is $0.325 \times 30 \times 60 = 585$ CPU time units. Since the *C(CPU_Demand)* function is linear, the final provision monetary cost measured in dollar will be about $\frac{C(585)}{C(42.55)} \approx \frac{585}{42.55} \approx 14$ times as the final monetary cost of database partitioning.

2.4.2.4. Performance of the New Database Partitions

In Section 2.4.2.3 we can conclude that the database partitioning method costs less money than resource provisioning; so the database partitioning method will be used for performance tuning to re-guarantee the performance SLA. In this subsection the performance in terms of percentile query response time of the new database partitions after the partitioning process is presented in Figure 22. We define the performance SLA as follows: at least the 95th percentile query response time of each query type must be within a specific time threshold, TH; otherwise the performance SLA is said to be violated. When a performance SLA violation occurs, the partitioning process will be triggered and the new partitions will be generated to replace the old ones. This experiment is conducted as follows:

1. A tenant's behavior is simulated on a cloud database.

2. Half number of the query types are randomly selected from the TPC-H query type benchmark, and each type of query is executed for a random number of times (less than 300).
3. Once all queries are successfully finished, Step 2 is repeated until the experiment time of 5 hours is reached.
4. In Step 3 if a performance SLA violation is detected, the partitioning process is triggered and new partitions are generated before Step 2 is repeated.

In Figure 22, each colored line represents the percentile query response time of the query corresponding to that color. The red dashed line represents the pre-defined performance SLA (TH). From Figure 22 we can see that the partitioning process occurs 3 times at the three time points 1, 2, and 3, i.e., performance SLA violations occur at the 3 time points in 5 hours. At time point 1, which is at about 1,800 seconds in the experiment time, a performance SLA violation is detected for query type 18; at time point 2, which is at about 7,100 seconds in the experiment time, a performance SLA violation is detected for query type 21; and at time point 3, which is at about 14,700 seconds in the experiment time, a performance SLA violation is detected for query types 9, 12 and 19. The performance SLA violations are caused by query pattern changes since the query set running on the cloud database is randomly changed in Step 2. If the time threshold TH is defined as 7 seconds then from this figure, we can see that the performance SLA is re-guaranteed again after the partitioning process is completed (shown as the colored line falling below the red dashed line again after the partitioning process is completed). Figure 22 also can give a general idea to the service providers of what performance SLA should be made between them and their customers. If the customers are asking for a better

response time, like 5 seconds for example, then from Figure 22, the providers can know that such performance SLA is really hard to guarantee if they still use the current VM configuration. In that case, they can provide better computing resources to the customers by charging a service upgrade fee. So our algorithm can also help the providers make a profitable decision on deriving a correct performance SLA.

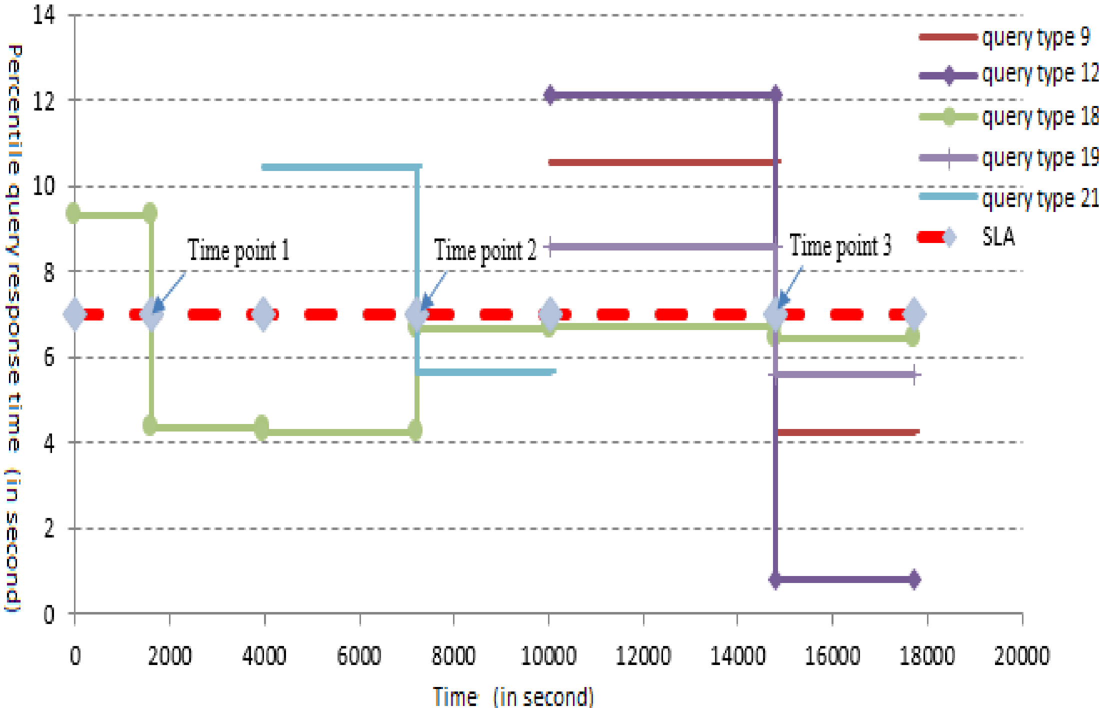


Figure 22. Query response time of 95th percentile of query for different query types before and after database partitioning

2.4.2.5. Query response time improvement of processing the whole TPC-H benchmark query set with partition distribution over the query response time without partition distribution

The database partitioning algorithm in AutoClustC is based on AutoClust, so before we conduct experiments we need to set up some parameters used by the database partitioning algorithm. The parameter setup needs to be done only once. We use the

same parameters of our partitioning algorithm which is published in [91]. The following Table 15 shows all parameters with their values.

Table 15. Parameters used by the database partitioning algorithm in AutoClustC

Name	Meaning	Default Value
r	Physical read ratio threshold of a query	20%
f_n	The threshold of the percentage of queries that satisfies r	40%
f_t	Query frequency threshold: a query must occur at least f_t percent in the whole query set	0
c_α	Confidence interval	1%
α	Confidence level	95%

In order to test how the distributed partitions on the standby PM improve the whole system performance, we construct virtual environments on three PMs, PM1, PM2 and PM3, as described in section 2.1.2. In PM1 we create 5 VMs each with 2 GB RAM and Xeon E5645@2.4 GHz CPU. In PM2 we create 4 VMs each with 1 GB RAM and Xeon E5310@1.6 GHz CPU. In PM3 we create 3 VMs each with 2 GB RAM and Xeon E5645@2.4 GHz CPU. When we simulate the environment of a data center, we use the VM created on PM1, PM2 or PM3 to represent the real PM in a real data center. If we use PM* to represent the VM in our experiment, then there are totally 12 PM* (5 PM* on PM1, 4 PM* on PM2 and 3 PM* on PM3) as shown in Table 16. We measure the communication delay of each physical server pair and the result is shown in Table 17. From Table 17 we can see that the communication delay is less than 1 millisecond since all three physical servers are located in the same room and connected by a high speed Ethernet. We also know the query response time for each query type of TPC-H benchmark ranges from 1 second to 15 seconds when those queries run on our physical server. So the scaler, ρ, used in out experiment can be set to 1/1000. Such small scaler

means the communication delay almost has no impact on the selection of a proper PM. This is because according to the formula, $overload_score_PMG_k \times (1 + VecNICCD_k \times \rho)$, which is used to compute the overall weighted overload score, we know $1 + \frac{VecNICCD_k}{1000} \approx 1$ since $VecNICCD_k$ cannot be bigger than 1.

When we perform our experiments, for each PM* we run 10-15 tenants (the number of tenants is randomly generated).

Table 16. Specifications of the virtual environment

PM #	VM (PM*) #	Spec.
PM1	VM1 (PM_1*)	2 GB RAM and Xeon E5645@2.4 GHz CPU
	VM2 (PM_2*)	
	VM3 (PM_3*)	
	VM4 (PM_4*)	
	VM5 (PM_5*)	
PM2	VM6 (PM_6*)	1 GB RAM and Xeon E5310@1.6 GHz CPU
	VM7 (PM_7*)	
	VM8 (PM_8*)	
	VM9 (PM_9*)	
PM3	VM10 (PM_{10}*)	2 GB RAM and Xeon E5645@2.4 GHz CPU
	VM11 (PM_{11}*)	
	VM12 (PM_{12}*)	

Table 17. Communication delay for each server pair

	PM1	PM2	PM3
PM1	0	0.704 millisecond	0.306 millisecond
PM2	0.704 millisecond	0	0.281 millisecond
PM3	0.306 millisecond	0.281 millisecond	0

We set the tested master database instance in a PM_1*, and the other PM*s can work as the standby PM*s which may be used to distribute partitions to. The tenants on PM1* are querying their database instances with a time interval of 500-1000 milliseconds (time interval is randomly generated for each tenant). If the database instances occupies more than 75% of the CPU of the PM_1*, we say PM_1* is overloaded and the SLAs of the tenants

on PM_1* may be violated due to the heavy workload. If such case occurs the corresponding standby PM*, which has the distributed partitions for the master database instance, has to be used in order to split the workload until the CPU utilization ratio falls below 75% of the CPU of the PM_1*. We read the CPU utilization every 10 seconds from the system and the performance of the whole system is shown in Figure 23.

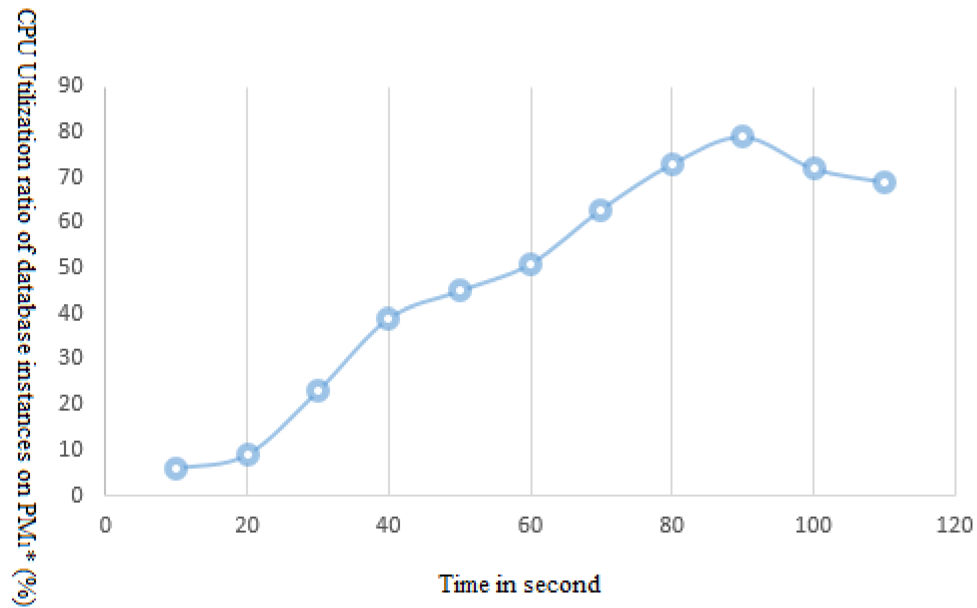


Figure 23. CPU utilization ratio of PM_1*

From Figure 23 we can see the overall CPU utilization is increasing when more tenants are querying their database instances. At 90 seconds, the CPU utilization on PM_1* is about 81%, which is above the 75% threshold. From this time on, there exists a high risk for all tenants such that the SLA might be violated under the heavy workload. So the distributed partitions of the tested database instance on the standby PM* start to be used to process the workload. That is why the CPU utilization is falling down after 90 seconds. The risk of SLA violation for all tenants on PM_1* will be reduced. In Figure 24 we can see how the overloaded PM_1* performs comparing to an un-overloaded PM_1*.

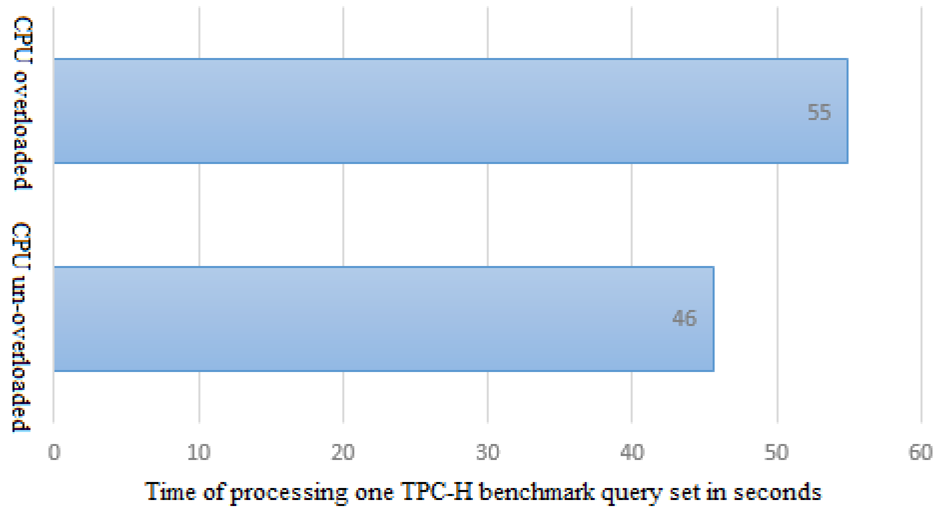


Figure 24. Average time of processing one TPC-H benchmark query set under the overloaded and un-overloaded status

From Figure 24 we can see when CPU is overloaded (at the 90th second in Figure 23), the CPU utilization is about 81%, and the average time of processing one TPC-H benchmark query set is 55 seconds. When CPU is un-overloaded (at the 110th second in Figure 23), i.e., the database instance on a standby PM* is involved to help the master database instance process the workload, the average time of processing one TPC-H benchmark query set is 46 seconds. So the time improvement is $\frac{55-46}{55} = 16.4\%$. This can reduce the SLA violation for all tenants on the PM_1*.

Chapter VI Conclusions and Future Work

In this research we have proposed a memory buffer management algorithm for cloud database, SLA-LRU, and a performance tuning algorithm for cloud database, AutoClustC.

The first algorithm, SLA-LRU, takes SLA into account and considers memory buffer page's frequency, buffer page's recency, and buffer page's value which is the cost of remove a page from the buffer pool, in order to perform buffer page replacement. SLA-LRU first will check whether the memory buffer pool is full or not. If there is no free space in the memory buffer pool, SLA-LRU then computes the SLA penalty change trend for each tenant using the pre-defined SLA penalty cost function for each tenant. After SLA-LRU identifies the tenant who has the lowest SLA penalty cost increment, the algorithm will free the corresponding tenant's memory buffer pages using a moving forward scanning method, which doubles the scanning length on the buffer page list for each iteration until the whole buffer page list is scanned or enough memory buffer pages have been freed.

The second algorithm, AutoClustC, is designed for dynamically tuning the cloud database when an SLA violation occurs by considering both resource provisioning tuning method and database partitioning tuning method. It uses an AR(2) and an ANN model l to estimate the tuning cost for database partitioning and resource provisioning respectively. Then the tuning method with the lower cost will be selected to tune the database in order to re-guarantee the performance related SLA. If database partitioning is selected, a database partitioning algorithm based on AutoClust is used to partition the database tables. Then the AutoClustC algorithm will distribute the resulting partitions to

the proper PM located in the same data center by considering the overall weighted overload score for each PM group and the selection probability of each PM in the selected PM group. The PM that has the highest selection probability in the PM group that has the least overload score is chosen to be the proper PM for partition distribution.

We have analyzed the worst-case time and space complexity of the two proposed algorithms. The time complexity of SLA-LRU is impacted by the number of entries in the buffer page list (n_b) and the space complexity of SLA-LRU is impacted by the number of tenants (n_t). The time complexity of AutoClustC is impacted by the number of entries in the historical CPU utilization data set (n_c), the number of records of the training data set for ANN (n_{tr}), the number of training iterations for ANN (n_{ti}), the number of PM groups in the data center (n_{PMG}) and the maximum number of PMs in one PM group (m_{PM}). The space complexity of AutoClustC is also impacted by the number of entries in the historical CPU utilization data set (n_c), the number of records of the training data set for ANN (n_{tr}), the number of training iterations for ANN (n_{ti}), the number of PM groups in the data center (n_{PMG}) and the maximum number of PMs in one PM group (m_{PM}).

We have also performed extensive experiments in order to study the performance of SLA-LRU and AutoClustC using the TPC-H benchmark. We have compared our SLA-LRU algorithm with two existing buffer management algorithms, LRU-2 and MT-LRU, in terms of query response time and the penalty cost improvement ratio. We have studied our AutoClustC algorithm by computing the prediction accuracy of the ANN model and the AR(2) model and comparing the performance of the new database partition results on the cloud database after a database repartitioning takes place with and without using

partition distribution. A summary of our theoretical and experimental performance evaluation results is presented in the following sections.

1. Summary of Performance Evaluation Results

1.1. Summary of the performance results for SLA-LRU

SLA-LRU is a cloud database buffer pool management algorithm based on the classic LRU-2 algorithm but takes SLA into consideration. It requires user-defined parameter, percentile of the buffer page list for the first scanning process (α), and a pre-defined SLA penalty cost function ($f_i(x)$) for each tenant i. In the dissertation we used 0.1 as the default value for α. By changing this value people can decide the scanning speed on the memory buffer page list. Below we summarize the results we have obtained so far for SLA-LRU.

1. SLA-LRU has a time complexity of O(n_b), and a space complexity of O(n_t), where n_b is the number of entries in the buffer page list and n_t is the number of tenants.
2. SLA-LRU can provide almost the same query response time as that of LRU-2 algorithm, and much better query response time compared to MT-LRU.
3. Both SLA-LRU and MT-LRU can significantly reduce the SLA penalty cost compared to LRU-2, and MT-LRU has a better penalty cost improvement ratio than that of SLA-LRU.
4. The overall performance of SLA-LRU by considering both query response time and SLA penalty cost improvement ratio is better than that of LRU-2 and MT-LRU.

1.2. Summary of performance results for AutoClustC

AutoClustC is a dynamic cloud database tuning algorithm based on the partitioning algorithm, AutoClust. So it requires the user-defined parameters used in AutoClust: 1) physical read ratio threshold of a query (*r*) (by changing this value, people can decide whether a query is a physical read mainly query or a logical read mainly query), 2) query frequency threshold (f_t) (by changing this value people can decide what queries are outlier queries), 3) confidence interval (*a*), and 4) confidence level (c_a) (by changing these two values a and c_a, people can decide how many queries have to be collected so that there are enough physical read mainly queries for re-partitioning). Besides the 4 parameters listed above, AutoClustC also needs one more user-defined parameter, time interval for the future CPU utilization forecasting (*t*), and five more system parameters, size of database (*S*), maximum number of attributes (*NA*), number of query types (*NQ*), number of users (*NU*), and the historical CPU utilization data (*U*). If the algorithm is run for the first time, a training data set for the ANN model has to be provided. Below we summarize the results we have obtained for AutoClustC.

1. AutoClustC has a time complexity of O(n_c) + O($n_{tr}n_{ti}$) + O($n_{PMG}m_{PM}$) + O($n_{PMG}{}^2$), and a space complexity of O(n_c) + O(n_{tr}) + O($n_{PMG}m_{PM}$) + O($n_{PMG}{}^2$), where n_c is the number of entries in the historical CPU utilization data set; n_{tr} is the number of records of the training data set for ANN; n_{ti} is the number of training iterations for ANN; n_{PMG} is the number of PM groups in the data center; and m_{PM} is the maximum number of PMs in one PM group.
2. AutoClustC is able to dynamically tune the cloud database when performance SLA is violated.

3. AutoClustC has high accuracy when it forecasts the costs for resource provisioning tuning method and database partitioning tuning method.
4. If the master database instance is the only database instance in the data center and database partitioning is selected as the tuning method, the new resulting partitions can provide better performance which is measured in 95th percentile query response time for each query type.
5. If a standby database instance is used and database partitioning is selected as the tuning method, the new resulting partitions distributed to the standby database instance can significantly reduce the chance of SLA violation by splitting the high volume workload on the master database instance.

2. Future Research

For future work, we plan to perform the following tasks in order to improve the weaknesses of our proposed algorithms in this dissertation.

Testing non-linear SLA penalty cost functions for SLA-LRU

For SLA-LRU, the SLA penalty cost function used in the algorithm is a step based function. We have not tested any non-linear penalty cost function for our algorithm. In the future research, we will use different non-linear functions as the SLA penalty cost functions and perform experiments using the new functions.

Dealing with dirty buffer pages for SLA-LRU

In the current SLA-LRU algorithm we did not address the issue of how to handle dirty buffer pages. If the buffer page is a dirty page we cannot simply remove the page from the buffer pool since the data in this buffer page may not be written back to disk yet. In the future research, we will conduct some research and improve the SLA-LRU algorithm

so that the advanced version of SLA-LRU can perform correct actions when doing page replacement on dirty buffer pages.

Dynamically adjusting the scanning length for SLA-LRU

In the current SLA-LRU algorithm, the most recent scanning length on the buffer page list is two times of the length of the previous scanning iteration, until the a full scan occurs on the buffer page list. One problem of such scanning method is that it has no ability to adjust buffer page releasing speed. In the future research we would like to adjust the buffer page releasing speed by dynamically changing the value of α, which is the percentile position used to separate least recent used buffer pages and most recent used buffer pages. The actual value of α can be determined by the ratio of the number of buffer pages that have been released over the number of buffer pages that are needed. Dynamically adjusting the scanning length would help SLA-LRU find enough buffer pages faster.

Testing more CPU utilization patterns for AutoClustC

For AutoClustC, the CPU utilization pattern used in the algorithm has to match some assumptions. In reality, the CPU utilization may fluctuate and not follow our assumptions, and this could cause low prediction accuracy when using our prediction algorithm. In the future research, we would like to consider more CPU utilization patterns, combine static provisioning and dynamic provisioning together, and use different statistic models to forecast the future CPU utilization according to different CPU utilization patterns.

Distributing partitions to proper PM based on different distribution levels

The current partition distribution algorithm used in AutoClustC is a same data center based algorithm. The standby PM which is used to distribute partitions has to be located

in the same data center as the PM of the master database instance. The weakness of such distribution method is that the failure of the data center will cause the service interruption of both master database instance and standby database instance. So we need to consider distributing partitions based on different distribution levels. In our future research, we will perform research on different data center distribution and different availability zone distribution. The PMs located in different data centers or different availability zones are 100% physically isolated, so the service cannot be interrupted when one data center fails.

References

[1] S. Li and L. Gruenwald, "An SLA and Operation Cost Aware Performance Re-tuning Algorithm for Cloud Databases," in *IEEE 9th International Conference on Cloud Computing*, 2016.

[2] Amazon, "Amazon SimpleDB," [Online]. Available: http://aws.amazon.com/simpledb/. [Accessed January 2017].

[3] Amazon, "Amazon DynamoDB," [Online]. Available: http://aws.amazon.com/dynamodb/. [Accessed January 2017].

[4] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes and R. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *ACM Transactions on Computer Systems*, 2008.

[5] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver and R. Yerneni, "PNUTS: Yahoo!'s Hosted Data Serving Platform," in *Proceedings of Very Large Data Bases Endowment (PVLDB)*, 2008.

[6] Amazon, "Amazon RDS," [Online]. Available: http://aws.amazon.com/rds/. [Accessed January 2017].

[7] Microsoft, "Microsoft SQL Azure," [Online]. Available: http://msdn.microsoft.com/en-us/library/windowsazure/ee336279.aspx. [Accessed January 2017].

[8] Amazon, "AWS benefits," [Online]. Available: https://aws.amazon.com/application-hosting/benefits/. [Accessed 3rd April 2017].

[9] P. Shivam, A. Demberel, P. Gunda, D. Irwin, L. Grit, A. Yumerefendi, S. Babu and J. Chase, "Automated and On-Demand Provisioning of Virtual Machines for Database Applications," in *Proc. of SIGMOD*, 2007.

[10] P. Xiong, Y. Chi, S. Zhu, J. Moon, C. Pu and H. Hacigumus, "Intelligent Management of Virtualized Resources for Database Management Systems in Cloud Environment," in *Proc. of ICDE*, 2011.

[11] D. Gmach, S. Krompass, A. Scholz, M. Wimmer and A. Kemper, "Adaptive Quality of Service Management for Enterprise Services," in *ACM Transactions*, 2008.

[12] S. Tozer, T. Brecht and A. Aboulnaga, "Q-cop: Avoiding Bad Query Mixes to Minimize Client Timeouts Under Heavy Loads," in *Proc. of ICDE*, 2010.

[13] S. Agrawal, V. Narasayya and B. Yang, "Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design," in *ACM International Conference on Management of Data (SIGMOD)*, 2004.

[14] A. Silberschatz, H. Korth and S. Sudarshan, Database System Concepts, 6th ed., McGraw-Hill, 2011.

[15] S. Navathe, S. Ceri, G. Wierhold and J. Dou, "Vertical Partitioning Algorithms for Database Design," in *ACM Transactions on Database Systems*, 1984.

[16] W. T. McCormick, S. P.J. and W. T.W., "Problem Decomposition and Data Reorganization by A Clustering Technique," in *Operation Research*, 1972.

[17] H. Yu, N. Powell, D. Stembridge and X. Yuan, "Cloud Computing and Security Challenges," in *ACM-SE Proceedings of the 50th Annual Southeast Regional Conference*, 2012.

[18] N. Bobroff, A. Kochut and K. Beaty, "Dynamic Placement of Virtual Machines for Managing SLA Violations," in *IFIP/IEEE International Symposium on Integrated Network Management*, 2007.

[19] D. She and J. Hellerstein, "Predictive Models for Proactive Network Management: Application to A Production Web Server," in *Proc. of the IEEE/IFIP Network Operations and Management Symposium*, 2000.

[20] S. Guinepain and L. Gruenwald, "Using Cluster Computing to Support Automatic and Dynamic Database Clustering," in *International Workshop on Automatic Performance Tuning (IWAPT)*, 2008.

[21] Database.com, "Database.com," [Online]. Available: http://www.database.com. [Accessed January 2017].

[22] Google, "Google Cloud SQL," [Online]. Available: http://code.google.com/apis/sql. [Accessed January 2017].

[23] Oracle, "Oracle Database Cloud Service.," [Online]. Available: http://cloud.oracle.com. [Accessed January 2017].

[24] E. J. O'neil, P. E. O'neil and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," in *ACM SIGMOD Record*, 1993.

[25] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," in *IBM Sys. J.*, 1970.

[26] H. T. Chou and D. J. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," in *Algorithmica*, 1986.

[27] N. Megiddo and D. S. Modha, "Arc: A Self-Tuning, Low Overhead Replacement Cache," in *FAST*, 2003.

[28] S. Bansal and D. S. Modha, "CAR: Clock with Adaptive Replacement," in *FAST*, 2004.

[29] U. Cesana and Z. He, "Multi-Buffer Manager: Energy-Efficient Buffer Manager for Databases on Flash Memory," in *ACM Trans. Embedded Computing Systems*, 2010.

[30] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala and S. Chaudhuri, "Sharing Buffer Pool Memory in Multi-Tenant Relational Database-as-a-Service," in *VLDB Endowment*, 2015.

[31] Microsoft, "SQL Server Buffer Manager," [Online]. Available: https://technet.microsoft.com/en-us/library/ms189628.aspx. [Accessed January 2017].

[32] T. Tannenbaum, K. Wright, K. Miller and M. Livny, Condor: A Distributed Job Scheduler, Cambridge, MA: MIT Press, 2001, pp. 307-350.

[33] I. Altair Engineering, "PBS Open Source Project," [Online]. Available: http://www.pbspro.org/. [Accessed January 2017].

[34] K. Kim, W. Wang, Y. Qi and M. Humphrey, "Empirical Evaluation of Workload Forecasting Techniques for Predictive Cloud Resource Scaling," in *IEEE International Conference on Cloud Computing*, 2016.

[35] T. Wood, P. J. Shenoy, A. Venkataramani and M. S. Yousif, "Black-Box and Gray-Box Strategies for Virtual Machine Migration," in *NSDI*, 2007.

[36] S. Cunha, J. M. Almeida, V. Almeida and M. Santos, "Self-Adaptive Capacity Management for Multi-Tier Virtualized Environments," in *Integrated Network Management*, 2007.

[37] A. Floratou, J. M. Patel, W. Lang and A. Halverson, "When Free Is Not Really Free: What Does It Cost to Run A Database Workload in The Cloud," in *TPCTC*, 2011.

[38] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant and K. Salem, "Adaptive Control of Virtualized Resources in Utility Computing Environments," in *EuroSys*, 2007.

[39] E. Cecchet, R. Singh, U. Sharma and P. J. Shenoy, "Dolly: Virtualization-Driven Database Provisioning for The Cloud," in *VEE*, 2011.

[40] J. Rogers, O. Papaemmanouil and U. Çetintemel, "A Generic Auto-Provisioning Framework for Cloud Databases," in *ICDE Workshops*, 2010.

[41] U. Sharma, P. J. Shenoy, S. Sahu and A. Shaikh, "A Cost-Aware Elasticity Provisioning System for The Cloud," in *ICDCS*, 2011.

[42] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff and D. Zagorodnov, "The Eucalyptus Open-Source Cloud-Computing System," in *Proceedings of 9th IEEE/ACM International Symposium om CLuster Computing and The Grid*, Shanghai, 2009.

[43] OpenStack, "Open Source Cloud Computing Software," [Online]. Available: https://www.openstack.org/. [Accessed January 2017].

[44] C. S. Yeo, R. Buyya, M. Dias de Assuncao, J. Yu, A. Sulistio, S. Venugopal and M. Placek, "Utility Computing on Global Grids," in *Handbook of Computer Networks*, Hoboken, NJ: Wiley Press, 2008.

[45] J. Shneidman, C. Ng, D. C. Parkes, A. AuYoung, A. C. Snoeren, A. Vahdat and B. N. Chun, "Why Markets Could (But Don't Currently) Solve Resource Allocation Problems in Systems," in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.

[46] W. W. Chu and I. Ieong, "A Transaction-Based Approach to Vertical Partitioning for Relational Database Systems," in *IEEE Transactions on Software Engineering*, 1993.

[47] E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms, Rockville, 1978.

[48] S. Navathe and M. Ra, "Vertical Partitioning for Database Design: A Graph Algorithm," in *ACM SIGMOD International Conference*, 1989.

[49] E. S. Abuelyaman, "An Optimized Scheme for Vertical Partitioning of a Distributed Database," in *International Journal of Computer Science and Network Security (IJCSNS)*, 2008.

[50] S. Papadomanolakis, D. Dash and A. Ailamaki, "Efficient Use of the Query Optimizer for Automated Physical Design," in *Proceedings of the 33rd VLDB International Conference*, 2007.

[51] J. Rao, C. Zhang, N. Megiddo and G. M. Lohman, "Automating Physical Database Design in a Parallel Database," in *SIGMOD*, 2002.

[52] N. Pasquier, Y. Bastidem, R. Taouil and L. Lakhal, "Efficient Mining of Association Rules Using Closed Item set Lattices," in *Information Systems*, 1999.

[53] L. Rodriguez, X. Li, D. A. Cuevas-Rasgado and F. Garcia-Lamont, "DYVEP: An Active Database System with Vertical Partitioning Functionality," in *Networking, Sensing and Control (ICNSC) of 10th IEEE International Conference*, 2013.

[54] A. Jindal and J. Dittrich, "Relax and Let the Database do the Partitioning Online," in *Business Intelligence for Real Time Enterprise (BIRTE)*, 2011.

[55] S. Pukdesree, V. Lacharoj and P. Sirisang, "Performance Evaluation of Distributed Database on PC Cluster Computers," in *World Congress on Engineering and Computer Science (WCECS)*, 2010.

[56] S. Das, D. Agrawal and A. E. Abbadi, "ElasTraS: An Elastic Transactional Data Store in The Cloud," in *USENIX Hot Cloud*, 2009.

[57] C. Garcia-Alvarado, V. Raghavan, S. Narayanan and F. M. Waas, "Automatic Data Placement in MPP Databases," in *Data Engineering Workshops (ICDEW) of IEEE 7th International Conference on Self Managing Database Systems (SMDB)*, 2012.

[58] R. Amossen, "Vertical Partitioning of Relational OLTP Databases Using Integer Programming," in *Data Engineering Workshops (ICDEW) of IEEE 5th International Conference on Self Managing Database Systems (SMDB)*, 2010.

[59] M. Goli and S. M. TaghiRouhaniRankoohi, "A New Vertical Fragmentation Algorithm Based on Ant Collective Behavior in Distributed Database System," in *Knowledge and Information Systems*, 2012.

[60] J. L. Deneubourg, S. Goss, N. Franks, A. S. Franks, C. Detrain and L. Chretien, "The Dynamics of Collective Sorting: Robot-Like Ants and Ant-Like Robots.," in *Proceedings of the First International Conference on Simulation of Adaptive Behaviour: From Animals to Animats*, 1991.

[61] M. R. Garey and D. S. Johnson, "A Guide to The Theory of NP-Completeness," in *Computers and Intractability*, W.H. Freeman, 1979.

[62] C. H. Cheng, W. K. Lee and K. F. Wong, "A genetic Algorithm Based Clustering Approach for Database Partitioning," in *IEEE Transactions on System, Man and Cybernetics*, 2002.

[63] D. Campbell, G. Kakivaya and N. Ellis, "Extreme Scale with Full SQL Language Support in Microsoft SQL Azure," in *SIGMOD*, 2010.

[64] P. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manner, L. Novik and T. Talius, "Adapting Microsoft SQL Server for Cloud Computing," in *ICDE*, 2011.

[65] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan and N. Zeldovich, "A Database-as-a-Service for The Cloud," in *The 5th Biennial Conference on Innovative Data Systems Research*, 2011.

[66] C. Curino, Y. Zhang, E. P. C. Jones and S. Madden, "Schism: A Workload-Driven Approach to Database Replication and Partitioning," in *VLDB*, 2010.

[67] L. Lim, "Elastic Data Partitioning for Cloud-based SQL Processing Systems," in *IEEE International Conference on Big Data*, 2013.

[68] U. Das, D. Agrawal and A. D. Abbadi, "ElasTraS: An Elastic, Scalable, and Self-Managing Transactional Database for The Cloud," in *ACM Transactions on Database Systems (TODS)*, 2013.

[69] K. A. Kumar, A. Quamar, A. Deshpande and S. Khuller, "SWORD: Workload-Aware Data Placement and Replica Selection for Cloud Data Management Systems," *VLDB Journal,* vol. 23, pp. 845-870, 2014.

[70] B. Sauer and W. Hao, "Horizontal Cloud Database Partitioning with Data Mining Techniques," in *12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, 2015.

[71] Amazon, "Amazon EC2 Purchasing Options," [Online]. Available: https://aws.amazon.com/ec2/purchasing-options/. [Accessed July 2016].

[72] Amazon, "Amazon EC2 Instance Types," [Online]. Available: https://aws.amazon.com/ec2/instance-types/. [Accessed July 2016].

[73] Amazon, "Amazon EC2 T2 instance," [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-instances.html. [Accessed January 2017].

[74] R. S. Shariffdeen, D. T. S. P. Munasinghe, H. S. Bhathiya, U. K. J. U. Bandara and H. M. N. D. Bandara, "Workload and Resource Aware Proactive Auto-Scaler for PaaS Cloud," in *IEEE 9th International Conference on Cloud Computing*, 2016.

[75] MySQL, "MySQL GitHub site," [Online]. Available: https://github.com/mysql/mysql-server. [Accessed January 2017].

[76] G. E. P. Box and G. M. Jenkins, Time Series Analysis: Corecasting and Control, San Francisco: Holden Day, 1976, p. 575.

[77] W. McCulloch and P. Walter, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics,* vol. 5, no. 4, p. 115–133, 1943.

[78] M. Zaki and C. J. Hsiao, "Efficient Algorithms for Mining Closed Item Sets and Their Lattice Structure," in *IEEE Trans Knowl Data Eng*, 2005.

[79] T. Uno, M. Kiyomi and H. Arimura, "LCM Ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets," in *Proc. of the IEEE ICDM workshop on frequent itemset mining implementations*, Brighton, UK, 2004.

[80] G. Zhang, B. E. Patuwo and M. Y. Hu, "Forecasting with Artificial Neural Networks: The State of The Art," *Int. J. Forecast,* vol. 14, pp. 35-62, 1998.

[81] MathWorks, "Description of TRAINGDX function," [Online]. Available: http://www.mathworks.com/help/nnet/ref/traingdx.html. [Accessed January 2017].

[82] MathWorks, "Description of LEARNGDM function," [Online]. Available: http://www.mathworks.com/help/nnet/ref/learngdm.html. [Accessed January 2017].

[83] Amazon, "AWS re:Invent 2014," [Online]. Available: https://gigaom.com/2014/11/12/amazon-details-how-it-does-networking-in-its-data-centers/. [Accessed 10 April 2017].

[84] A. Singh, M. Korupolu and D. Mohapatra, "Server-Storage Virtualization: Integration and Load Balancing in Data Centers," in *ACM/IEEE Conf. of Supercomputing*, 2008.

[85] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik and M. Handley, "Improving Datacenter Performance and Robustness with Multipath TCP," in *ACM SIGCOMM Computer Communication Review*, 2011.

[86] M. Newman, Networks: An Introduction, Oxford: Oxford University Press, 2009.

[87] M. Ali, K. Bilal, S. Khan, B. Veeravalli, K. Li and A. Zomaya, "Drops: Division and Replication of Data in Cloud for Optimal Performance and Security," *IEEE Transactions on Cloud Computing,* 2015.

[88] TPC, "TPC-H benchmark," [Online]. Available: http://www.tpc.org/tpch/default.asp. [Accessed January 2017].

[89] W. Gilbert, "On Periodicity in Series of Related Terms," *Proceedings of the Royal Society of London,* vol. 131, pp. 518-532, 1931.

[90] M. a. H. M. Mao, "A Performance Study on The VM Startup Time in The Cloud," in *the IEEE 5th International Conference on Cloud Computing*, 2012.

[91] L. Li and L. Gruenwald, "Self-Managing Online Partitioner for Databases (SMOPD) – A Vertical Database Partitioning System with a Fully Automatic Online Approach," in *International Database Engineering & Applications Symposium(IDEAS)*, 2013.

[92] MySQL, "MySQL Server 5.5 Page Replacement Source Code," [Online]. Available: https://github.com/mysql/mysql-server/blob/5.5/storage/innobase/buf/buf0lru.c. [Accessed January 2017].