UMinho|2016

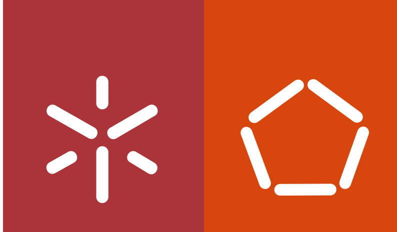
Clayton Maciel Costa **Efficient adaptive query processing on large database systems available in the cloudenvironment**

Universidade do Minho
Escola de Engenharia

Clayton Maciel Costa

Efficient adaptive query processing on large database systems available in the cloud environment

julho de 2016



Universidade do Minho
Escola de Engenharia

Clayton Maciel Costa

Efficient adaptive query processing on large database systems available in the cloud environment

Tese de Doutoramento em Informática

Trabalho realizado sob a orientação do
Professor Doutor António Luís Pinto Ferreira de Sousa

julho de 2016

DECLARAÇÃO

Nome: Clayton Maciel Costa

Número do Bilhete de Identidade: FH625844

Endereço de correio eletrônico: clayton.maciel@ifrn.edu.br

Título da tese: Efficient adaptive query processing on large database systems available in the cloud environment

Orientador: Professor Doutor António Luís Pinto Ferreira de Sousa

Ano de conclusão: 2016

Designação do Doutoramento: Doutoramento em Informática

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE.

Universidade do Minho, / /2016

Assinatura

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração da presente tese. Confirmo que em todo o trabalho conducente à sua elaboração não recorri à prática de plágio ou a qualquer forma de falsificação de resultados.

Mais declaro que tomei conhecimento integral do Código de Conduta Ética da Universidade do Minho.

Universidade do Minho, / /2016

Nome completo: Clayton Maciel Costa

Assinatura: Clayton Maciel Costa

"A man's mind is stretched by a new idea or sensation,
and never shrinks back to its former dimensions"

Oliver Wendell Holmes

Acknowledgments

First, I would like to thank God for letting me reach this goal.

Many people directly or indirectly contributed to the development of this work in different ways and moments.

I would like to acknowledge my Ph.D. Advisor, Professor António Luís Sousa, from the Department of Informatics at the University of Minho, for his wise supervision, great wisdom, guidance, extensive knowledge and experience. His guidance played a major role in the development of this work and the good results obtained is a testimony of that. This research work showed us that many problems are still unsolved and many questions are unanswered. Some solutions are still under development in order to obtain answers and therefore, many topics are still open to further discussion.

I also would like to acknowledge my Ph.D. Advisor, Professor Cícilia Raquel Maia Leite. It was a privilege to collaborate with her during these years of research work, projects, publications and international presentations. I am particularly grateful for her guidance in this wonderfull experience, and also her trust, friendship and scientific support.

I would like to further thank the Federal Institute of Rio Grande do Norte for the financial support in my qualification. In particular, I am very grateful to MSc. Professsor Jose Yvan who established a technical cooperation between the institutions under which this research work was developed.

I also would like to thank the Federal Institute of Rio Grande do Norte/Campus Ipanguaçu (IFRN/IP) for providing the physical infrastructure to develop the work reported in this thesis. Furthermore, I am also very grateful the High-Assurance Software Lab/INESC TEC of the University of Minho for providing the required physical infrastructure to work while I was in Portugal. Both institutes provided all the necessary environments to develop the experimental trials which were carried out, including the use of existent and acquired equipment and also the use of specific components required for accomplishing the goals of this research.

I would like to thank my friends of the Department of Informatics (DI-IFRN/IP) for the support of academic works.

I am very grateful to all my friends of IFRN/IP and all those who were present in this important step of my life.

I wish to thank my spouse, Ceiça, and my son, Pedro Davi, for long hours of absence and for their unconditional support. With them I shared a multitude of joy, sorrow, anxiety and hope throughout this challenging work. I am very grateful for believing in me, always encouraging me and for always being by my side in the hardest times.

At last, not least... I wish to thank my parents. My dear mother, for her love, dedication, life lessons and courage. My dear father, for his constant concerns and financial

investment throughout my entire life. This research work and the degree itself are dedicated to them. I am sure that they are proud of me for achieving this goal as much as I am imensely proud of them. Throughout the years, they have made all possible efforts and sacrifices to give the best to their sons. I am what I am today thanks to them.

To all, my sincere thanks.

Efficient adaptive query processing on large database systems available in the cloud environment

Abstract

Nowadays, many companies are migrating their applications and data to cloud service providers, mainly because of their ability to answer quickly to business requirements. Thereby, the performance is an important requirement for most customers when they wish to migrate their applications to the cloud.

Therefore, in cloud environments, resources should be acquired and released automatically and quickly at runtime. Moreover, the users and service providers expect to get answers in time to ensure the service SLA (Service Level Agreement). Consequently, ensuring the QoS (Quality of Service) is a great challenge and it increases when we have large amounts of data to be manipulated in this environment.

To resolve this kind of problems, several researches have been focused on shorter execution time using adaptive query processing and/or prediction of resources based on current system status. However, they present important limitations. For example, most of these works does not use monitoring during query execution and/or presents intrusive solutions, i.e. applied to the particular context.

The aim of this thesis is the development of new solutions/strategies to efficient adaptive query processing on large databases available in a cloud environment. It must integrate adaptive re-optimization at query runtime and their costs are based on the SRT (Service Response Time – SLA QoS performance parameter). Finally, the proposed solution will be evaluated on large scale with large volume of data, machines and queries in a cloud computing infrastructure.

Finally, this work also proposes a new model to estimate the SRT for different request types (database access requests). This model will allow the cloud service provider and its customers to establish an appropriate SLA relative to the expected performance of the services available in the cloud.

Keywords: cloud computing; service level agreement; performance; service response time

Processamento eficiente adaptativo de consultas em grandes bases de dados disponíveis em ambiente de nuvem

Resumo

Atualmente, muitas companhias têm migrado suas aplicações e dados para fornecedores de serviços em nuvem, pois um dos principais benefícios dessa tecnologia é a capacidade de responder rapidamente às necessidades do negócio. Assim, o desempenho é um dos mais importantes requisitos para a maioria dos clientes que desejam migrar suas aplicações para a nuvem.

Em ambiente de nuvem, os recursos devem ser adquiridos e libertados automaticamente e rapidamente em tempo de execução. Além disso, os utilizadores e fornecedores de serviços esperam sempre garantir o contrato SLA (Acordo de Nível de Serviço). Consequentemente, garantir o QoS (Qualidade de Serviço) é um grande desafio, que se torna mais complexo quando existe uma grande quantidade de dados a serem manipulados neste ambiente.

Para resolver estes tipos de problemas, diversas pesquisas têm sido realizadas focando o menor tempo de execução dos pedidos do utilizador na nuvem usando técnicas de processamento adaptativo de consultas e/ou utilizando técnicas de predição de recursos baseados no estado atual do sistema. Contudo, esses trabalhos apresentam limitações importantes. Por exemplo, a maioria desses trabalhos não utiliza monitorazação durante a execução da consulta e/ou apresenta soluções intrusivas, isto é, aplicadas a um contexto particular.

Portanto, o objetivo desta tese consiste no desenvolvimento de uma nova solução/estratégia para o processamento eficiente (adaptativo) de consultas sobre grandes bases de dados disponíveis em ambiente de nuvem. Ela irá integrar técnicas de otimização adaptativas em tempo de execução da consulta e seus custos são baseados no SRT (Tempo de Resposta do Serviço – parâmetro QoS de desempenho do SLA). A solução proposta será avaliada em larga escala utilizando uma grande base de dados, máquinas e consultas em um ambiente real de computação na nuvem.

Finalmente, este trabalho também propõe um novo modelo para estimar o SRT para diferentes tipos de pedidos (pedidos de acesso a banco de dados). Este modelo permitirá que um fornecedor de serviços em nuvem e seus clientes possam estabelecer um contrato SLA adequado, relativo ao desempenho esperado dos serviços disponíveis em nuvem.

Palavras-chave: computação em nuvem; acordo de nível de serviço; desempenho; tempo de resposta do serviço.

Index

Acknowledgments vii
Abstract ix
Resumo xi
List of acronyms xvi
List of figures xvi
List of tables xviii
Chapter 1 – Introduction 19
1.1 Contextualization 21
1.2 Motivation 22
1.3 Research objectives 24
1.4 Contributions 25
1.5 List of publications from this work 26
1.6 Thesis outline 27
Chapter 2 – State of art 29
2.1 Introduction 31
2.2 Data warehouse and OLAP 31
2.2.1 Multidimensional modeling of data warehouses 34
2.2.2 OLAP applications 35
2.3 SLA in cloud computing 37
2.3.1 Definition and lifecycle of a SLA 37
2.3.2 QoS parameters of a SLA 39
2.3.3 QoS performance parameters of a SLA 43
2.4 Data processing 46
2.4.1 Query processing in database 46
2.4.2 Adaptive query processing 48
2.4.3 Query processing in cloud 50
2.5 Related works 51
2.6 Conclusion 56
Chapter 3 – Service response time measurement model of service level agreements 57
3.1 Introduction 59
3.2 Request definition 59
3.2.1 Type 1 requests: select-range and select-aggregation 61
3.2.2 Type 2 requests: select-joins 61
3.2.3 Type 3 requests: select-sets-grouping-nesting-ordering 62

3.3 Service response time measurement model of service level agreements 63
3.3.1 Recommended SRT definition 64
3.3.2 SRT measurement model 65
3.4 Case study – validation and results 69
3.4.1 Experimental environment 69
3.4.2 Methodology.. 69
3.4.3 Used requests 71
3.4.4 Results... 74
3.4.5 Analysis of results... 78
3.5 Conclusion.. 80
Chapter 4 – Efficient adaptive query processing on large database systems available in the cloud environment.. 81
4.1 Introduction 83
4.2 Estimated cost model... 83
4.3 Architecture 86
4.4 SiclopDB framework – components.. 88
4.4.1 MetaData and performance 88
4.4.2 Dynamic query optimizer (DQO).. 90
4.4.3 Dynamic query scheduler (DQS) 92
4.4.4 Dynamic query monitoring (DQM) 103
4.5 Conclusion... 105
Chapter 5 – Experimental evaluation: validation and results 107
5.1 Introduction 109
5.2 Experimental environment.. 109
5.3 Methodology 109
5.4 Used requests 111
5.5 Results and analysis 114
5.5.1 Type 1 requests... 114
5.5.2 Type 2 requests... 120
5.5.3 Type 3 requests... 122
5.5.4 All type of requests 125
5.6 Conclusion... 126
Chapter 6 – Conclusion.. 127
6.1 Final considerations 129
6.2 Future work 130
References.. 133

Annex **141**
Annex A1 – Type 1 requests 143
Annex A2 – Type 2 requests 151
Annex A3 – Type 3 requests 155
Annex A4 – paper 1 – (2013) 163
Annex A5 – paper 2 – (2015) 165
Annex A6 – paper 3 – (2016) 171

List of acronyms

AMI – Amazon Machine Image

COS – CPU Overload Simulator

CSMIC – Cloud Service Measurement Index Consortium

DBMS – Database Management System

DOS – Disk I/O Overload Simulator

DP – Disk Performance

DQM – Dynamic Query Monitoring

DQO – Dynamic Query Optimizer

DQS – Dynamic Query Scheduler

DW – Data Warehouse

EBS – Elastic Block Store

ETL – Extract-Transform-Load

IAAS – Infrastructure as a Service

KPI – Key Performance Indicator

OLAP – On-Line Analytical Processing

OpenMP – Open Multi-Processing

PP – Processor Performance

QoS – Quality of Service

RSRT – Recommended Service Response Time

SLA – Service Level Agreement

SMI – Service Measurement Index

SQL – Structured Query Language

SRT – Service Response Time

TTP – trusted third party

VM – Virtual Machine

List of figures

Figure 2-1. Data Warehouse Architecture. 32

Figure 2-2. Database of a Company: (a) Relational Scheme of Human Resources Sector and (b) Relation Scheme of Sales Sector. 33

Figure 2-3. Example of Materialized View in Data Warehouse. 33

Figure 2-4. Star Model of Materialized_View Fact. 34
Figure 2-5. Fact: Profit Employee. 35
Figure 2-6. Fact: Top Selling Products. 35
Figure 2-7. Example of Drill-down and Roll-up Operation using OLAP Applications. 36
Figure 2-8. SLA Lifecycle. 37
Figure 2-9. Parameters and Sub-parameters defined in the SMI (Siegel & Perdue, 2012). 43
Figure 2-10. Query Processing in the Cloud. 51
Figure 3-1. Request-response communication of the client-server computing model. 60
Figure 3-2. Steps to obtain the Recommended SRT. 66
Figure 3-3. SRT Calculator – GUI Interface. 68
Figure 3-4. Methodology of experiments to obtain the Recommended SRT. 70
Figure 3-5. Processor Status through sysstat tool. 70
Figure 3-6. Disk Read/Write Status through dstat tool. 71
Figure 3-7. SRT averages on all VMs for type 1 requests. 75
Figure 3-8. SRT averages on all VMs for type 2 requests. 76
Figure 3-9. SRT averages on all VMs for type 3 requests. 77
Figure 3-10. Recommended SRT Result. 79
Figure 4-1. Ideal Computational Cost: Computation Cost (x10) vs Time (seconds). 86
Figure 4-2. SiclopDB Framework Architecture. 88
Figure 4-3. Flowchart of query processing in SiclopDB framework. 92
Figure 5-1. Methodology of experiments of SiclopDB framework. 110
Figure 5-2. Type 1 Requests (Select-Range): average virtual machines used for workloads uniformly arriving every 30 seconds for the Recommended SRTs: 80, 100 and 120 seconds. 115
Figure 5-3. Type 1 Requests (Select-Range): average virtual machines used for workloads randomly arriving between 10 and 60 seconds for the Recommended SRTs: 80, 100 and 120 seconds. 116
Figure 5-4. Type 1 Requests (Select-Aggregation): average virtual machines used for workloads uniformly arriving every 30 seconds for the Recommended SRTs: 80, 100 and 120 seconds. 118
Figure 5-5. Type 1 Requests (Select-Aggregation): average virtual machines used for workloads randomly arriving between 10 and 60 seconds for the Recommended SRTs: 80, 100 and 120 seconds. 119
Figure 5-6. Type 2 Requests: average virtual machines used with workloads uniformly arriving every 30 seconds for the Recommended SRTs: 130, 150 and 180 seconds. 121
Figure 5-7. Type 2 Requests: average virtual machines used with workloads randomly arriving between 10 and 60 seconds for the Recommended SRTs: 130, 150 and 180

seconds. 122

Figure 5-8. Type 3 Requests: average virtual machines used with workloads uniformly arriving every 30 seconds for the Recommended SRTs: 800, 1000 and 1200 seconds. . 124

Figure 5-9. Type 3 Requests: average virtual machines used with workloads randomly arriving between 10 and 60 seconds for the Recommended SRTs: 800, 100 and 1200 seconds. 125

Figure 5-10. All Type Requests: average virtual machines used with workloads uniformly arriving every 30 seconds and randomly arriving between 10 and 60 seconds. .. 126

List of tables

Table 2-1. Characteristics of related work. 55

Table 3-1. Recommended SRT Result 78

Chapter 1 – Introduction

The amounts of data generated by new technologies is increasing every day. With this growth, also increases the challenge to manage, manipulate, store and query these data. To address these challenges, a solution is to provide computing as a service, currently known as "Cloud Computing". In this way, this chapter presents a contextualization of the problem, motivation, objectives, contributions and list of publication of this thesis.

1.1 Contextualization

The cloud computing facilitates access to services and computer resources, independently of platform and architecture. Moreover, it provides users with the idea of infinite computing resources and data storage. However, as well as utility computing, all its architectural structure is on-demand and pay based on usage, i.e. pay only when it matters.

In the cloud environment, the infrastructure, the platform and the application services are available on demand and they should be available, whenever requested, for access anywhere in the world (Coutinho, de Carvalho Sousa, Rego, Gomes, & de Souza, 2015). Whereas the dimension and heterogeneity of data stored, in general, are very large. Thus, the systems efficiency and scalability become necessary to ensure the availability or release of resources for each request from users.

Given the rapid growth of the amount of data due to technological advances, to manage such massive amount of information becomes a challenging problem. The use of cloud computing platforms allows new conceptions of management and manipulation of data, because in a cloud environment, resources can be acquired and released automatically, quickly and elastic at runtime (Das, Agarwal, Agrawal, & El Abbadi, 2013).

In the cloud computing model, the service providers' objective is to optimize their profit while servicing several customers. This is obtained recurring to some level of abstraction (virtualization) according to the type of service, such as: storage, processing, bandwidth and active user accounts. To ensure the QoS (Quality of Service) there is a SLA (Service Level Agreement) associated to the service delivery. The SLA is a formal contract defined between a cloud service provider and its customers that define the level of service expected from the service provider. They are output-based and their purpose is specifically to define what the customer will receive. Therefore, it provides QoS parameters on the levels of availability, functionality, performance, penalties, billing etc (Emeakaroha et al., 2012; Garg, Versteeg, & Buyya, 2013).

To ensure the QoS parameters new challenges arise due to high heterogeneity and dynamicity of clouds. For example, new QoS parameters need to be measured and the provisioning of resources, service delivery and monitoring need to be automated and the dynamic reallocation of resources must be decentralized and global (Wu & Buyya, 2010; Wu, Garg, & Buyya, 2011). Furthermore, the same QoS parameter can have different definitions between service providers.

Considering that, the performance is an important requirement for most customers when they migrate their applications to the cloud. The SRT (Service Response Time) QoS parameter measures the total time between the time that a request arrives at the cloud provider and the time that it completes its execution. It is one of the best execution efficiency indicators of a request, allowing to know how fast a service can execute, and is the main QoS parameter used in this thesis.

The measuring of SRT parameter in the cloud is a very complex task because it depends on many system variables, such as request type, database model and current system performance (Schad, Dittrich, & Quiané-Ruiz, 2010). Furthermore, it is common in a cloud environment that the requests rate is highly unpredictable. Therefore, guaranteeing a specific response time for any level of request rate is regarded as a significant challenge to the paradigm of cloud computing. Moreover, the growth of data stored in the cloud makes this challenge ever harder.

1.2 Motivation

Nowadays, many companies have migrated their applications and data to the cloud due to the benefits of this technology (Zhou et al., 2014). For example, the applications and data stored in the cloud can be accessed anywhere. Another important benefit is the significant reduction of costs and time of experimentation and development when compared with local infrastructures because it eliminates the need of one or more physical servers in company premises, thus minimizing the electricity cost and the necessity of specialists for repairs.

Moreover, the cloud platforms are substantially scalable, which is highly beneficial for the ever-fluctuating storage needs of the IT environment. Before the cloud era, companies were struggling with their storage needs and wasting time upgrading servers. But with the advent of cloud computing, expanding storage needs are no more an issue as every change is managed on the spot.

This ability to answer quickly to business requirements is one of the major motivations for companies to migrate their applications and data to the cloud. According to CDW's Cloud Computing Tracking Poll (Ray, 2012), 84% of organizations are using at least one cloud application and 76% of

small business cloud users say they have reduced the cost of applications moved to the cloud, saving an average of 24% annually. Furthermore, according to (Larkin & Rose, 2015), companies around the world must increase their investments in cloud computing projects about 40% over the previous year.

According to (Mangard & Poschmann, 2015), cloud computing and virtualization is popular more than ever. Companies like Microsoft, Google, Amazon, IBM, Oracle, Rackspace and many others are investing billions of dollars trying to get a foothold in this new area of lucrative business. This rapid increase in the number of cloud service providers is directly related to the emergence of server-less companies like Netflix, Instagram, Pinterest, Snapchat and many others that are using commercial cloud infrastructure.

Given this context, consider, for example, an institution/company/authority that wishes to migrate their OLAP applications to a cloud service provider, with the objective of allocate computing resources on-demand and ensure the Service Response Time (SRT QoS Parameter). Moreover, in the migration process it is not interesting for the company to change the data structure. In this case, an elastic solution becomes necessary to ensure the quality of services offered.

A solution is to use adaptive query processing. It has the ability to dynamically and automatically allocate or release resources (elasticity of resources) at query runtime. This technique is very important when statistical information about the services available may be minimal and the availability of physical resources may change. This is a typical scenario of cloud environments. However, traditional and adaptive query optimzers' main objective is to reduce response time. Moreover, in the context of cloud computing, users and providers of services expect to get answers in time to ensure the service SLA.

According to (Iqbal, Dailey, & Carrera, 2009), from the user's point of view, this SRT parameter is considered one of the mains QoS parameters. However, nowadays, the major cloud providers like Amazon ("AWS EC2 Service Level Agreement," 2015, "AWS S3 Service Level Agreement," 2015) and Google (Sanderson, 2012) only emphasize on CPU availability and cost measure. Therefore, the SRT parameter is not handled in SLA due to its complexity.

In the literature, several works have been focused in development of techniques and algorithms for efficient query processing to ensure the SRT parameter (Alves, Bizarro, & Marques, 2011; Amazon Web Services, 2015; Cervino, Kalyvianaki, Salvachua, & Pietzuch, 2012; Chi, Moon, Hacigümüş, & Tatemura, 2011; Coelho da Silva, Nascimento, de Macêdo, Sousa, & Machado, 2012, 2013;

Curino, Jones, Madden, & Balakrishnan, 2011; Dean & Ghemawat, 2008a, 2008b; Guitart, Carrera, Beltran, Torres, & Ayguadé, 2008; Kllapi, Sitaridi, Tsangaris, & Ioannidis, 2011; Mian, Martin, & Vazquez-Poletti, 2013; Rogers, Papaemmanouil, & Cetintemel, 2010; Sharma, Shenoy, Sahu, & Shaikh, 2010, 2011; Vigfusson, Silberstein, Cooper, & Fonseca, 2009)

However, as presented in Chapter 2, these works present elasticity and/or scalability limitations in their algorithms. Moreover, many solutions are not adaptive, intrusive and/or they do not use formal definition of their services. Therefore, it is necessary to develop new methods, techniques and tools that allow a service to ensure suitably to the SRT parameter, which is one of the main aims of this work.

1.3 Research objectives

The objective of this thesis consists in development of a new solution to efficient query processing on large databases available in a cloud environment. It integrates adaptive re-optimization at query runtime using costs based on the SRT QoS parameter. This work focuses on OLAP applications because in this kind of environment the adaptive processing produces positive effects on query runtime. Based on these premises, the following specific aims and goals need to be achieved:

(i) **Objective 1**: Analyze the methods, techniques and tools for efficient query processing in the cloud. In this way, it is required to understand the methods and techniques of traditional and adaptive queries processing on databases systems in centralized, parallel and distributed environments. Moreover, it is necessary to understand the techniques of query processing in data warehouse and OLAP applications and the techniques for query processing and optimization in the cloud. Finally, it is necessary to research in papers, tutorials, technical reports and technologies for comprehension of the state of the art and related works of this thesis.

(ii) **Objective 2**: Development of a model to estimate the Recommended SRT. For this, it is necessary to understand the requirements for good performance of queries in a cloud environment. The query processing in the cloud comprises a series of challenges to be overcome, including, scalability, performance and availability of services, self-management, data security and the quality assurance of the data service (SLA agreement). As result, it should develop a new model to estimate the SRT for different types of database access requests in cloud environment.

(iii) **Objective 3**: Development of a new solution for efficient query processing/optimization on large database systems available in the cloud environment. The new solution should be based on traditional and adaptive query processing techniques and its efficiency based on the SRT QoS parameter. It should integrate dynamic re-optimization techniques and the queries/subqueries are executed into several steps, where each step concurrently executes a dynamic execution strategy at query runtime. The dynamic execution costs of queries are based on the model proposed in Objective 2 and the proposed strategies will be deployed in the Amazon EC2 cloud infrastructure. In this thesis, the solution was evaluated on structured data, considering that some cloud computing platforms support SQL queries directly or indirectly. This makes the proposed solution relevant for these kind of problems.

1.4 Contributions

This thesis proposes a new solution to efficient query processing on large databases available in a cloud environment. It uses adaptive query processing based on heuristic rules and the cost of failing the SLA. Furthermore, it proposes a model for measuring the SRT estimated for different types of database access requests in this environment. The specific contributions of this thesis are:

(i) State of the art of traditional and adaptive queries processing techniques on databases systems in centralized, parallel and distributed environments. Moreover, the state of the art of techniques for query processing and optimization in the cloud;

(ii) A new model to estimate the SRT for different types of requests in cloud environment. It is very relevant when companies wish to migrate their applications, OLAP or not, to cloud service providers, with the goal to allocate computational resources on demand, to guarantee the quality of service in terms of service response time;

(iii) New algorithms and strategies based on traditional and adaptive query processing techniques (heuristic rules). Its efficiency is based on the model proposed to ensure the SRT QoS performance parameter;

(iv) Implementation of the SICLOPDB Framework based on the proposed strategies and evaluated on large scale with large volume of data, machines and queries in a real scenario of cloud computing.

1.5 List of publications from this work

The following papers were published during the development of this work:

(1) (Costa & Sousa, 2013) :: **[Annex A4] COSTA, C.M., SOUSA, A.L. Adaptive Query Processing in Cloud Database Systems. In 3rd International Conference on Cloud and Green Computing (CGC 2013), Karlsruhe, Germany, 2013.**

This paper shows the initial idea and main contribution of this work. Moreover, its architecture is presented. This short paper does not present any experiments, it presents only related works, architecture and contributions. It was published of work in progress Section.

(2) (Costa, Leite, & Sousa, 2015) :: **[Annex A5] COSTA, C.M., LEITE, C.R.M., SOUSA, A.L. Service Response Time Measurement Model of Service Level Agreements in Cloud Environment. In 5th International Symposium on Cloud and Service Computing (SC2 2015), Chengdu, China, 2015.**

In this paper, we propose a model to estimate the Recommended SRT for different types of requests on large databases available in the cloud environment. The proposed model is a non-intrusive solution and it the model was evaluated utilizing Amazon EC2 cloud infrastructure small instances type and the TPC-DS (Tpc BenchmarkTM Ds, 2012) like benchmark was used only for generating an OLAP database.

(3) (Costa, Leite, & Sousa, 2016) :: **[Annex A6] COSTA, C.M., LEITE, C.R.M., SOUSA, A.L. Efficient SQL Adaptive Query Processing in Cloud Databases Systems. In 2016 IEEE Conference on Evolving and Adaptive Intelligent Systems (IEEE EAIS 2016), Natal, Brazil, 2016**.

This paper presents a main contribution of this thesis. It presents the partitioning and monitoring strategies for adaptive processing of different types of queries (database access requests), a dynamic provisioning strategy and their algorithms. Moreover, it presents an implementation of the proposed solution and its architecture. Finally, it shows the experiments using Amazon EC2 cloud infrastructure small instances type and the TPC-DS like benchmark was used only for generating an OLAP database of structured data.

1.6 Thesis outline

To improve the understanding of the reader, we provide here a brief description of each chapter with its aim.

Chapter 2: State of the Art and Related Works: presents researches, concepts and technologies related to the object of study of this doctoral thesis. Firstly, we present an overview of Data Warehouse and OLAP (On-Line Analytical Processing) applications. Then, we discuss the SLA contract in cloud environment as well as specification of QoS parameters to this kind of SLA. After we discuss traditional and adaptive query processing of databases and query processing in the cloud. Finally, we present related works to query processing/optimization in cloud environments.

Chapter 3: Service Response Time Measurement Model of Service Level Agreements: In this chapter, we present a model for measuring a Service Response Time estimated for different request types on large databases available in a cloud environment. Firstly, we present the formal definition of a request used in this work. After, we present the SRT measurement model, its definition and tools. Finally, we discuss the experiments of the proposed model utilizing Amazon EC2 cloud infrastructure and the TPC-DS like benchmark and, finally their results.

Chapter 4: Efficient Adaptive Query Processing on Large Database Systems Available in the Cloud Environment: In this chapter, we present a new solution to efficient query processing on large databases available in a cloud environment and the SiclopDB Framework, which implements the proposed solution for this problem and its architecture. Firstly, we present the SLA violation cost and the total computational cost of a request used in this work. After, we discuss the SiclopDB framework architecture and its components. Then, we present a new partitioning and monitoring strategies for adaptive processing of different types of queries in the cloud, the dynamic provisioning strategy and finally their algorithms.

Chapter 5: Experimental Evaluation - Validation and Results: In this chapter we present the experiments of a case study using the strategies of query processing presented in Chapter 4. Therefore, firstly, we present the environment where the experiments were executed. Then, we present the methodology of the experiments. After, we show the requests used in the experiments. Finally, we present the results obtained as well as its analysis.

Chapter 6: Conclusions: In this chapter we conclude our work by describing the objectives achieved and we present some ideas that would be interesting for future research in this area.

Chapter 2 – State of art

2.1 Introduction

In this chapter, we present researches, concepts and technologies, which provides the support to this doctoral thesis. Hence, for better understanding, this chapter is organized as follows:

2.2 Data warehouse and OLAP: presents an overview of Data Warehouse and OLAP (On-Line Analytical Processing) applications.

2.**3 SLA in cloud computing**: discusses the SLA contract in cloud environment as well as specification of QoS parameters to this kind of SLA.

2.4 **Data processing**: discusses traditional and adaptive query processing of databases and query processing in the cloud.

2.5 **Related works**: presents related works to query processing/optimization in cloud environment.

2.6 **Conclusion**: presents the final considerations of this chapter.

2.2 Data warehouse and OLAP

A DW (Data Warehouse) is a computational system used to facilitate reporting and analysis of large volumes of data. Hence, a DW's main objective is to provide data to business analysts to support decision-making. In practice, as shown in Figure 2-1, a DW integrates multiple databases to provide a consolidated view (materialized view) of them, focusing on the business analysis goals.

Strategic information for decision-making using Data Warehouse have aroused great interest in organizations (Kimball & Ross, 2013), as this is a technology that when integrated with appropriate applications has many benefits, including: the speed of decision-making, optimized resource management and the discovery of new business opportunities.

As shown in Figure 2-1, for the construction and manipulation of a DW, applications for data extraction and processing are required. They are called ETL (Extract-Transform-Load) tools, and are used to build a materialized view from heterogeneous data sources. To access, manage and analyze data in a DW, OLAP (Online Analytical Processing) tools are used, and the results used to support decision-making.

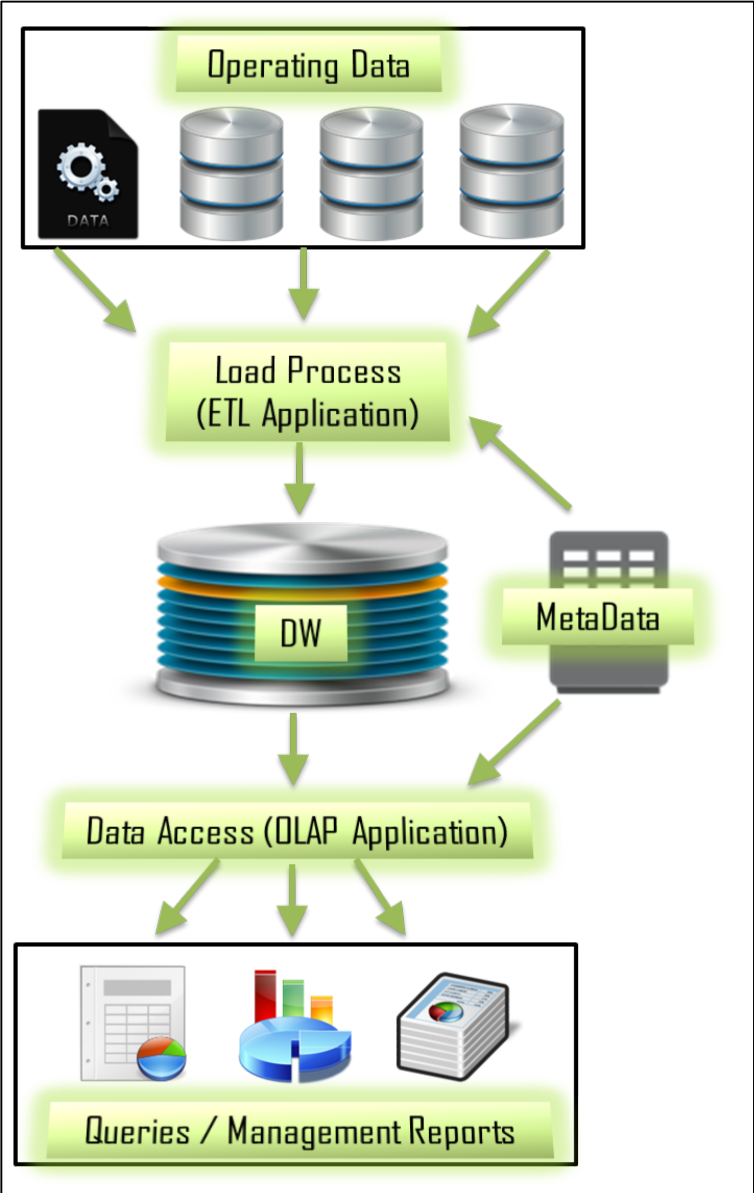


Figure 2-1. Data Warehouse Architecture.

Consider, for example, two relational schemes shown in Figure 2-2. These schemes simulate heterogeneous data sources, and they need to be designed and integrated in order to provide a materialized view. The schemes represent different sectors of a company, human resources (Figure 2-2 (a)) and sales sector (Figure 2-2 (b)).

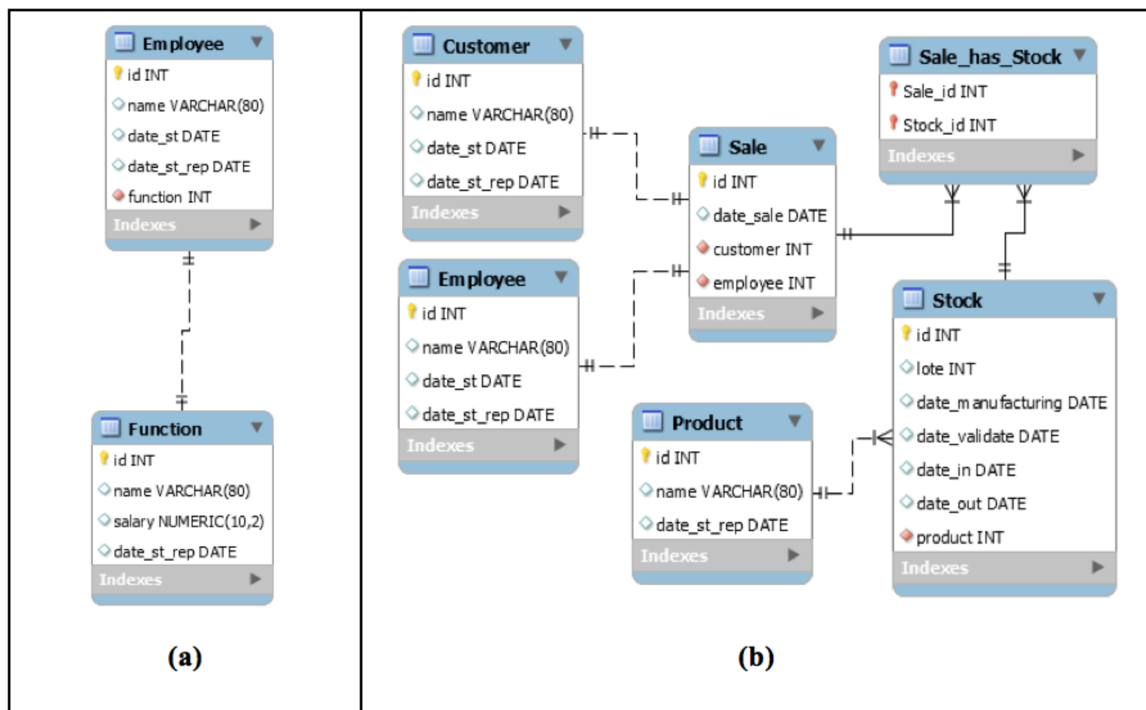


Figure 2-2. Database of a Company: (a) Relational Scheme of Human Resources Sector and (b) Relation Scheme of Sales Sector.

Using ETL tools, after the extraction process of data sources, the transformation process constructs the syntactic and semantic mappings between relational schemes, respecting the integrity constraints. Finally, in the loading process, a materialized view is generated in accordance with its mappings.

A possible materialized view is shown in Figure 2-3. It presents characteristics of both relational schemes of Figure 2-2. It is worth noting that the materialized view in DW must be created according to the goal to be achieved.

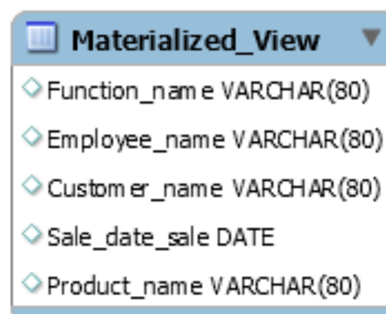


Figure 2-3. Example of Materialized View in Data Warehouse.

In a DW, the views are represented by multidimensional models, as will be seen in Section 2.2.1,

and the data is represented in data cubes, as will be seen in Section 2.2.2.

2.2.1 Multidimensional modeling of data warehouses

In DW, a scheme model widely used is the multidimensional model. It allows users to do operations on data simply. In this model, there is the relationship of facts and dimensions, in which facts are performance measures and dimensions are contexts of a fact.

In relational databases, a table, an attribute or a set of attributes of a table represents a dimension and a fact represents joins between two or more dimensions. For example, in Figure 2-3, the fact is the Materialized_View and its dimensions are Function_name, Employee_name, Customer_Name, Sale_date_sale and Product_name.

The star or snowflake models can represent multidimensional schemes. The most widely used model is the star model, in which a fact and its dimensions are shown explicitly. For example, Figure 2-4 shows the materialized view of Figure 2-3 in this model, there is a fact in the center of the star and its dimensions in tips. However, it is not necessary to have five dimensions in order to be called a star model.

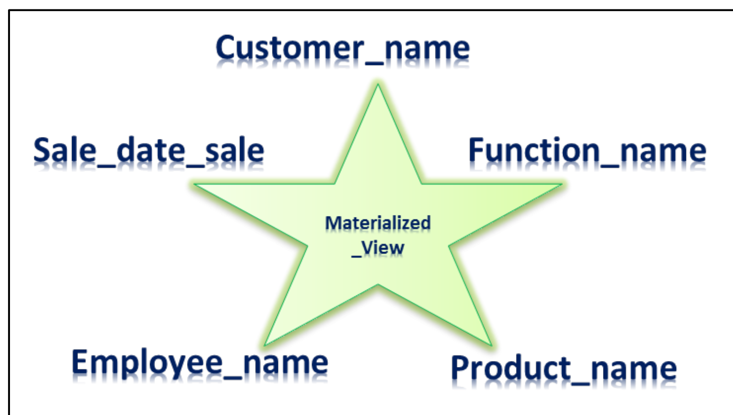


Figure 2-4. Star Model of Materialized_View Fact.

Considering the relational schemes in Figure 2-2 and the materialized view in Figure 2-3 a few facts can be represented, for example:

- A user wants to know in DW about functions and salaries of employees who best sold between 2013 and 2015. This query would be represented by the following model:

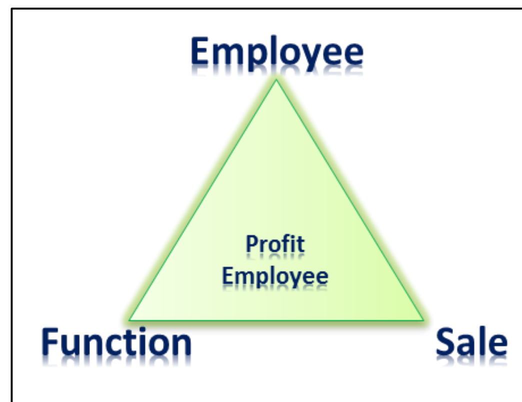


Figure 2-5. Fact: Profit Employee.

- A user wishes to know the top selling products between 2010 and 2015. Hence, the following model represents such query:

Figure 2-6. Fact: Top Selling Products.

2.2.2 OLAP applications

In a DW, data can be represented in different ways, but the most used is the data cubes, i.e. the data is in a cube in which each side represents a dimension. Currently, the cube modeling is the mostly used because there are many powerful tools using such approach. They are called OLAP applications

OLAP applications are used for analysis of DW's complex data. It allows that analysts, managers and executives have fast, consistent and interactive access to a wide variety of views of information (Kimball & Ross, 2013). Currently there is a great need to provide information at the right level of detail to support the decision-making activity. Thus, OLAP techniques provide this functionality (Elmasri & Navathe, 2010).

OLAP functionality is characterized by the dynamic multidimensional analysis of consolidated data. Thus, OLAP applications offer users several interfaces to make any operations (queries and manipulation) on the data in a DW. For example, operations such as drill-down and roll-up, which are the mostly used.

The drill-down operation consists of drilling a slice of the data cube, i.e. decomposing part of a cube to form a new cube, which therefore will be in greater level of detail. The roll-up operation consists of generating a data cube in a more generalized level, i.e. this operation is the opposite of the drill-down operation, creating a more general cube compared to the original one. These operations are exemplified in Figure 2-7, in which each data cube is a fact with three dimensions: Customer, Sale and Date_Sale.

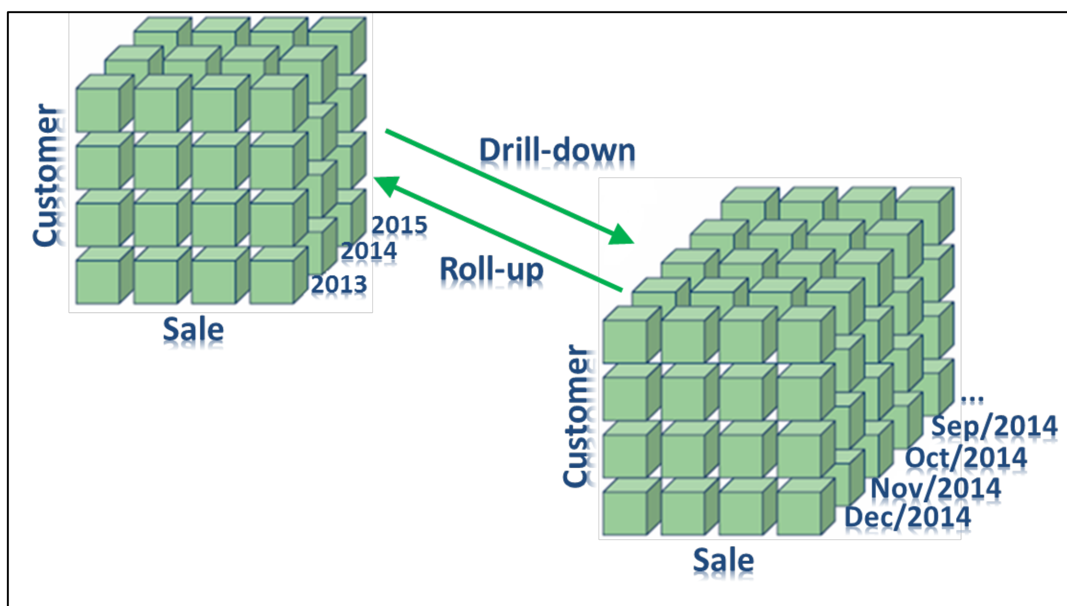


Figure 2-7. Example of Drill-down and Roll-up Operation using OLAP Applications.

In fact, OLAP applications can handle large amounts of complex data and the speed at which executives obtain information and make decisions determines the competitiveness of a company and its long-term success (Kimball & Ross, 2013). Therefore, considering that nowadays many companies are migrating their applications and data to the cloud, a great challenge to cloud computing providers is to ensure the quality of service, and performance for these types of applications deployed in this environment. This is the scenario of this work: OLAP services that manipulate large amounts of data in the cloud.

2.3 SLA in cloud computing

2.3.1 Definition and lifecycle of a SLA

The SLA (Service Level Agreement) is a formal service contract between a cloud service provider and its customers (Patel, Ranabahu, & Sheth, 2009; Wu & Buyya, 2010), usually a document that defines the levels of availability, functionality, performance, penalties and billing expected from the provider to its customers.

The Figure 2-8 presents the SLA lifecycle in three high level stages. The first stage is the **SLA Contract Definition**, which corresponds to the discovery of the service provider, model specification, negotiation and optimization of the SLA and as result, a SLA template is obtained. The second stage, called SLA Operation consists in the implementation, monitoring, evaluation, renegotiation and accounting services of SLA. Finally, the SLA Closing/Breaking, which involves the end/breach of contract between the parties. It is important to identify the causes to breach of contract: irrecoverable loss of data, provider's lack of performance, etc. In the following, we detail each stage of the SLA lifecycle.

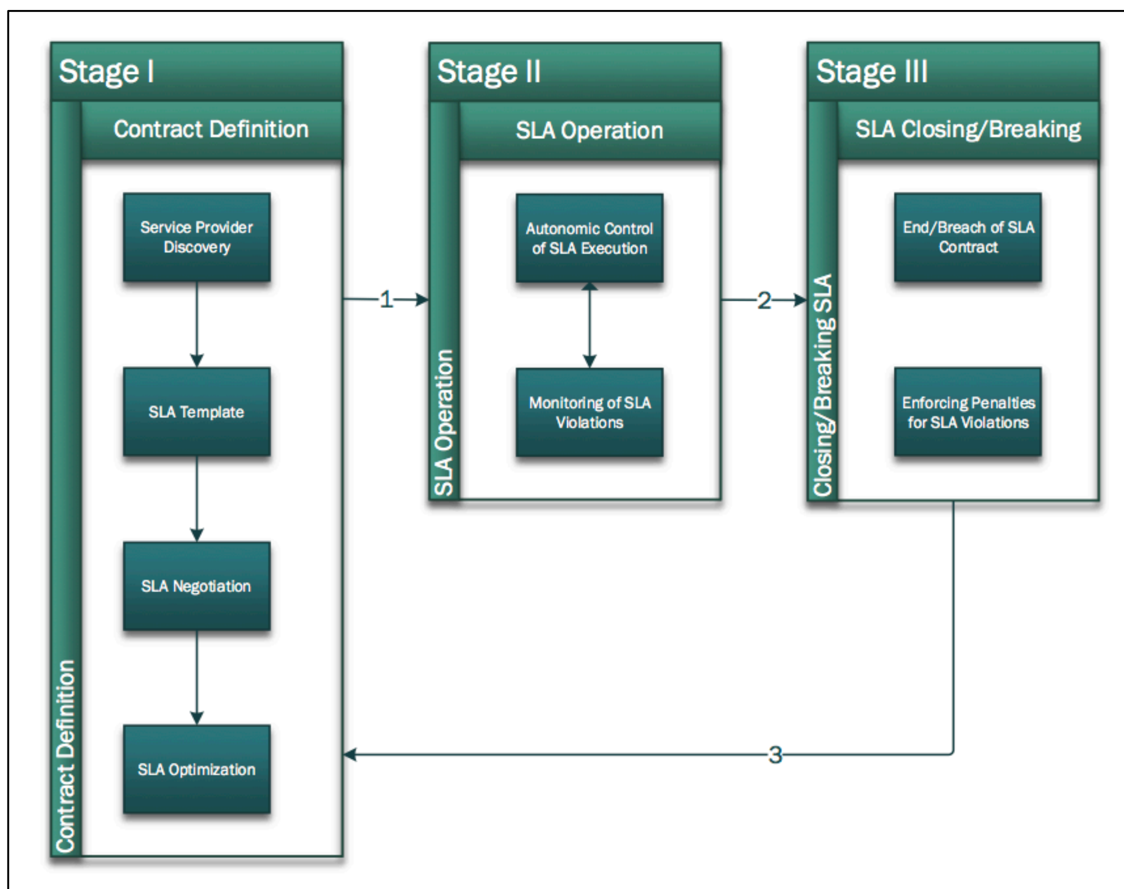


Figure 2-8. SLA Lifecycle.

STAGE I – SLA Contract Definition: Discovery of the service provider, model specification, negotiation and optimization of the SLA

In a cloud environment, it is important to locate resources that can efficiently satisfy the customer requirements/demands, because it is possible to find environments with different kinds of resources, standards, technologies and administrative policies. In addition, similar objects in different cloud environments may have different meanings.

In the process of discovering a service provider, the ideal for users is to have the largest possible amount of information of the cloud services environment. Primarily, this information must include the resources capacity, its availability and to know if they are accessible to a wide public. Thus, users can quickly find the services that best suit their objectives.

After choosing the provider, the terms of the contract between the parties is negotiated and defined from an existing SLA template. Among the terms we can highlight the following QoS parameters: (i) the provider ability to deliver the services, (ii) the desired performance of the provider from the workload of the user, (iii) the guarantees of availability and performance, (iv) the accounting parameters, (v) measurement/reporting mechanisms, and finally, (vi) the service costs and parameters of penalties in case of SLA violation. Furthermore, it is important that the parties have not ambiguous parameters, even if the parties use different protocols. Therefore languages like WSLA (Keller & Ludwig, 2003) and WS-Agreement (Andrieux et al., 2005) can be used to minimize this problem.

Finally, with the contract established between the parties, the responsibilities of each party should be detailed, as well as the consequences resulting from the breach of standards, software failures and other events that may influence the system behavior. This part of the process can consume a lot of time, and effort.

STAGE II – SLA Operation: execution and monitoring SLA violations

In this stage starts the execution and monitoring of provisioned resources to the costumer's requests. The real-time monitoring checks the execution of an instance of a service according to the settings of SLA, in order to detect whether the contract is being ensured or not.

The instance of a service is parameterized and compared to the SLA QoS parameters. When it is approaching or reaching a limit value, the environment must react to avoid a SLA violation. For example, (re)allocating or releasing resources to effectively optimize a task. Therefore, the provider

must consider how to optimize the use of resources and how to preserve the Quality of Service according to the priorities established on the SLA contract.

According to (Wu & Buyya, 2010), there are three types of infrastructures for SLA monitoring: (i) Trust module on the provider's side; (ii) Trust module on the customer side; and (iii) TTP (trusted third party), a trust module using third party. To avoid any suspect, the TTP monitoring is the preferred approach to manage this process.

When a breach of contract happens, the renegotiating of the SLA is a difficult task because no one wants to lose and therefore, tolerances should be part of the renegotiation. Moreover, flexibility in contract is also important because changes can be necessary to answer some external demand.

Other information such as global statistics are also relevant to check SLA and to account and establish the costs of the used resources. Thus resources usage should generate a list describing which services/resources were used, the measurement used and for how long, as well as relating the values agreed by the use of each of them in accordance to the definitions established by SLA (Wu & Buyya, 2010; Wu et al., 2011).

STAGE III – SLA Closing: closure or breach of contract and penalties for SLA violations

At this stage, the SLA and its settings are excluded from the service provider and the contract is finished. It is important to identify the causes that led the parties to the breach the contract, in case it has indeed been violated. Many penalty clauses of SLAs are linear and they do not present a good performance and best models can be extended to these clauses (Lee, Wang, Zomaya, & Zhou, 2010). Therefore, due to different types of violations, the penalty clauses need to be extensive.

The SLA cost parameters should provide information such as the price for the use of resources and instance of a service, the country's currency and the period for which the price is valid. The SLA model design should be flexible enough to allow different types of charges. In addition, the use of a service or resources above the agreed limit may cause additional costs to be charged.

2.3.2 QoS parameters of a SLA

Cloud computing has become an important paradigm for outsourcing IT resources. Currently there are many cloud providers offering different services with different prices, parameters and performance levels, even when those providers offer similar services. For example, Amazon EC2 offers IAAS (Infrastructure as a Service) services with the same computing power with different prices for different regions. In addition, several companies, including small and medium enterprises,

have started using the cloud infrastructure (Emeakaroha et al., 2012; Garg et al., 2013). Thereby, there is a wide range of different contracts with different SLA requirements. Thus, it becomes difficult for a customer to choose the most suitable provider to co-locate their applications.

A major challenge from the customer's point of view is to find the best cloud service, which can ensure/satisfy their QoS parameters agreed in contract. Therefore, it is important to consider which is the cloud service best suited for a particular customer profile. Afterwords, the customers need to have a way to identify and measure key performance criteria that are important to their applications. For example, financial organizations usually require security and privacy QoS requirements, but the availability QoS requirement, although important, is not a priority of these organizations (Chi et al., 2011).

Therefore, how to select a feasible service to meet the demands of different users has become a popular research area. In order to improve the customer's satisfaction, many studies (Alrifai & Risse, 2009; Canfora, di Penta, Esposito, & Villani, 2005; Liang, Zou, Guo, Yang, & Lin, 2013; Siegel & Perdue, 2012; Zeng et al., 2004; Zeng, Benatallah, Dumas, Kalagnanam, & Sheng, 2003; Zheng, Ma, Lyu, & King, 2009; Zheng, Zhang, & Lyu, 2010) focused on the QoS optimization.

The CSMIC (Cloud Service Measurement Index Consortium) consortium is widely used and aims to define the QoS parameters to be used by most cloud provider and to provide a methodology for calculating a relative index to compare the services of different cloud providers.

The CSMIC started in 2010 by the members of CA Technologies, a software company headquartered in New York, and by researchers at Carnegie Mellon University, located in Pennsylvania in the United States. Currently, many others members are part of this consortium, such as: Accenture, a global company for consulting, technology services and outsourcing serving customers in more than 120 countries; Cask LLC, a telecommunications company located in San Diego, California; DSCI (Data Security Council) of India, an organization of technological innovations on the protection and technological development of security and data privacy; IAOP (International Association of Outsourcing Professionals), a global organization of standards and defense of outsourcing in the business world; Mycroft, an innovative IT company located in England; TM Forum, a global nonprofit company for service providers and their suppliers in telecommunications and entertainment industries; TPI, a consulting company in outsourcing in the United States; researchers at the Public University of London; researchers at Stony Brook University, located in New York; and finally, researchers at the University of Melbourne in Australia.

The major product of this consortium is the SMI (Service Measurement Index) (Garg, Versteeg, & Buyya, 2011; Garg et al., 2013; Siegel & Perdue, 2012), a framework that aims to measure the services commonly offered in cloud environments. Specifically, the SMI consists of a set of KPIs (Key Performance Indicators), providing a global view of QoS parameters and their metrics used by cloud service providers. With SMI, the customers can make a better selection of a cloud service provider (Emeakaroha et al., 2012; Garg et al., 2013; Siegel & Perdue, 2012; Vaulx, Simmon, & Bohn, 2015). A KPI is a key QoS parameter, in which has one or more QoS sub-parameters. The following are the main QoS parameters defined in the SMI:

(i) **Accountability**: this group includes QoS parameters that define a relationship of trust between customer and service provider. It is a fact that no organization would like to install their applications and store their critical data in a cloud environment, in which there is no good ethics and/or responsibility, especially when it comes to data safety and reliability. Among the sub-parameters considered important to measure ethics and responsibilities of a cloud services provider, we can highlight auditability, compliance, data ownership, ethicality and sustainability.

(ii) **Agility**: this group includes QoS parameters in order to measure the evolutionary flexibility of the provider capacity, identifying how fast new capabilities can be integrated into the IT according to business needs. This QoS parameter is quite interesting for organizations because the expansion and faster change of IT resources represent fewer costs for organizations. Parameters considered as agility of cloud services are elasticity, portability, adaptability and flexibility.

(iii) **Cost:** one of the first questions arising in organizations before migrating data to a cloud environment is whether it is profitable or not. Cost is clearly one of the main QoS parameters for IT and the business, and sometimes it takes many hours or weeks of discussion to reach an agreement. However, in the SMI, the cost is the simplest quantifier and has the following sub-parameters: acquisition & transition cost, on-going cost and profit or cost sharing.

(iv) **Performance**: this group includes QoS parameters for the performance of cloud services. There are different solutions offered by cloud providers in face of the need for different IT organizations. Among the sub-parameters that measure the performance of cloud services, we can highlight accuracy, interoperability, service response time, throughput and efficiency. This set is one of the most important

group of SLA QoS parameters because it is the main aim for most customers using cloud computing. Therefore, for the cloud services to ensure customer expectations in terms of performance, it is necessary to understand how these sub-parameters are measured.

(v) **Assurance**: this group includes QoS parameters that measure the probability of a cloud service to perform as expected or agreed in the SLA agreement. It is essential for every organization to expand its business and provide better services to their customers. Therefore, reliability, resiliency and service stability are important factors when contracting a cloud service.

(vi) **Security and Privacy**: data security and privacy are important to the majority of organizations. Data hosting under the responsibility of another organization is always a critical issue and requires strict security policies of cloud providers. For example, financial organizations require high level of security of their data. They require their data to be safe and private from any tampering or unauthorized access. Finally, this set includes sub-parameters such as confidentiality, privacy, integrity and availability of data.

(vii) **Usability**: Usability represents one of the main quality parameters. It represents the ease of benefiting from the service and from the information it provides (Corradini, Polzonetti, Re, & Tesei, 2008). The easier it is, more organizations will migrate its applications to the cloud. Usability can depend on multiple factors such as accessibility, installability, learnability and operability.

As shown in Figure 2-9, Currently, SMI has over 50 parameters and sub-parameters, each one can be measured and evaluated by a customer for an appropriate choice of a cloud service provider. Thus, the SMI provides a global view of QoS parameters needed for a cloud service provider. In addition, it assists customers in understanding and measuring the parameters that will be used in stage of template specification, negotiation and optimization of the SLA as shown in Section 2.3.1. One can still note that SMI indirectly helps controlling the monitoring of SLA violations, because a service provider when properly selected increases the probability that SLA requirements are guaranteed.

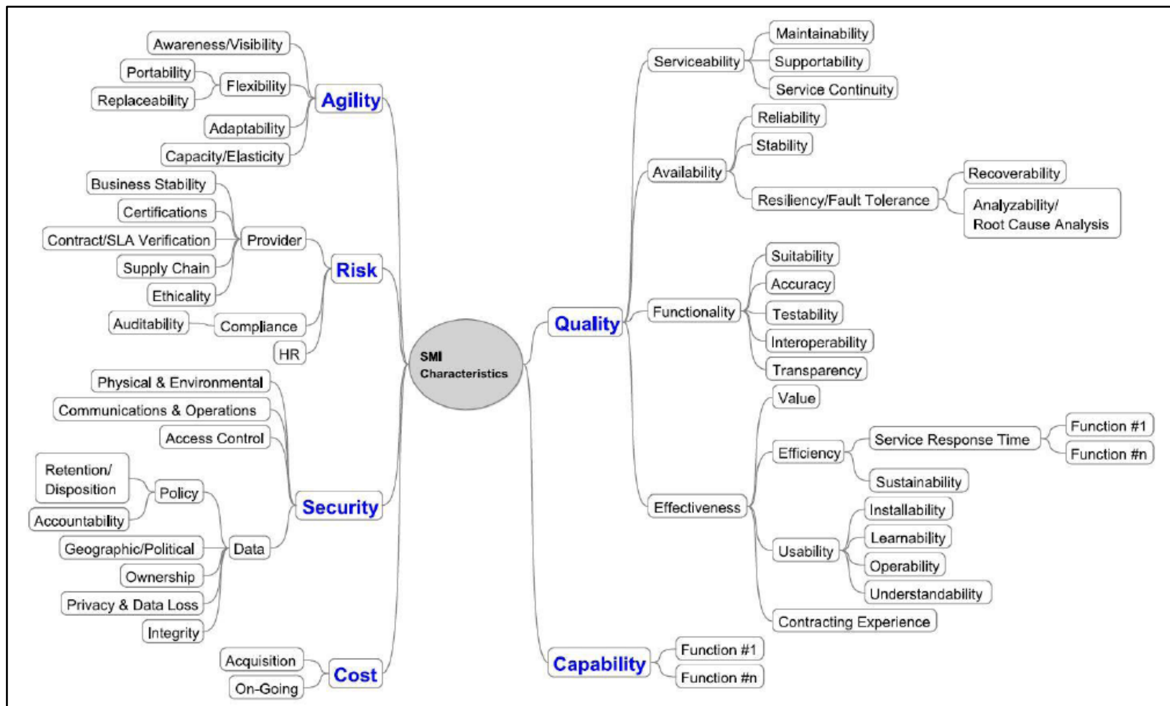


Figure 2-9. Parameters and Sub-parameters defined in the SMI (Siegel & Perdue, 2012).

2.3.3 QoS performance parameters of a SLA

The QoS performance parameters are among the most important clauses in a SLA, because it is of great interest to customer to know clearly their expectations when executing a request in a service of a cloud provider (Emeakaroha et al., 2012). To measure them, the SMI defined four sub-parameters: service response time, accuracy, throughput, efficiency, elasticity and scalability. In the following, we present these parameters in detail.

Service Response Time: the execution efficiency of a service can be measured in terms of response time; i.e. how fast the service is ready for use. For example, if a user indirectly requests a virtual machine on a cloud provider, then the service response time is the time given for the provider to begin serving the request. In this example, it includes the virtual machine provision, start, IP address allocation and application(s) start.

According to the SMI, the service response time depends on various factors such as average response time, maximum response time assured by the service provider and the percentage in which the response time exceeds the maximum time promised by the provider:

- The average response time of a service (T_ms) is given by:

$$T_{ms} = \sum_{i} T_{rsi}/n \quad \textbf{(2.1)}$$

in which T_{rsi} is the time between the moment that a user i requests a service and the service is ready to process the i request. n number of times a service was requested. Therefore, T_ms the sum of the service response times divided by the number of times the service was requested.

- The Maximum Response Time (T_{max}) corresponds to the maximum promised time a cloud service is ready to execute a request.
- The Violations of Maximum Response Time of a Service (T_{max}) is given by the quotient between the number of times that the response time was higher than the maximum promised response time and the number of requests, expressed in percentage.

$$VT_{max} = (n'/n)\times 100 \quad \textbf{(2.2)}$$

in which n' is the number of service requests that the cloud service provider was not able to ensure the contract and n is the total number of service requests.

Accuracy: the performance accuracy of a service is measured by the degree of closeness of requirements met when compared to the expected requirements. For computing resources, such as virtual machines, the accuracy can be equated by the number of times the service provider breached the SLA contract. Hence, if f_i is the number of times the cloud service provider does not met the requirements for a user i, and n the number of users who accessed the service, then the accuracy rate is defined as:

$$\sum_{i} \frac{f_i}{n} \quad \textbf{(2.3)}$$

Throughput and efficiency: Throughput and efficiency are important measures to evaluate the performance of services in cloud providers. The throughput corresponds to the number of activities performed by the cloud service per time unit. The throughput depends on several factors that can affect the performance of a task. For example, consider a user application that has n tasks, which are subjected to run on m machines of a service provider. Let Let $T_e(n, m)$ be the execution time of n tasks in m machines. Let T_0 be the overhead time due to factors such as delays in the startup

of infrastructure and delays in communication between tasks. Thus, the total throughput of a cloud service is given by:

$$\boldsymbol{\alpha} = \frac{\boldsymbol{n}}{\boldsymbol{T_e(n,m) + T_0}} \qquad \textbf{(2.4)}$$

The efficiency of a cloud application indicates the effective use of leased resources. Therefore, the larger the efficiency value, the lower the overhead. Thus, system efficiency is given by:

$$\frac{\boldsymbol{T_e(n,m)}}{\boldsymbol{T_e(n,m) + T_0}} \qquad \textbf{(2.5)}$$

Besides the presented QoS parameters, other parameters related indirectly to the performance of cloud services are elasticity, scalability and availability. **Availability** corresponds to the percentage of time a customer can access the service. Let T_d be the time the service was available and T_{dn} the time the service was not available; the availability is given by:

$$\frac{\boldsymbol{T_d}}{\boldsymbol{T_d + T_{dn}}} \boldsymbol{\times 100} \qquad \textbf{(2.6)}$$

Elasticity is defined in terms of how much a cloud service can be scaled, even during a service overload. Elasticity is defined by two parameters: the average time needed to expand or contract the capacity of the service, and the maximum capacity of the service. The capacity is the maximum number of compute units that can be provided at peak times.

Scalability is determined by the capacity of a system to handle a large number of requests from simultaneous applications. The ability to scale resources is an essential part of the elasticity provided by cloud computing. However, this measure is more applied to the performance perspective of user applications.

There are two types of scalability: horizontal, which means the increase of cloud resources of the same type. For example, the booting of more virtual machines of the same type during overloads. The vertical scalability is defined as the ability to increase the capacity of a cloud service, such as a virtual machine by adding physical memory resources, CPU speed and/or network bandwidth. The horizontal scalability is given by the elasticity and the vertical scalability can be calculated according to the maximum increase in the available resources of a cloud service.

2.4 Data processing

The QoS performance parameters presented can be used to resolve the new challenges of data management in cloud environment. Mainly, the challenges related to query optimization ensuring the response time. In the literature, there are several works related to query processing and optimization in traditional DBMSs (Deshpande, Ives, & Raman, 2007; Gounaris, Paton, Fernandes, & Sakellariou, 2002; Zhao, Hu, & Meng, 2010). These works provide the basis for the current requirements, such as, data management in the cloud. Thus, this section presents the state of the art in query processing in databases.

2.4.1 Query processing in database

DBMSs (Database Management Systems) implement various techniques to execute efficiently a query in their database(s). These techniques are based on the data model managed by the DBMS. In this section, we will address the relational DBMS optimization techniques, which are the basis for most other models.

In relational DBMS, a SQL (Structured Query Language) query first goes through a lexical analyzer and, then, a syntactic analyzer for correctness and query validation. Then, the validated query is rewritten in a tree data structure, called the query tree. In the literature, many authors use graphs to represent the query tree. Then the DBMS optimizer chooses an effective strategy, also known as efficient execution plan, to execute the query tree. A query plan (or query execution plan) is an ordered set of steps used to access data in a SQL relational database management system.

The main goal of the optimizer is to find an appropriate query plan, among others, to process a query that gives the lowest response time to the user. Finally, the optimized execution plan is executed and the result of the query is returned to the user.

Therefore, the query optimizer is an indispensable component in a relational DBMS engine. To improve the performance of a query, traditional optimizers use two techniques, not necessarily in the following order:

(i) **Optimizing the query plan based on heuristic rules**, which modify the internal representation of the query tree. The heuristics rules use equivalence expressions to transform an initial query tree in a final optimized query tree. An example of a classic heuristic rule is to apply SELECT and PROJECT operations before applying JOIN or any other binary query operation. DBMSs can achieve a good optimization

with a set of algorithms that use several heuristics rules in order to reach maximum query performance (Deshpande et al., 2007; Elmasri & Navathe, 2010; Gounaris et al., 2002).

(ii) **Optimizing the query plan based on costs**, since the publication of the System-R paper (Selinger et al., 1979), cost-based optimizers have been widely adopted. Usually the costs are quite complex to calculate, because they depend on estimates, properties of execution plans and specific cost formulas for each query plan operator (Bruno, Chaudhuri, & Ramamurthy, 2009). According to (Elmasri & Navathe, 2010), the cost of running a query includes the following components: (i) I/O Cost to access the hard disk: search operations, reading and writing of data blocks on hard disk; (ii) Storage Cost: temporary files generated during query execution; (iii) Computing Cost: processing of query operations in main memory and CPU, such as read-write on records and/or buffers; (iv) Memory Usage Cost: related to the number of necessary memory buffers during query execution; (v) Communication Cost: related to the cost of transport of the query and its results from a database site to the site or terminal where the query originated. The calculation of this cost is quite important because it is most expensive cost in distributed database systems (Abadi, 2010; Elmasri & Navathe, 2010).

The cost-based optimization presents some inferences. For example, for large databases it is more important to minimize the I/O cost of access to the hard drive. In small and parallel databases, it is interesting to reduce the computational cost. In distributed databases, it is interesting to lower the communication cost. In native XML databases, in addition to observing the computational cost, it is interesting to follow some guidelines, such as avoiding standardization, employing unique element names, pre-calculating values and transforming data with their queries.

For decades, different techniques were developed, such as, Query Hiting and Semantic Query Optimization (Bruno et al., 2009; Elmasri & Navathe, 2010). The Query Hiting technique, quite common in current databases, instructs the optimizer to restrict its search space to a certain subset of query plans (for example, imposing a choice of plans that use a particular type index, or determining the order and/or join method) (Florescu & Kossmann, 2009). The semantic optimization techniques use restrictions of the database scheme, as CHECK, TRIGGER and STORE PROCEDURE to improve query performance. Consider, for example, that in a STUDENT table, its

NAME column has a NOT NULL restriction. Therefore, the following query, SELECT * FROM STUDENT WHERE NAME IS NULL, would be executed quickly because the semantic optimizer would notice before executing the query, via restriction, that there is not any student with null names.

We observe a complex universe of rules, algorithms, formulas and guidelines that compose the traditional query optimization, whose main objective is to improve the performance of queries to the database (faster response time for users). However, in the context of cloud databases, as shown in Section 2.4.3, it is not a priority to improve query performance, but to achieve service quality, i.e. a suitable performance according to the SLA contract. Therefore, we understand that this universe must be readapted for cloud computing technologies.

2.4.2 Adaptive query processing

In DBMSs, the optimizer improves the performance of compile-time queries. Thus, optimization is just one-step before the effective execution of the query. However, in parallel and distributed environments in which statistical information about the availability of databases can be minimal and the availability or loading of physical and virtual resources are subject to change, query optimization can have a poor perform, especially when queries move and/or return large amounts of data, since it is not possible to have precise cost estimate and a good selectivity, for the environment is highly unpredictable and volatile (Deshpande et al., 2007; Gounaris et al., 2002).

In this context, a solution to produce a good query execution plan is to use adaptive query processing techniques, which interacts with environmental changes modifying the query execution plan at runtime. The adaptive processing aims to improve query performance by modifying its query plan in accordance with environmental changes (infrastructure, workload, etc.) at runtime of the query (Deshpande et al., 2007).

Two important tasks in adaptive query processing is to modify and build, when necessary, new operators in query plan at runtime. The modification of the query plan may occur in the physical and/or logical query plans. Changes in logical query plan consist in modify the SELECT, PROJECT and ORDER operations, and the query plan format. For example, a change in the execution site of the SELECT operator may occur if during query execution, a data replica of the remote site becomes available in the query site itself. Changes in the physical query plan consist of modifying operations indexes and joins algorithms. For example, a hash join can be replaced by an index join, if an index attribute of the junction becomes available during the query execution.

In general, the adaptive query processing techniques emphasize the following problem areas (Deshpande et al., 2007; Gounaris et al., 2002):

(i) **Fluctuations in the main memory**: correspond to techniques that try to adapt the shortage of memory and memory availability in excess. In this case, the query execution plans may be forced to release/acquire some or all of the resources they have during query execution;

(ii) **Users preferences**: cases in which techniques are built for users who are interested in quickly obtaining partial results of a query

(iii) **Data input rates**: correspond to techniques that adapt to data input rates because the quality of a query execution plan depends greatly on the estimation accuracy of input parameter values (Yin, Hameurlain, & Morvan, 2015). They are generally applied in parallel and distributed systems, in which the response times of the remote data sources are quite unpredictable;

(iv) **Current statistics**: correspond to techniques to acquire statistical information at runtime of the query, ensuring that the information is valid for the current conditions and consequently adapting best query execution plan. Therefore, the optimizer may be recalled repeatedly;

(v) **Performance fluctuations**: problem that often occurs in parallel systems. In this case, the techniques are adapted to the site performance fluctuations trying to find data replicas in sites with lower load for processing the query;

(vi) **Any change in the environment**: combines the previous techniques. Some are widespread and can adapt to various types of environmental changes, that is, computer resources, availability of processor and memory, data characteristics, operator costs, selectivity and data input rates.

Therefore, the adaptive query processing is mainly useful in highly dynamic, unpredictable and volatile environments, especially if databases are integrated in a cluster (Foster & Kesselman, 2003), being this the common infrastructure of computing clouds (Zhang & Ardagna, 2004). Adaptive query processing has the ability to dynamically and automatically allocate or release resources (elasticity of resources) during the query runtime and hence, it has a good performance in query response. However, this technique needs to be readapted to the cloud environment, since, along with traditional optimizers; it does not ensure all requirement for query processing in cloud.

2.4.3 Query processing in cloud

The infrastructure of a computing cloud consists of a cluster with hundreds or thousands of computers, which are used for storage and data management. In the cluster, computers are networked and their communication takes place through the system as if they were a single large machine. These Computers are called master or slave nodes. Master nodes are responsible for the metadata management of the entire cluster, scheduling the execution of tasks on slave nodes. Slave nodes are responsible for storing data.

Figure 2-10 shows the query processing in a cloud. First, the query is scheduled, and then partitioned into sub-queries, which go to the slave nodes that store the relevant data to process them. Then each sub-query is performed on the slave nodes and one single result is presented to the user. According to (Zhao et al., 2010), the query on the cloud platform is different from central or parallel database. In the cloud platform, client query is often presented against the master nodes. After that, the master nodes decide which slave nodes are relevant to the query and then the query is passed to the slave nodes to do the query processing directly.

Hence, there are two important differences between query processing in the cloud and query processing in traditional, parallel and distributed environment (Kllapi et al., 2011; Padhy, Patra, & Satapathy, 2012):

(i) **Interest in Data Environment**: the processing and technology in parallel/distributed environment employ system-level measures, such as, database throughput rate, average length of query response, and so on. In the cloud environment, the interest is linked to profit optimization in business level as in SLA contracts.

(ii) **Scalability and Workload**: The great scalability and dynamic workload required in cluster makes query processing in cloud environment a different problem when compared to parallel/distributed processing. In the cloud, these problems must be solved by “contractual elasticity” i.e. providing use of resources to avoid SLA contract violations.

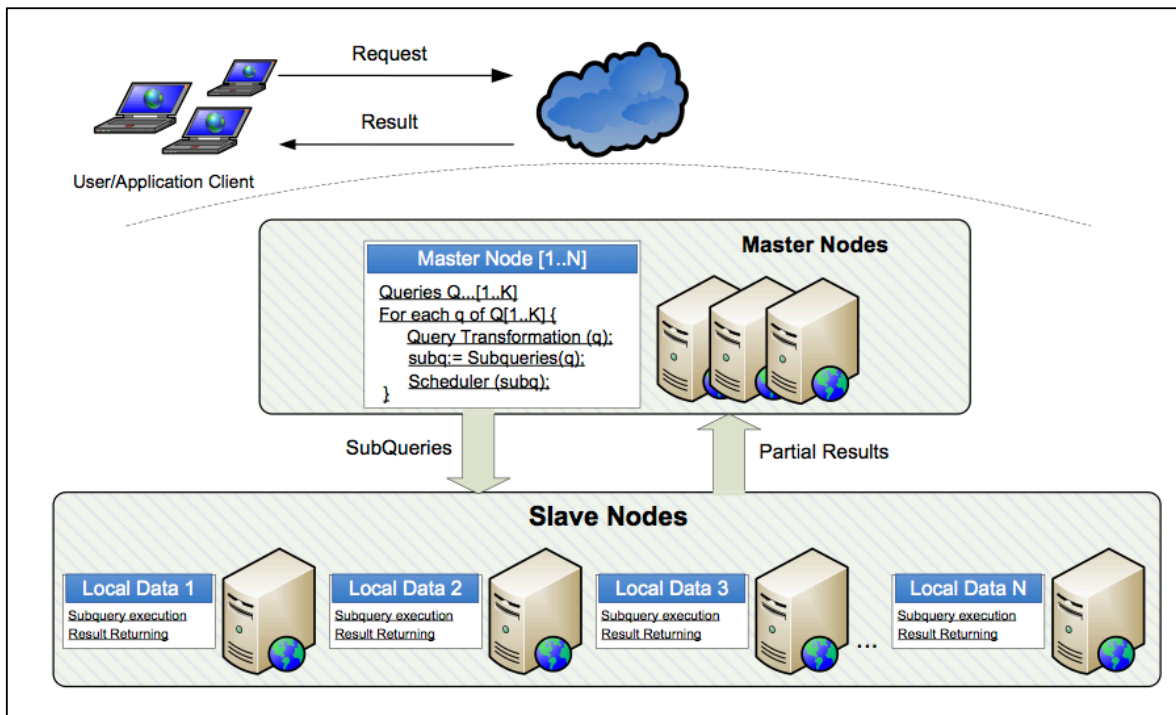


Figure 2-10. Query Processing in the Cloud.

2.5 Related works

Currently, several studies have been focused in search of techniques for efficient query processing in the cloud (Alves et al., 2011; Amazon Web Services, 2015; Cervino et al., 2012; Chi et al., 2011; Coelho da Silva et al., 2012, 2013; Curino et al., 2011; Dean & Ghemawat, 2008a, 2008b; Guitart et al., 2008; Kllapi et al., 2011; Mian et al., 2013; Naskos et al., 2015; Rogers et al., 2010; Sharma et al., 2010, 2011; Vigfusson et al., 2009). Among these, we can cite the ESQP (Efficient SQL Query Processing) (Kllapi et al., 2011; Zhao et al., 2010), which is a SQL query processing framework that uses replicas stored in the cloud. It aims to minimize the time of query execution, exploring replicated data. It adopts the MapReduce framework strategy (Dean & Ghemawat, 2008a, 2008b) to decompose an SQL query into several subqueries in accordance with the corresponding data replicas. The ESQP employs techniques including index and pipeline, to improve the processing efficiency of the subquery. However, the ESQP does not perform adaptive query processing, acting proactively, which may not be suitable in highly unpredictable environments on the availability of resources.

Another important work is the development of the SLA-Tree framework (Chi et al., 2011), which provides a new data structure to efficiently ensure the SLA agreement. SLA-Tree uses the response time of queries according to the SLA contract. The response time in this work is the difference

between the time that the query is submitted to the system and the moment the query execution is concluded in the provider. SLA-Tree considers that there is only one buffer for each cluster node and, as in (Zhao et al., 2010) the workload is known before query execution. That way, the query is not monitored during its execution.

(Vigfusson et al., 2009) present an adaptive algorithm to optimize the response time of queries in distributed databases. The algorithm partitions and adaptively identifies the best level of parallelism for each query. The authors propose an adaptive provisioning algorithm for only select-range queries and consider variations in performance of VMs (Virtual Machines). On the other hand, this work does not observe an SLA and does not specify the frequency of the monitoring algorithm during queries execution.

(Iqbal et al., 2009) present an SLA-oriented resource manager focused on cloud computing and based on open source technology. It provides adaptive resource allocation and dynamic load balancing for Web applications in order to ensure a SLA. One of the limitations of this work is that it uses resource increase not providing the mechanisms for resource shrink. In addition, the work does not check DBMS variables for database access requests, addressing only the level of the application server layer. Thus, monitoring is given using only system variables.

(Rogers et al., 2010) present a framework for the provisioning of resources that identifies a set of minimum cost of resources (i.e. a set of potentially heterogeneous virtual machines) that can collectively satisfy a variable workload on time within the quality expectations of the service. The authors describe two solutions for the resource-provisioning problem. The black box provisioning, which considers profiles of performance and cost of different types of VMs under the variation rates of queries that are given previously (it uses execution samples). The goal is to capture the input rate that each VM supports without violating the service quality associated with the queries executed by the system. The white box provisioning, which estimates how much computational resources are required to execute the workload using the database optimizer statistics to predict the consumption of physical resources (I/O, memory, CPU) for each query. Despite the fact, the work presents a solution that involves minimizing costs in the use of resources to customer requests and worries about latency of these consultations of a given SLA, it does not use a monitoring strategy during requests execution, which may not be suitable in highly unpredictable environments on the availability of resources.

(Alves et al., 2011) propose the FloodDQ system, a MapReduce system that uses deadlines for running queries without discarding data or reducing results accuracy. The FloodDQ uses adaptive processing providing the increase or decrease of resources during query execution. If during query execution it is determined that it passed a margin of query execution safety (progress monitoring estimate and computing resources), the system requests more computational nodes to execute the query. The safety margin is calculated by using the estimated query execution progress using algorithms (Morton, Balazinska, & Grossman, 2010; Morton, Friesen, Balazinska, & Grossman, 2010). This work restricts its scope to pipeline single queries (queries without joins). The calculation of the number of nodes to be added or removed is based on the data processing rate and the work assumes that all nodes have the same data processing capability. Moreover, this work uses the strategy of regular intervals monitoring, which requires that VMs have the same performance.

(Sharma et al., 2010, 2011) propose the Kingfisher, a provisioning framework based on applications cost in a cloud environment. It aims at minimizing the customer cost. That is, the provisioning is based on the best use of resources while minimizing the cost of the customer according to resources usage. This work uses integer linear programming to calculate the costs and decision-making in elasticity. The variables used for the calculations involve performance characteristics of different types of servers and their costs for core, provisioning mechanisms supported in the cloud and a model to estimate the cost/overhead of each mechanism. Kingfisher uses a proactive method to know when provisioning and assumes an ideal workload predictor that uses recovered statistics of a monitoring system. This predictor is able to obtain estimates of future workloads. Finally, the Kingfisher does not perform adaptive processing queries.

(Curino et al., 2011) propose the Kairos system, which uses nonlinear programming in order to minimize the number of servers and make load balancing for running queries. The Kairos uses techniques to measure the hardware requirements for workload on the database, thereby achieving to predict resources that will be used for query. This work also does not use adaptive query processing.

(Cervino et al., 2012) propose an adaptive algorithm of VM provisioning. It uses a stream processing system distributed in the cloud. The provisioning of VMs to be allocated for a given query is based on streams rate of the current workload. The methodology consists of periodically calling the algorithm and calculating the number of VMs that are required to process the demand. It scales the

number of VMs based only on input stream rate. In addition, there is the dynamic resources provisioning during query processing.

(Coelho da Silva et al., 2012, 2013) present a non-intrusive framework for adaptive query processing in databases implanted in a cloud environment. This work observes query response time of the SLA contract; makes adaptive monitoring considering the heterogeneous environment, and therefore, it considers that the VMs may have different performances. One limitation of this work is that it limits the scope to only select-range queries.

(Mian et al., 2013) propose a resource-provisioning framework in a public cloud to execute requests in large amounts of data. This work proposes an SLA cost model and presents a provisioning method based on SLA time, predicting the best value to execute requests at any given time. For validation, the framework was evaluated using Amazon EC2. This work does not use monitoring strategy during requests execution, which might not be suitable in highly unpredictable environments on the availability of resources.

(Naskos et al., 2015) propose a probabilistic model checking-based approach to resizing an application cluster of VMs so that elasticity decisions are amenable to quantitative analysis. Experiments using real datasets were conducted, and the results shown a significantly decrease on the frequency of user-defined threshold violations. However, this work does not perform adaptive query processing and does not use monitoring during requests execution. There is only the monitoring of the incoming workload and the current system state.

Another important work is the Amazon Auto Scaling (Amazon Web Services, 2015), which allows scaling requests following criteria, for example, the average CPU utilization. The automatic scheduler is based on analysis of requests traffic in execution and this solution works with Axis2 Web services running on Amazon EC2. Finally, (Goiri, Julià, Fitó, Macías, & Guitart, 2012; Guitart et al., 2008) uses admission control and dynamic resource provisioning. This work is responsible for allocating resources and tries to ensure the desired QoS during system overload. The server machines of their systems are able to adapt automatically to changes in workload. However, these works do not use monitoring during requests execution.

Finally, Table 2-1 summarizes the related works. As shown in the table(Amazon Web Services, 2015; Cervino et al., 2012; Chi et al., 2011; Curino et al., 2011; Guitart et al., 2008; Kllapi et al., 2011; Mian et al., 2013; Naskos et al., 2015; Rogers et al., 2010; Sharma et al., 2011; Zhao et al., 2010) do not use the strategy of monitoring during requests execution. In (Vigfusson et al., 2009)

the algorithm is adaptive optimizing the response time of queries. However, it does not observe the SLA agreement and does not specify the frequency of the monitoring algorithm during query execution. (Iqbal et al., 2009) presents an adaptive SLA-oriented resource manager. However, it only predicts the provisioning of resources and does not check DBMS variables for database access requests, addressing only the level of the application server layer. (Alves et al., 2011) uses the strategy of regular monitoring intervals during requests execution and therefore does not consider that VMs may have different performance. In addition, it limits its scope to single pipeline queries (queries without joints). (Coelho da Silva et al., 2012, 2013) consider that VMs may have different performances and there is the adaptive monitoring query execution. However, the scope is limited only to select-range queries.

Table 2-1. Characteristics of related work.

Researches	Adaptive Query Processing	Based on SRT on the SLA contract	Type of Environment it is Applied	Query Restriction	Scaling: provisioning or release of resources
(Goiri et al., 2012; Guitart et al., 2008)	No	Yes	Heterogeneous	Not restricted	Provisioning of Resources
(Vigfusson et al., 2009)	Yes	No	Heterogeneous	Select-range	Provisioning of Resources
(Iqbal et al., 2009)	No	Yes	Heterogeneous	Not restricted	Provisioning of Resources
(Rogers et al., 2010)	No	Yes	Heterogeneous	Not restricted	Provisioning of Resources
(Alves et al., 2011)	Yes	Yes	Homogeneous	Select-range	Provisioning and Release of Resources
(Curino et al., 2011)	No	Yes	Heterogeneous	Not restricted	Not applied
(Kllapi et al., 2011; Zhao et al., 2010)	No	Yes	Heterogeneous	Not restricted	Provisioning of Resources
(Sharma et al., 2010, 2011)	No	No	Heterogeneous	Not restricted	Provisioning of Resources
(Chi et al., 2011)	No	Yes	Heterogeneous	Not restricted	Provisioning of Resources
(Cervino et al., 2012)	No	No	Heterogeneous	Not restricted	Provisioning of Resources
(Coelho da Silva et al., 2012, 2013)	Yes	Yes	Heterogeneous	Select-range	Provisioning and Release of Resources
(Mian et al., 2013)	No	Yes	Heterogeneous	Not restricted	Provisioning of Resources

Researches	Adaptive Query Processing	Based on SRT on the SLA contract	Type of Environment it is Applied	Query Restriction	Scaling: provisioning or release of resources
(Naskos et al., 2015)	No	No	Heterogeneous	Not restricted	Provisioning of Resources
(Amazon Web Services, 2015)	No	No	Heterogeneous	Not restricted	Provisioning of Resources

2.6 Conclusion

This chapter presented the concepts of data warehouse and OLAP applications, SLA contract, data processing in databases and related works. Nowadays, many companies have migrated their applications and data to the cloud due to the benefits of this technology. Therefore, it is very important for users to choose "the best" cloud service provider, i.e. the provider most suitable for their needs. For this, it is important to know the QoS parameters of the SLA agreement.

From the user's point of view, the SRT (Service Response Time) parameter is considered one of the main QoS parameters. However, the major cloud providers have ignored or inappropriately treated the SRT parameter in SLA due to its complexity. Therefore, one of contributions of this work, presented in Chapter 3, is to propose a model for obtaining the SRT, so it can be treated adequately in SLA contracts.

In turn, we can observe that most works in the literature focus on queries with short execution time and on the prediction of the resources to be used for query processing through the current system context. These works may not be suitable in highly unpredictable environments on the availability of resources. Other related works focus on adaptive query processing. However, they present limitations of elasticity and/or scalability in their algorithms: (i) the absence of adaptive monitoring query processing; (ii) use of intrusive solutions and/or proprietary technology; and (iii) lack of formalism in the definition of the QoS parameters in their solutions. As a result, the same service may have different understanding among cloud service providers. Therefore, the main contribution of this thesis, to be presented in Chapter 4, is a new solution to efficient query processing on large databases available in a cloud environment. This solution must overcome some of the above limitations.

.

Chapter 3 – Service response time measurement model of service level agreements

3.1 Introduction

This chapter proposes a model for measuring a Service Response Time estimated for different request types on large databases available in a cloud environment. Therefore, to better understanding, this chapter is organized as follows:

3.2 Request definition: presents the formal definition of a request used in this work.

3.3 Service response time measurement model of service level agreements: presents the SRT measurement model, its definition and tools.

3.4 Case study - validation and results: presents the experiments of the proposed model using the Amazon EC2 cloud infrastructure, a TPC-DS like benchmark and finally, their results.

3.5 Conclusion: presents the final considerations of this chapter.

3.2 Request definition

In computational context, a request corresponds a task to be executed by a Web Service sent by a customer who has access to the service. The model request-response is the base of data communication on the Internet. Browsing a Web page is an example of request–response communication. For example, as shown in Figure 3-1, a customer submits a **request** message to a service of a Web server. The server provides the resources, it executes tasks and it returns a **response** message to the customer.

This work focuses on database access requests of OLAP applications in a cloud environment. This problem presents a lot of data processing. A request message is a SQL (Structured Query Language) query composed by one or more tables and it can be of different types. For example, select-range, select-aggregation, select-joins and select-sets-grouping-nesting-ordering. Therefore, in this work a request can be formally defined as follows:

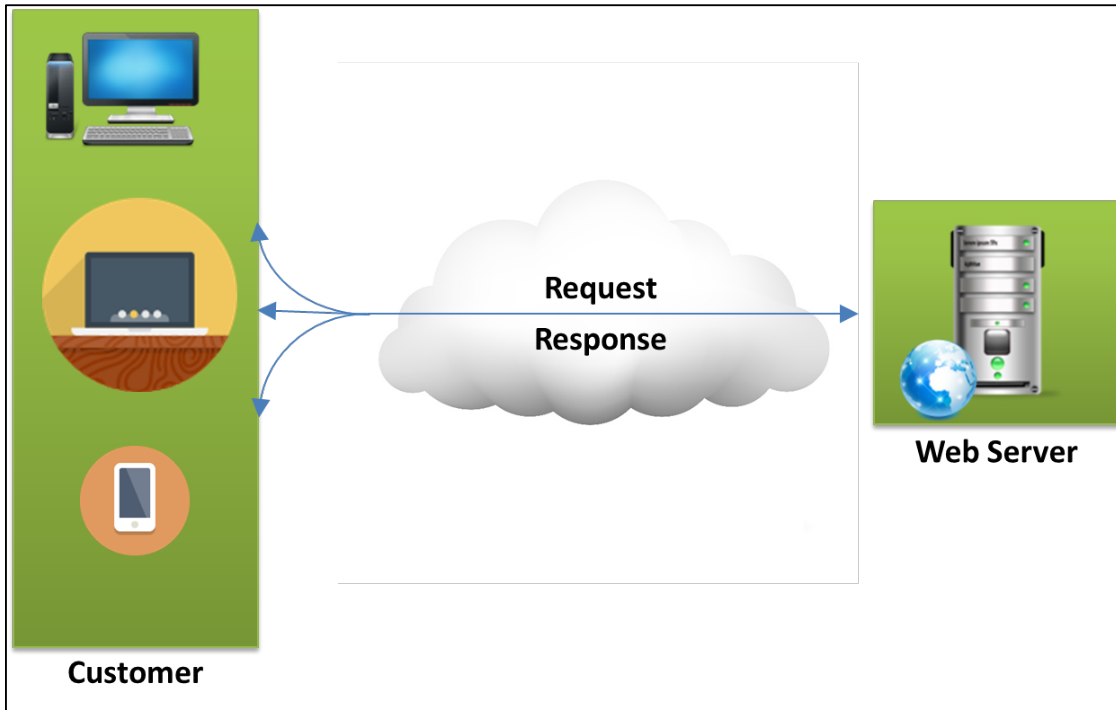


Figure 3-1. Request-response communication of the client-server computing model.

[WITH AS table]
SELECT [DISTINCT | ALL] | [OPER] < list of attributes> | <nested SQL query>
FROM <list of tables> | <nested SQL query>
[WHERE <predicate> | <nested SQL query> | [OPERCON]]
[UNION] | [INTERSECT] | [EXCEPT]
[GROUP BY <attributes>]
[HAVING <predicative>]
[ORDER BY attribute [ASC | DESC]];
[FECTH...]

[OPER] is an aggregation operator (AVG, for example), [OPERCON] is a set operator (UNION, INTERSECT OR EXCEPT, for example), [FETCH] is the operator to control the pagination of quantity of tuples returned and [WITH] is the operator responsible for generating a virtual view.

The most basic request of this work presents at least two SQL clauses: SELECT and FROM. This request is generic enough to accommodate the requests presented in the TPC-DS Benchmark (*Tpc Benchmark™ Ds*, 2012), which are used in the experiments of the proposed model. The TPC-DS is a decision support benchmark standard, it illustrates decision support systems that (i) examine large volumes of data; (ii) give answers to real-world business questions; (iii) execute queries of various operational requirements and complexities (e.g., ad-hoc, reporting, iterative OLAP, data mining); (iv) they are characterized by high CPU and IO load. Finally, (v) they are periodically synchronized with source OLTP databases through database maintenance functions.

To better understanding the proposed model described in next section, the requests were classified between three types, according to level of complexity. Therefore, the following subsections define each one of them.

3.2.1 Type 1 requests: select-range and select-aggregation

Type 1 requests represent the select-range and/or select-aggregation requests. Select-range are the database access requests that will return only tuples that are in a given range of a table. An index can be used to select the tuples. The range is used when a column, key or not, is compared with a constant using: =, <>, >, > =, <, <=, IS NULL, <=>, BETWEEN or IN. For example:

```
SELECT * FROM table WHERE key_column = 10;
SELECT * FROM table WHERE key_column BETWEEN 10 and 20;
SELECT * FROM table WHERE key_column IN (10,20,30);
SELECT * FROM table WHERE key_part1= 10 and key_part2 IN (10,20,30);
SELECT * FROM table WHERE date_column BETWEEN '2015-08-10' and '2015-09-10';
```

The select-aggregation requests are the database access requests that use aggregate operators for arithmetic expressions. For example, COUNT, SUM, AVG, MAX and MIN. They are applied to a set or multi-set of values and it returns the result of operation to the user. These operators can also be used in the HAVING clause (as shown in section 3.2.3). Examples of requests with aggregate operators:

```
SELECT MAX(column) FROM table WHERE key_column = 10;
SELECT column, AVG(column) FROM table WHERE column > 10;
SELECT MIN(key_column) FROM table;
SELECT COUNT(*) FROM table;
SELECT SUM(column), COUNT(column) FROM table;
```

3.2.2 Type 2 requests: select-joins

Type 2 requests represent the database access requests that uses one or more of the following operators: cross join, inner join, left outer join, right outer join or full outer join. A SQL join clause combines records from two or more tables in a relational database. It creates a set that can be saved as a table or used as it is (Elmasri & Navathe, 2010). These are examples of requests with joins:

```
SELECT * FROM table1, table2;
SELECT column1, column2 FROM table1 CROSS JOIN table2;
SELECT * FROM table1, table2 where table1.key_column = table2. key_column;
SELECT column1
        FROM table1 INNER JOIN table2 ON (table1.key_column =table2.key_column);
```

```
SELECT *
        FROM table1 LEFT OUTER JOIN table2 ON (table1.key_column = table2.key_column);
SELECT column1, column2
        FROM table1 RIGHT OUTER JOIN table2 ON (table1.key_column = table2.key_column);
SELECT *
        FROM table1 FULL OUTER JOIN table2 ON (table1.key_column = table2.key_column);
```

3.2.3 Type 3 requests: select-sets-grouping-nesting-ordering

Type 3 requests represent the database access requests that uses aggregation, joins, union, grouping and/or nesting operators. They can be UNION, INTERSECTION, EXCEPT, ANY, IN, UNIQUE, EXISTS, NOT EXISTS, GROUP BY, HAVING, ORDER BY or FETCH WITH. For example:

```
SELECT * FROM table1 UNION SELECT * FROM table2;

((SELECT * FROM table1 UNION ALL SELECT * FROM table2) UNION SELECT * FROM table3);

SELECT column FROM table1 INTERSECT SELECT column FROM table2;

SELECT column FROM table1 EXCEPT SELECT column FROM table2;

SELECT * FROM table1 WHERE column = 'WA' AND
EXISTS (SELECT column FROM table2 WHERE table2.key_column = table1.key_column);

SELECT column FROM table1 WHERE
NOT EXISTS (SELECT *
            FROM table2
            WHERE key_column = table1.key_column AND column = 'Name');

WITH Query_View (column1, column2, column3)
AS (
  SELECT column1, column2, column3 FROM table1 WHERE column1 IS NOT NULL
)
SELECT column1, column2, column3 FROM Query_View
GROUP BY column1, column2 ORDER BY column1;

SELECT column1, column2, column3
FROM table1
  INNER JOIN table2
    ON table1.key_column = table2.key_column
  INNER JOIN table3
    ON table2.key_column = table3.key_column
  INNER JOIN table4
    ON table3.key_column = table4.key_column
WHERE table1.column = 'Europe'
  AND table2.column IN(N'DE', N'FR')
  AND table3.column IN(287, 290, 288)
  AND SUBSTRING(table2.Name,1,4) IN (N'Vers', N'Spa')
GROUP BY table1.column1, table2.column
ORDER BY table1.column2, table1.column3;
```

```
SELECT COUNT(*)
FROM table1, table2, table3, table4
WHERE   table1.key_column = table2.key_column and
        table2.key_column = table3.key_column and
        table3.key_column = table4.key_column and
        table4.column = 8 and
        table3.column >= 30 and
        table2.column = 5 and
        table1.column1 = 'ese'
ORDER BY COUNT(*);
```

3.3 Service response time measurement model of service level agreements

The CSMIC consortium (Garg et al., 2013; Siegel & Perdue, 2012) emphasizes only a SRT (Service Response Time) parameter among QoS performance parameters. Its definition corresponds the time between the instant the request arrives at the provider and the instant it starts executing, i.e. the time that the service takes to start the execution of a request. However, other parameters should be considered.

Let $\mathrm{TSRT_R}$ be total execution time of a request R , i.e. the total time that a request takes to be executed in provider and its results to be presented to the user. Thus, $\mathrm{TSRT_R}$ of a request is composed by the sum of the times:

$$\boldsymbol{TSRT_R = SRT'_R + SRT''_R + SRT'''_R} \quad (3.1)$$

where SRT'_R is the service response time, i.e. the time that the service takes to start the execution of a request (CSMIC consortium definition), SRT''_R is the time of execution effectively of a request and SRT'''_R is the time that the result takes to be presented to user.

Ensuring the $TSRT_R$ parameter is a very difficult task because it depends on many factors. Moreover, due to the unpredictability of data traffic on the Internet, it becomes almost impossible to solve this challenge. In the literature, most researches focus on ensuring SRT''_R parameter, which is also the objective of this thesis. Thus, in this work, to better understanding the SRT''_R is called only SRT.

Therefore, in this work, the SRT corresponds to the time that a service takes to execute effectively a request. This way, the SRT of a service starts when a customer request is ready to execute and it finishes when the request executes effectively. Including, for example, startup time of virtual machine or wait of a fragment request, etc.

3.3.1 Recommended SRT definition

Nowadays, many companies have migrated their applications and data to the cloud due to the benefits of this technology. However, we can see that major cloud providers like Amazon ("AWS EC2 Service Level Agreement," 2015, "AWS S3 Service Level Agreement," 2015) and Google APP Engine (Sanderson, 2012) emphasizing availability, CPU instance and cost measure. Therefore, the SRT parameter is not completely handled or inappropriately treated in SLA. In order to ensure customer expectations relative to performance, cloud service providers have to understand how to incorporate suitably the SRT parameter in their SLA.

A contribution of this thesis is the proposal of a model for obtaining the SRT, so it can be treated adequately in SLA contracts. Thereby, it is necessary to define what a Recommended SRT is.

Let R_i be a database access request in a cloud, where i represents one of the following request types: (i) select-range and/or select-aggregation, (ii) select-joins or (iii) select-sets-grouping-nesting. The Average Service Response Time of a request R_i ($ASRT_{Ri}$) executed by n physical/virtual machines is given by:

$$\boldsymbol{ASRT_{Ri} = \sum_{Ri} SRT_{Ri}/n} \qquad \textbf{(3.2)}$$

in which SRT_{Ri} is the time between the moment a request R_i is ready to run and the service executes the request effectively.

Let A_{Ri} be a set of average service response times for all type i requests, i.e. $A_{Ri} = \{ ASRT^1_{Ri}, ASRT^2_{Ri}, ASRT^3_{Ri}, \dots, ASRT^k_{Ri} \}$, where k is the number of type i requests. Let $\hat{A}_{Ri}$ be a set of half the size of A_{Ri} ($k/2$) with the highest averages of A_{Ri}.

Thus, the Recommended SRT ($RSRT_{Ri}$) for a set of type i requests deployed in the cloud is given by median of $\hat{A}_{Ri}$:

$$\boldsymbol{RSRT_{Ri} = \uparrow \hat{A}_{Ri\frac{k}{2}+0.5} \quad for\ odd\ k} \qquad \textbf{(3.3.1)}$$

Or

$$\boldsymbol{RSRT_{Ri} = \frac{\left\{\uparrow \hat{A}_{Ri\frac{k}{2}} + \uparrow \hat{A}_{Ri\frac{k}{2}+1}\right\}}{2} \quad for\ even\ k} \qquad \textbf{(3.3.2)}$$

It is worth noting that Recommended SRT presents a pessimistic estimate of response time, because it is based on requests that require more time to process, i.e. on median of the upper half that represents the highest requests response time.

The discussion of Recommended SRT occurs in SLA construction phase (SLA Contract Definition), which evaluates several tasks of customer applications on the cloud service provider. The complex applications most used by a curtomer are defined and selected. In this work, the complex applications are those which use high load of system (large use of CPU and disk read/write).

3.3.2 SRT measurement model

A cloud computing platform is a cluster with hundreds or thousands of Computers (nodes) for data computing and storage. There are two types of nodes in the cluster: master nodes and slave nodes. Master nodes store metadata and manage all cluster slave nodes. The slave nodes store the data and their replicas for security.

In this context, Figure 3-2 shows the steps to obtain the Recommended SRT of a cloud computing platform: (1) acquisition of customer applications; (2) selection and classification of applications according to the request types: (i) select-range and/or select-aggregation, (ii) select-joins or (iii) select-sets-grouping-nesting; (3) experiments of customer applications deployed on master nodes and slave nodes of cloud provider; and finally, (4) analysis of results, which should define a Recommended SRT for each request type and system load.

It is worth noting that in contract level, the confidence and validation of the results will depend mainly on good practice in step 2, because good selectivity of customer applications will reduce SLA violation.

In step 3, to assist the tests, three tools were implemented and deployed in the cloud provider, they are COS (CPU Overload Simulator), DOS (Disk I/O Overload Simulator) and SRT Calculator. Following we detail each of them and their functions.

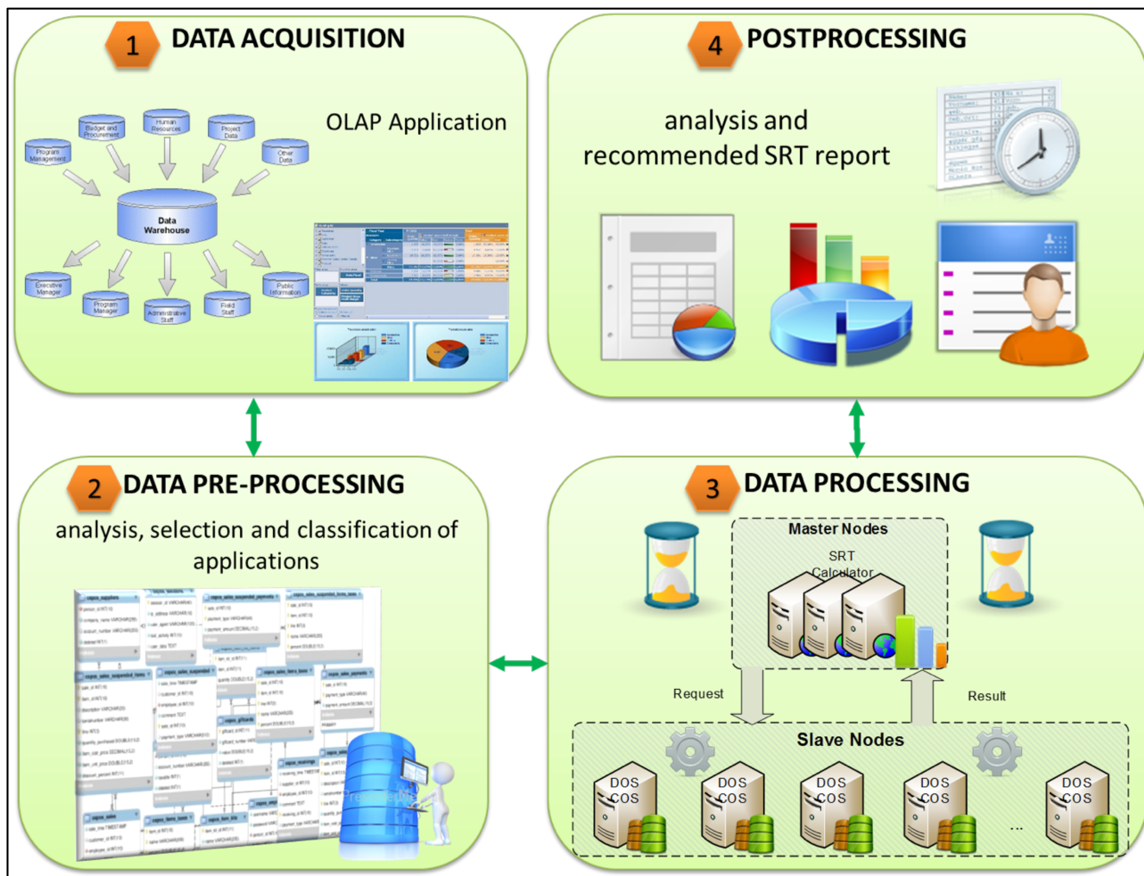


Figure 3-2. Steps to obtain the Recommended SRT.

COS: CPU Overload Simulator

The COS tool was deployed in slave nodes of a cloud service provider and is used to simulate partial and total CPU overload, i.e. the overload can also be by the processor core.

The COS tool generates an overload of threads of similar execution priority of the processes running in the operating system. Although the set of threads are running in the same process, if the COS tool executes itself more than once it will generate a set of threads in different processes causing large number of processes of equal priority competing for the processor. Thus, to overload the processor, the tool executes itself generating a large amount of processes, each having a large number of threads.

The tool allows serial execution by processor core, i.e. each core will be overloading by time. Thus, it allows configuring how many cores are overloaded. To analyze the CPU, the user can use the *sysstat* tool, which checks the processor usage in real time. The *sysstat* tool is a package with a collection of performance monitoring tools for major Linux distributions.

DOS: Disk I/O Overload Simulator

The DOS tool was deployed in slave nodes of provider and it serves to simulate disk read/write overload. Reading and writing is quantified in bytes read and written from/to disk.

The DOS tool generates an overload of threads of database access requests with similar execution priority of the processes running in operating system. Unlike COS tool, DOS simply run once, generating a very large set of threads of equal priority in the same process, overloading the system and competing with any another database access request that arrives at the processor.

The tool allows also defining the quantity of threads to be generated, in which each one simulates a database access request in the machine. This way, the tool allows a wide variation in quantity of bytes to read and write from/to disk.

The overloading in gigabytes is allowed as long as (i) the machine has enough main memory and (ii) the secondary disk has storage in terabytes, because temporary data can be written to disk in runtime of request. To analyze the disk read/write, the user can use the *dstat* tool, which allows to monitor the server resources in real-time. It is supported by most major Linux distributions such as RedHat, CentOS and Debian.

SRT Calculator

The SRT Calculator tool was deployed in master nodes of the cloud service provider and was used to execute the tests in the specified slave nodes. The SRT Calculator computes a set the Recommended SRT as defined in section 3.3.1 and generates a parameterized report to be analyzed and discussed between the cloud service provider and its customers.

The report presents the Recommended SRT for each request type and overload variation in slave nodes, through the COS and DOS tools. Beyond, for each request type, statistical parameters are generated from the set of the requests response times (usually values in nanoseconds), such as arithmetic average, sample variance, standard deviation, mode and coefficient of variation. Therefore, these parameters can be evaluated to validate the results. For the better understanding, the summary of SRT Calculator algorithm is shown below:

```
– Config_VM; //Configuration File of Physique/Virtual Machine (Slave Nodes).
– REQUEST-TYPE[i]; //Requests Type, i equals 1, 2 or 3.
1. BEGIN
2.     SLAVE-NODE[i..n] = Config_VM;
3.     FOR EACH SLAVE-NODE DO
4.         FOR EACH REQUEST-TYPE DO
5.             ExecuteRequest(SLAVE-NODE[i], REQUEST-TYPE [i]);
```

```
6.            ENDFOR
7.            REPORT(REQUEST-TYPE);
8.        ENDFOR
9.        REPORT(ALL-REQUEST);
10.
11.       VOID REPORT(REQUEST)
12.           BEGIN
13.             Avegare(); //(ns) – (ms) – (s) – (min)
14.             Sample Variance(); //(ns) – (ms) – (s) – (min)
15.             Standard Deviation(); //(ns) – (ms) – (s) – (min)
16.             Mode; //(ns) – (ms) – (s) – (min)
17.             Coefficient of Variation(); //(ns) – (ms) – (s) – (min)
18.             Recommended SRT (); //(ns) – (ms) – (s) – (min)
19.           END
20.   END
```

To use the SRT Calculator it is necessary to classify the customer applications in one of three requests types. In addition, a set of physical/virtual machines of the cloud must be selected to store customer applications. This way it is necessary to configure the following files: (1) network configuration file and database connection of slave nodes; (2) configuration file for requests with select-range and/or aggregating functions requests; (3) configuration file for requests with one or more joins; and finally, (4) configuration file for requests with set of operations, grouping and/or nesting.

Figure 3-3 shows the main GUI of SRT Calculator at the instant an experiment terminates. Next Section presents a case study of the proposed model using Amazon EC2 cloud infrastructure and TCP-DS, which was used to generate an OLAP database and some requests.

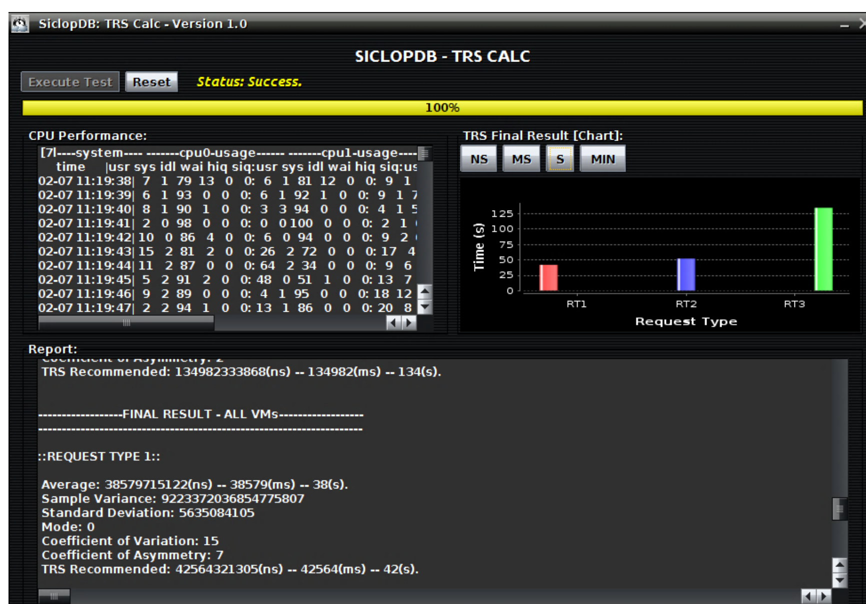


Figure 3-3. SRT Calculator – GUI Interface.

3.4 Case study – validation and results

This Section presents a case study of the proposed model to obtain the Recommended SRT utilizing small instances of Amazon EC2 cloud infrastructure. First, we present the environment and the experiments methodology. Then, we show the requests used and finally, we present the results obtained as well as its analysis.

3.4.1 Experimental environment

The tools (COS, DOS and SRT Calculator) were implemented in Java language using concurrent programming with threads and a Java API based on OpenMP - Open Multi-Processing (Bull & Kambites, 2000). They were deployed in the Amazon EC2 cloud infrastructure in small instances (homogeneous environment). Due to the limitations of Amazon, it was used 20 VMs (Virtual Machines), each one with an Intel Xeon Processor with turbo up to 3.3GHz, 1.7 GB of main memory and 160 GB of disk storage.

It was created an AMI (Amazon Machine Image) of VM with the database. This image allows startup a new VM quickly. The Amazon EBS (Elastic Block Store) was used to store the AMI. Therefore, VM startup and instantiation times were not considered. However, the time of network authentication and database connection were considered in experiments.

Each VM runs the Ubuntu 12.04 operating system and PostgreSQL 9.3 DBMS. This work focuses on OLAP applications with very large and complex database. Thus, the TPC-DS was used to generate a database of approximately 13 GB, fully replicated in each VM. Furthermore, 150 requests of several complexities were selected. Therefore, we consider the database and the generated requests as representative of customer applications.

3.4.2 Methodology

Figure 3-4 presents the methodology of the experiments. As shown, SRT Calculator tool was deployed in a master node chosen arbitrarily and it communicates with other VMs (slave nodes). Furthermore, the 150 requests were classified according to level of complexity between three types.

Thus, the SRT Calculator executes all requests of each type in all VMs, varying the overload on the slave nodes through using the COS and DOS tool (they were deployed in slave nodes). The PP (Processor Performance) represents the CPU overload levels. The DP (Disk Performance) verifies the overload of reading and/or writing on disk.

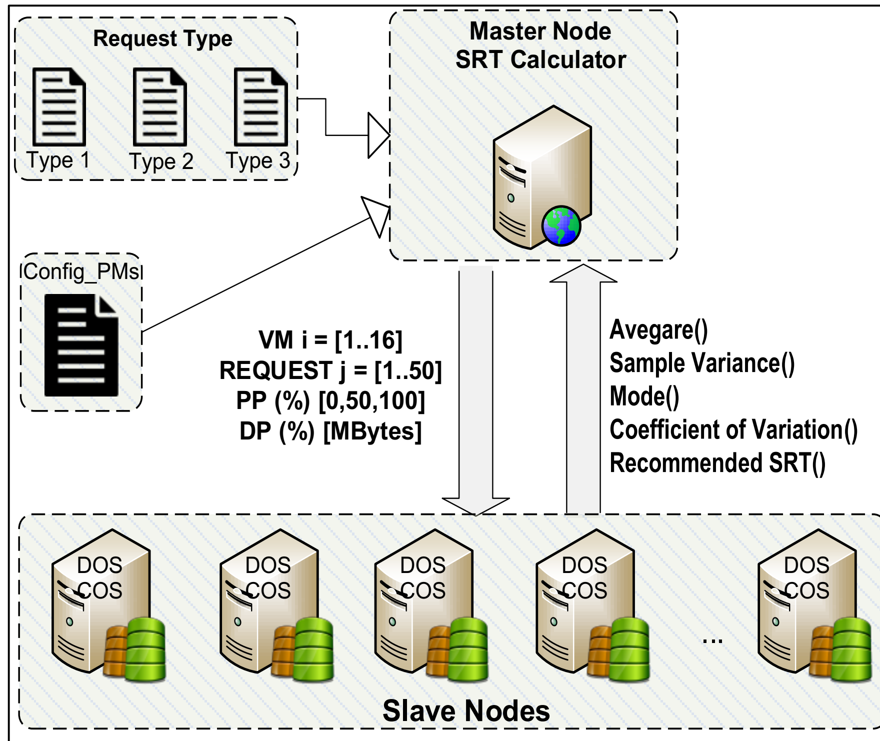


Figure 3-4. Methodology of experiments to obtain the Recommended SRT.

To view the rate of CPU and disk usage, the *sysstat* and *dstat* tools were used. Figure 3-5 shows a screenshot of the *sysstat* tool when three CPU cores are overloaded. Figure 3-6 shows a screenshot of *dstat* tool when approximately 20 Megabytes of data for reading and some Kilobytes of data for writing on disk are being used.

```
IFRN\1801682@ip166464l: ~
15:50:37     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
15:50:38     all   99,33    0,00    0,67    0,00    0,00    0,00    0,00    0,00    0,00
15:50:38       0  100,00    0,00    0,00    0,00    0,00    0,00    0,00    0,00    0,00
15:50:38       1  100,00    0,00    0,00    0,00    0,00    0,00    0,00    0,00    0,00
15:50:38       2   97,03    0,00    2,97    0,00    0,00    0,00    0,00    0,00    0,00

15:50:38     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
15:50:39     all   99,00    0,00    1,00    0,00    0,00    0,00    0,00    0,00    0,00
15:50:39       0  100,00    0,00    0,00    0,00    0,00    0,00    0,00    0,00    0,00
15:50:39       1  100,00    0,00    0,00    0,00    0,00    0,00    0,00    0,00    0,00
15:50:39       2   97,98    0,00    2,02    0,00    0,00    0,00    0,00    0,00    0,00

15:50:39     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
15:50:40     all   99,00    0,00    1,00    0,00    0,00    0,00    0,00    0,00    0,00
15:50:40       0  100,00    0,00    0,00    0,00    0,00    0,00    0,00    0,00    0,00
15:50:40       1  100,00    0,00    0,00    0,00    0,00    0,00    0,00    0,00    0,00
15:50:40       2   97,00    0,00    3,00    0,00    0,00    0,00    0,00    0,00    0,00

15:50:40     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
15:50:41     all   97,33    0,00    2,67    0,00    0,00    0,00    0,00    0,00    0,00
15:50:41       0   98,00    0,00    2,00    0,00    0,00    0,00    0,00    0,00    0,00
15:50:41       1   97,00    0,00    3,00    0,00    0,00    0,00    0,00    0,00    0,00
15:50:41       2   97,00    0,00    3,00    0,00    0,00    0,00    0,00    0,00    0,00
```

Figure 3-5. Processor Status through sysstat tool.

```
IFRN\1801682@ip166464l: ~
 91   9   0   0   0   0| 893M 9896k 2809M  264M|  23M  416k|30-01 15:52:48
 91   9   0   0   0   0| 889M 9896k 2812M  265M|  21M  364k|30-01 15:52:49
 79  10   6   6   0   0| 894M 9896k 2825M  246M|  13M   44k|30-01 15:52:50
----total-cpu-usage---- ------memory-usage----- -dsk/total- ----system----
usr sys idl wai hiq siq| used  buff  cach  free| read  writ|     time
 80   5   8   6   0   0| 893M 9896k 2843M  230M|  18M    0 |30-01 15:52:51
 94   6   0   0   0   0| 898M 9536k 2806M  262M|  26M  740k|30-01 15:52:52
 82   5   7   5   0   0| 907M 9536k 2800M  258M|  18M    0 |30-01 15:52:53
 96   4   0   0   0   0| 906M 9536k 2824M  236M|  23M    0 |30-01 15:52:54
 98   2   0   0   0   0| 901M 9536k 2847M  218M|  24M    0 |30-01 15:52:55
 95   5   0   0   0   0| 908M 9520k 2798M  261M|  22M  356k|30-01 15:52:56
 95   5   0   0   0   0| 904M 9528k 2823M  240M|  25M   12k|30-01 15:52:57
 94   6   0   0   0   0| 904M 9528k 2847M  215M|  25M    0 |30-01 15:52:58
 94   6   0   0   0   0| 902M 9368k 2809M  255M|  24M  828k|30-01 15:52:59
 96   4   0   0   0   0| 901M 9368k 2804M  261M|  22M  260k|30-01 15:53:00
 95   5   0   0   0   0| 901M 9368k 2801M  264M|  22M  432k|30-01 15:53:01
 93   7   0   0   0   0| 898M 9376k 2811M  258M|  23M  232k|30-01 15:53:02
 93   7   0   0   0   0| 884M 9376k 2833M  250M|  22M    0 |30-01 15:53:03
 94   5   0   0   0   0| 884M 9376k 2856M  227M|  23M    0 |30-01 15:53:04
 96   4   0   0   0   0| 883M 9264k 2846M  237M|  24M  528k|30-01 15:53:05
 95   4   0   0   0   0| 883M 9248k 2823M  261M|  24M  696k|30-01 15:53:06
 95   5   0   0   0   0| 883M 9248k 2820M  264M|  25M  432k|30-01 15:53:07
 94   6   0   0   0   0| 882M 9248k 2820M  264M|  23M  408k|30-01 15:53:08
 95   5   0   0   0   0| 890M 9248k 2825M  251M|  23M  232k|30-01 15:53:09
```

Figure 3-6. Disk Read/Write Status through dstat tool.

3.4.3 Used requests

This Section presents some of the requests used in the case study. The TPC-DS offers many database requests for experiments. For this case study, many requests from the TPC-DS were selected. The classification of each request was based on results of explain analyze command of the PostgreSQL DBMS.

Type 1 requests are select-range and/or select-aggregation requests. They have approximately 140,000 tuples of selectivity using the *catalog_sales* table of TPC-DS. In the following, some examples of type 1 requests used in the experiments are presented:

```
select * from catalog_sales where cs_item_sk between 1 and 1000;
select * from catalog_sales where cs_item_sk between 1001 and 2000;
select * from catalog_sales where cs_item_sk between 2001 and 3000;
select * from catalog_sales where cs_item_sk between 3001 and 4000;
select * from catalog_sales where cs_item_sk between 4001 and 5000;
```

Type 2 requests are select-joins requests and optional select-aggregation functions. The selectivity of these requests varied between 1000 and 60,000 tuples using at least 20 different tables of TPC-DS. In the following, some examples of type 2 requests used in the experiments are presented:

```
select  count(*)
from store_sales,household_demographics,time_dim, store
where ss_sold_time_sk = time_dim.t_time_sk
   and ss_hdemo_sk = household_demographics.hd_demo_sk
   and ss_store_sk = s_store_sk
   and time_dim.t_hour = 8
   and time_dim.t_minute >= 30
   and household_demographics.hd_dep_count = 5
```

```
    and store.s_store_name = 'ese'
order by count(*);

select  i_item_id, avg(ss_quantity) agg1, avg(ss_list_price) agg2, avg(ss_coupon_amt) agg3,
        avg(ss_sales_price) agg4
 from store_sales, customer_demographics, date_dim, item, promotion
 where ss_sold_date_sk = d_date_sk and
       ss_item_sk = i_item_sk and
       ss_cdemo_sk = cd_demo_sk and
       ss_promo_sk = p_promo_sk and
       cd_gender = 'M' and
       cd_marital_status = 'M' and
       cd_education_status = '4 yr Degree' and
       (p_channel_email = 'N' or p_channel_event = 'N') and
       d_year = 2001
 group by i_item_id
 order by i_item_id;


select  sum(cs_ext_discount_amt)  as "excess discount amount"
from
   catalog_sales,item,date_dim
where i_manufact_id = 577
and i_item_sk = cs_item_sk
and d_date between '1998-03-18' and
        (cast('1998-03-18' as date) + 90)
and d_date_sk = cs_sold_date_sk
and cs_ext_discount_amt
     > (
         select 1.3 * avg(cs_ext_discount_amt)
         from catalog_sales,date_dim
         where cs_item_sk = i_item_sk
          and d_date between '1998-03-18' and
                         (cast('1998-03-18' as date) + 90)
          and d_date_sk = cs_sold_date_sk
      ) ;
```

Type 3 requests are select-sets-grouping-nesting requests and, optional select-aggregation and select-joins. They present very complex query plans and its selectivity is between 100,000 and 200,000 tuples. It uses at least 20 different tables of TPC-DS. In the following, some examples of type 3 requests used in the experiments are presented:

```
WITH all_sales AS (
 SELECT d_year,i_brand_id,i_class_id,i_category_id,i_manufact_id
       ,SUM(sales_cnt) AS sales_cnt
       ,SUM(sales_amt) AS sales_amt
 FROM (SELECT d_year,i_brand_id,i_class_id,i_category_id,i_manufact_id
             ,cs_quantity - COALESCE(cr_return_quantity,0) AS sales_cnt
             ,cs_ext_sales_price - COALESCE(cr_return_amount,0.0) AS sales_amt
       FROM catalog_sales JOIN item ON i_item_sk=cs_item_sk
```

```
                    JOIN date_dim ON d_date_sk=cs_sold_date_sk
                    LEFT JOIN catalog_returns ON (cs_order_number=cr_order_number
                                              AND cs_item_sk=cr_item_sk)
      WHERE i_category='Shoes'
      UNION
      SELECT d_year,i_brand_id,i_class_id,i_category_id,i_manufact_id
           ,ss_quantity - COALESCE(sr_return_quantity,0) AS sales_cnt
           ,ss_ext_sales_price - COALESCE(sr_return_amt,0.0) AS sales_amt
      FROM store_sales JOIN item ON i_item_sk=ss_item_sk
                  JOIN date_dim ON d_date_sk=ss_sold_date_sk
                  LEFT JOIN store_returns ON (ss_ticket_number=sr_ticket_number
                                          AND ss_item_sk=sr_item_sk)
      WHERE i_category='Shoes'
      UNION
      SELECT d_year,i_brand_id,i_class_id,i_category_id,i_manufact_id
           ,ws_quantity - COALESCE(wr_return_quantity,0) AS sales_cnt
           ,ws_ext_sales_price - COALESCE(wr_return_amt,0.0) AS sales_amt
      FROM web_sales JOIN item ON i_item_sk=ws_item_sk
                 JOIN date_dim ON d_date_sk=ws_sold_date_sk
                 LEFT JOIN web_returns ON (ws_order_number=wr_order_number
                                       AND ws_item_sk=wr_item_sk)
      WHERE i_category='Shoes') sales_detail
 GROUP BY d_year, i_brand_id, i_class_id, i_category_id, i_manufact_id)
 SELECT  prev_yr.d_year AS prev_year,curr_yr.d_year AS year,curr_yr.i_brand_id
                    ,curr_yr.i_class_id,curr_yr.i_category_id,curr_yr.i_manufact_id
                    ,prev_yr.sales_cnt AS prev_yr_cnt,curr_yr.sales_cnt AS curr_yr_cnt
                    ,curr_yr.sales_cnt-prev_yr.sales_cnt AS sales_cnt_diff
                    ,curr_yr.sales_amt-prev_yr.sales_amt AS sales_amt_diff
 FROM all_sales curr_yr, all_sales prev_yr
 WHERE curr_yr.i_brand_id=prev_yr.i_brand_id  AND curr_yr.i_class_id=prev_yr.i_class_id
   AND curr_yr.i_category_id=prev_yr.i_category_id AND curr_yr.i_manufact_id=prev_yr.i_manufact_id
   AND curr_yr.d_year=2000 AND prev_yr.d_year=2000-1
   AND CAST(curr_yr.sales_cnt AS DECIMAL(17,2))/CAST(prev_yr.sales_cnt AS DECIMAL(17,2))<0.9
 ORDER BY sales_cnt_diff;

with wss as
 (select d_week_seq,
        ss_store_sk,
        sum(case when (d_day_name='Sunday') then ss_sales_price else null end) sun_sales,
        sum(case when (d_day_name='Monday') then ss_sales_price else null end) mon_sales,
        sum(case when (d_day_name='Tuesday') then ss_sales_price else  null end) tue_sales,
        sum(case when (d_day_name='Wednesday') then ss_sales_price else null end) wed_sales,
        sum(case when (d_day_name='Thursday') then ss_sales_price else null end) thu_sales,
        sum(case when (d_day_name='Friday') then ss_sales_price else null end) fri_sales,
        sum(case when (d_day_name='Saturday') then ss_sales_price else null end) sat_sales
 from store_sales,date_dim
 where d_date_sk = ss_sold_date_sk
 group by d_week_seq,ss_store_sk
 )
  select  s_store_name1,s_store_id1,d_week_seq1
       ,sun_sales1/sun_sales2,mon_sales1/mon_sales2
       ,tue_sales1/tue_sales2,wed_sales1/wed_sales2,thu_sales1/thu_sales2
       ,fri_sales1/fri_sales2,sat_sales1/sat_sales2
```

```
from
(select s_store_name s_store_name1,wss.d_week_seq d_week_seq1
       ,s_store_id s_store_id1,sun_sales sun_sales1
       ,mon_sales mon_sales1,tue_sales tue_sales1
       ,wed_sales wed_sales1,thu_sales thu_sales1
       ,fri_sales fri_sales1,sat_sales sat_sales1
 from wss,store,date_dim d
 where d.d_week_seq = wss.d_week_seq and
       ss_store_sk = s_store_sk and
       d_month_seq between 1200 and 1200 + 11) y,
(select s_store_name s_store_name2,wss.d_week_seq d_week_seq2
       ,s_store_id s_store_id2,sun_sales sun_sales2
       ,mon_sales mon_sales2,tue_sales tue_sales2
       ,wed_sales wed_sales2,thu_sales thu_sales2
       ,fri_sales fri_sales2,sat_sales sat_sales2
 from wss,store,date_dim d
 where d.d_week_seq = wss.d_week_seq and
       ss_store_sk = s_store_sk and
       d_month_seq between 1200+ 12 and 1200 + 23) x
where s_store_id1=s_store_id2
  and d_week_seq1=d_week_seq2-52
order by s_store_name1,s_store_id1,d_week_seq1;
```

3.4.4 Results

Following methodology presented in section 3.4.2, firsty, 50 requests of the same type are executed sequentially on all VMs. After, the experiments were repeated 5 times, erasing the DBMS cache for each repetition. Following, the same experiments were executed considering the overloaded CPU and finally considering the overloaded disk. Finally, the experiments were executed similarly for each request type.

Therefore, the results were grouped by type of request and overload variation in slave nodes. So, to type 1 Requests, the result of experiments on all VMs are presented in Figure 3-7. It shows the SRT averages to 50 requests executed on all VMs (all slave nodes) as well as when they are not overloaded (current) and when they are with CPU and Disk Overloaded. As discussed in the section 3.3, this work used the $\boldsymbol{SRT''_R}$ parameter, which is called only SRT.

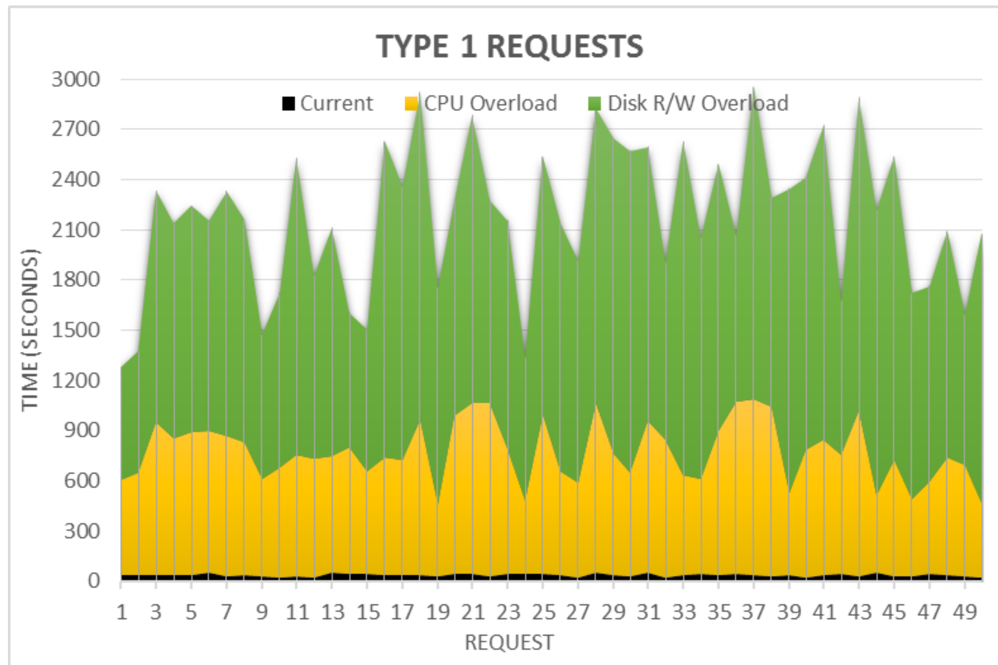


Figure 3-7. SRT averages on all VMs for type 1 requests.

Therefore, based on definition of Recommended SRT and considering that the processor and disk not overloaded (Current Status in Figure 3-7) we have the following result:

::TYPE 1 REQUESTS::
Average: 34,46(s)
Sample Variance: 71,02897959
Standard Deviation: 8,42786922
Mode: 35
Coefficient of Variation: 24,45696233
Recommended SRT: 42(s)

Overload with COS tool (CPU Overload in Figure 3-7), the result is as follows:

::TYPE 1 REQUESTS::
Average: 741,3(s)
Sample Variance: 32053,03061
Standard Deviation: 179,0336019
Mode: 620
Coefficient of Variation: 24,15130202
Recommended SRT: 865(s)

Overload with DOS tool (Disk R/W Overload in Figure 3-7), the following values were found:

::TYPE 1 REQUESTS::
Average: 1402,16(s)
Sample Variance: 134948,0555
Standard Deviation: 367,3527671
Mode: 1450
Coefficient of Variation: 26,19906196

Recommended SRT: 1718(s)

To type 2 requests, the result of experiments in all VMs are presented in Figure 3-8. It shows the SRT averages to 50 requests executed on all VMs (all slave nodes) when they are not overloaded (current) and when they are with CPU and Disk Overloaded.

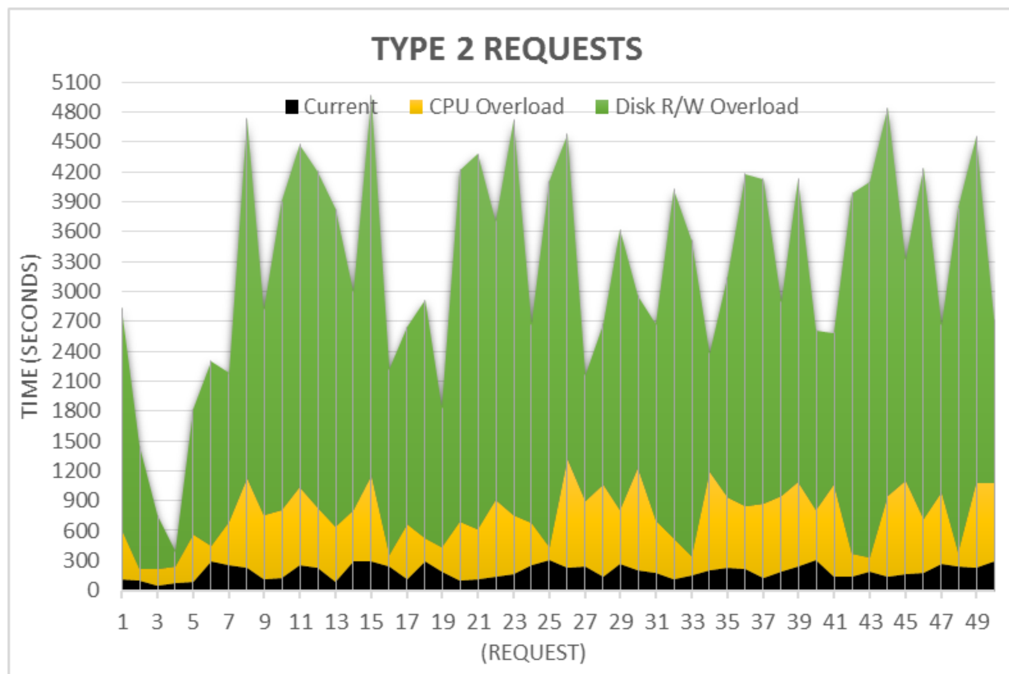


Figure 3-8. SRT averages on all VMs for type 2 requests.

Therefore, based on definition of Recommended SRT and considering the processor and disk not overloaded (Current Status in Figure 3-8) we have the following result:

::TYPE 2 REQUESTS::
Average: 187,6(s)
Sample Variance: 5059,755102
Standard Deviation: 71,13195556
Mode: 288
Coefficient of Variation: 37,91682066
Recommended SRT: 242(s)

Overload with COS tool (CPU Overload in Figure 3-8), the result is as follows:

::TYPE 2 REQUESTS::
Average: 567,86(s)
Sample Variance: 76106,16367
Standard Deviation: 275,8734559
Mode: 127
Coefficient of Variation: 48,58124466
Recommended SRT: 783(s)

Overload with DOS tool (Disk R/W Overload in Figure 3-8), the following values were found:

::TYPE 2 REQUESTS::
Average: 2514,8(s)
Sample Variance: 977864,7347
Standard Deviation: 988,8704337
Mode: 2618
Coefficient of Variation: 39,32203093
Recommended SRT: 3455(s)

To type 3 requests, the result of experiments on all VMs are presented in Figure 3.9. It shows the SRT averages to 50 requests executed on all VMs (all slave nodes) when they are not overloaded (current) and when they are with CPU and Disk Overloaded.

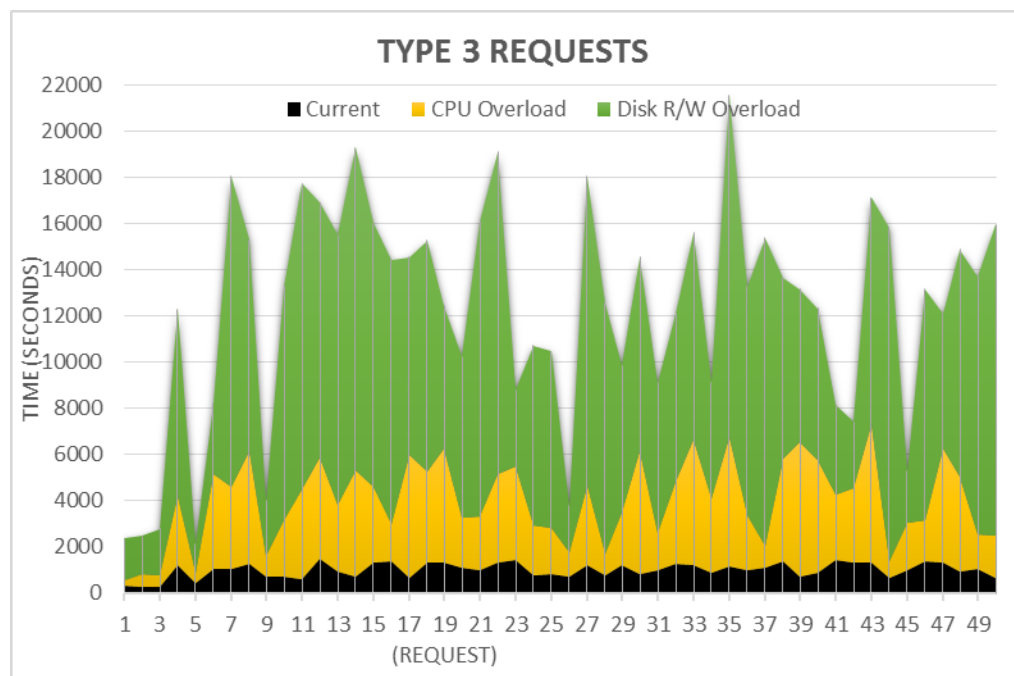


Figure 3-9. SRT averages on all VMs for type 3 requests.

Therefore, based on definition of Recommended SRT and considering that the processor and disk not overloaded (Current Status in Figure 3-9) we have the following result:

::TYPE 3 REQUESTS::
Average: 981,52(s)
Sample Variance: 106462,9486
Standard Deviation: 326,286605
Mode: 1001
Coefficient of Variation: 33,24299097
Recommended SRT: 1283(s)

Overload with COS tool (CPU Overload in Figure 3-9), the result is as follows:

::TYPE 3 REQUESTS::
Average: 3044,6(s)
Sample Variance: 2667600,653
Standard Deviation: 1633,279111
Mode: 2960
Coefficient of Variation: 53,64511301
Recommended SRT: 4431(s)

Overload with DOS tool (Disk R/W Overload in Figure 3-9), the following values were found:

::TYPE 3 REQUESTS::
Average: 8284,32(s)
Sample Variance: 16121155,85
Standard Deviation: 4015,11592
Mode: 8200
Coefficient of Variation: 48,46645133
Recommended SRT: 11391(s)

3.4.5 Analysis of results

Table 3-1 and Figure 3-10 summarizes the results of Recommended SRT. According to results, the SRT was higher when CPU or disk were overloaded, mainly the disk, which caused also overload in CPU.

Table 3-1. Recommended SRT Result.

Request Type	Recommended SRT		
	Current	*CPU Overload*	*Disk R/W Overload*
1	42(s)	865(s)	1718(s)
2	242(s)	783(s)	3455(s)
3	1283(s)	4431(s)	11391(s)

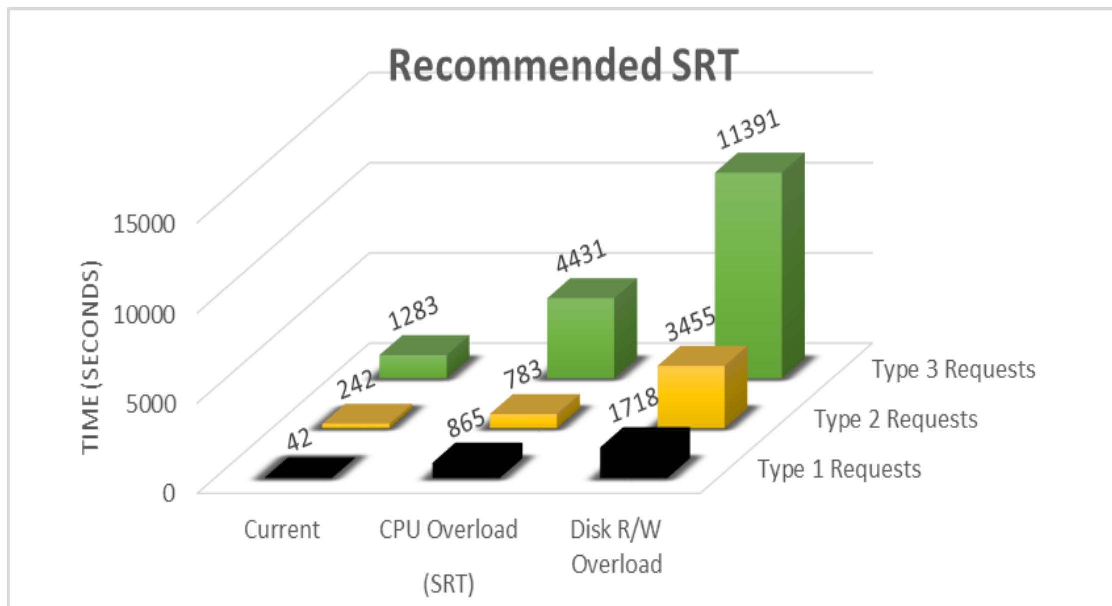


Figure 3-10. Recommended SRT Result.

It is worth noting that the number of rows and columns returned from a request (query selectivity) increases significantly the total time of its execution. For example, type 3 requests have very high selectivity and therefore, they have higher Recommended SRT. Other example, even with overloaded CPU, type 2 requests have Recommended SRT smaller than type 3 requests with current CPU utilization.

In general, type 1 and type 2 requests have smaller Recommended SRTs than type 3 requests. However, type 1 requests when overloaded CPU, its Recommended SRT is bigger than type 2 requests because the selectivity of type 1 requests is greater than type 2 requests.

The results obtained provide the basis for negotiation between the cloud service provider and its customers in order to establish an expected Service Response Time of their services. Furthermore, these values can be used by monitoring tools, when a limit value is achieved, the environment can react recovering or minimizing the consequences of SLA violation. For example, allocating, reallocating and/or releasing resources at run-time.

Therefore, an appropriately defined SRT brings benefits to both sides: the customers will have accurate information about the performance of their applications running in cloud and the provider will reduce penalties, since it knows the expected behavior of customer applications.

3.5 Conclusion

This chapter presented a measurement model that allows the cloud service provider and its customers to establish an appropriate SLA relative to SRT performance of its applications available in the cloud.

The proposed model is a non-intrusive solution and can be applied when companies wish to migrate their applications, OLAP or not, to cloud service providers, with the goal to allocate computational resources on demand, to ensure the quality of service in terms of Service Response Time. Finally, our proposed model focuses on OLAP applications with very large and complex databases. The model was evaluated using structured data of TPC-DS like Benchmark, considering that many cloud computing platforms support SQL requests directly or indirectly, this makes the proposed solution relevant for these kind of problems.

Chapter 4 – Efficient adaptive query processing on large database systems available in the cloud environment

4.1 Introduction

This chapter presents a new solution to efficient query processing on large databases available in the cloud environment. We present partitioning and monitoring strategies for adaptive processing of different types of queries (database access requests), a dynamic provisioning strategy and their algorithms. Furthermore, we present the SiclopDB framework, an implementation of the proposed solution and its architecture. Therefore, to better understanding, this chapter is organized as follows:

> **4.2 Estimated cost model**: presents the SLA violation cost and the total computational cost of a request used in this work.
>
> **4.3 Architecture**: presents the SiclopDB framework architecture and its components.
>
> **4.4 SiclopDB framework - components**: presents a new partitioning and monitoring strategies for adaptive processing of different types of requests in the cloud. Moreover, it shows the new dynamic provisioning strategy and their algorithms.
>
> **4.5 Conclusion**: presents the final considerations of this chapter.

4.2 Estimated cost model

To measure whether the SRT parameter is being violated or not, it is necessary to define the SLA violation cost per unit of time as well as the computational cost used for the provider to execute a user's request.

As defined in Chapter 3, in this work uses the $\boldsymbol{SRT''_R}$ parameter, called only $\boldsymbol{SRT_R}$. Therefore, let $\boldsymbol{T_s}$ be the start time of a request that arrives on cloud service provider. Thus, one of two situations may occur: (1) the request is ready to execute or (2) the request is waiting for a service to start its execution. When the request is ready to execute, $\boldsymbol{T_s}$ will be $\boldsymbol{0}$. Therefore, in this work the start time of a request corresponds to the time that it starts its execution effectively.

After its complete execution, the finish time $\boldsymbol{T_f}$ is obtained. Considering that the request can be partitioned, the complete execution is given when the last fragment arrives at the master node and the complete response is sent to the user.

Let $\boldsymbol{SRT_{Ri}}$ be the total execution time of a request $\boldsymbol{R_i}$, i.e. $\boldsymbol{SRT_{Ri} = T_f - T_s}$. Let $\boldsymbol{RSRT_i}$ the Recommended SRT for requests of type $\boldsymbol{i}$, i.e. the maximum time promised in which the service provider must execute a type $\boldsymbol{i}$ request. If $\boldsymbol{SRT_{Ri}}$ is bigger than $\boldsymbol{RSRT_i}$, the SLA has been violated. Therefore:

$$(SRT_{Ri} > RSRT_i) \rightarrow V_i \tag{4.1}$$

Let TV_i be the number of times that the request of type i violated the Recommended SRT and $TSRT_i$ the quantity of type i requests executed by a service. Therefore, as shown below, violations of recommended SRT is given by the percentage of times that response time was bigger than the maximum time promised. As will be shown in Section 4.4, this parameter is used to define the optimistic or pessimistic approach of complex requests execution.

$$PV_{Ri} = \frac{TV_i}{TSRT_i} \times 100 \tag{4.2}$$

Let C be the SLA violation cost per unit of time. If $V_i > 0$, the penalties of the provider are computed by cost per unit of time, multiplied by the time elapsed minus the maximum time promised.

$$VRSRT_i = max\{(SRT_{Ri} - RSRT_i) \times C, 0\} \tag{4.3}$$

in which $VRSRT_{Ri}$ represents the penalties of provider. The $\mathbf{max}$ function returns the maximum value between a set of values.

It is worth noting that SLA violation cost per unit of time C and the estimated costs must be presented at the construction step of a SLA. In addition, the result of SLA violations need to be discussed between the service provider and its customers.

The total computational cost for the provider to execute a user request is important to identify the lowest computational cost required by provider to execute a request in SRT time. Let $C_{MP_{Ri}}$ be the cost of main memory given by quantity in bytes of memory used per unit of time. Let $C_{CPU_{Ri}}$ be the cost of CPU given by percentage of CPU core used per unit of time. Finally, let $C_{DB_{Ri}}$ be the cost of database given by quantity in bytes of data disk pages retrieved per unit of time.

Considering that, each machine of slave node in cloud infrastructure has different costs. Therefore, the total cost of a machine M to execute a request R_i or to execute subpart of request R_i is given by:

$$C_{M_{Ri}} = C_{M_{MP_{Ri}}} + C_{M_{CPU_{Ri}}} + C_{M_{DB_{Ri}}} \tag{4.4}$$

A request $\boldsymbol{R_i}$ can use $\boldsymbol{k}$ slave nodes to complete its execution. Thus, the total computational cost ($\boldsymbol{TCC_{Ri}}$) is given by sum of costs of all slave nodes effectively used to execute $\boldsymbol{R_i}$:

$$\boldsymbol{TCC_{Ri} = C_{M_{Ri}^{1}} + C_{M_{Ri}^{2}} + C_{M_{Ri}^{3}} + \cdots + C_{M_{Ri}^{k}}} \tag{4.5}$$

Therefore, the total computational cost to execute effectively a request $\boldsymbol{R_i}$ is given by:

$$\boldsymbol{TCC_{Ri} = \sum_{k} C_{M_{Ri}^{k}}} \tag{4.6}$$

This work considers an environment with several machines having different performances. Consequently, each machines may have different costs and as the purpose of this work is to use the adaptive algorithm, the costs can change in query run-time depending on the necessity of provisioning or releasing of resources. As operations in this work are read-only, there are no costs associated in data updates and the data traffic between nodes in the cloud infrastructure.

It is worth noting that having many machines allocated minimizes the penalties of the provider, but can increase the computational cost. However, having a smaller amount of machines allocated reduces the computational costs, but can increase the penalty costs.

Therefore, a big challenge is to use the optimal number of machines to execute all requests in SRT time using lowest computational cost. The minimum cost for each request is the lowest computational cost to ensure the SRT. Considering a request with Recommended SRT of 100 seconds and after some previous analyses, it was obtained the minimum cost of 10. Then, the machines will execute the request in exactly 100 seconds, using suitable computing resources. The example is shown in Figure 4-1, which presents this approximation of the minimum cost and Recommended SRT corresponds to the ideal computational cost.

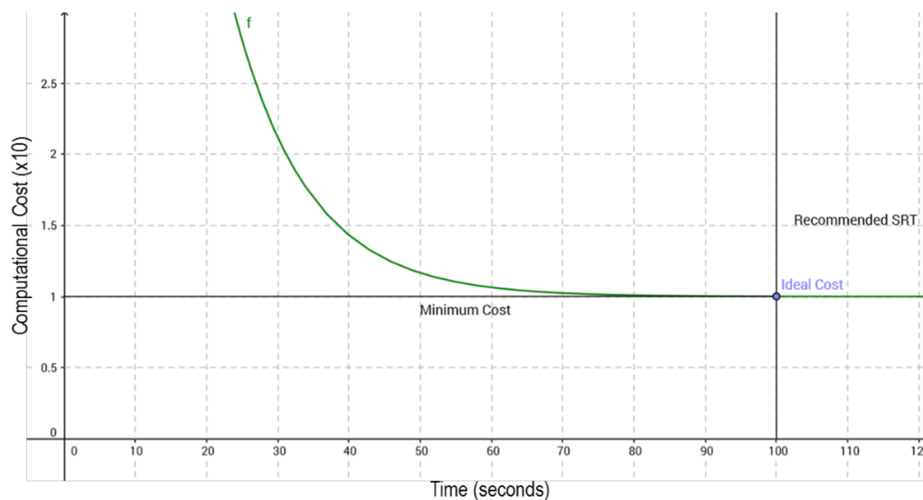


Figure 4-1. Ideal Computational Cost: Computation Cost (x10) vs Time (seconds).

4.3 Architecture

Based on the cloud computing infrastructure described in Figure 2-10 (query processing in the cloud), to obtain the elasticity in query processing, it is necessary an adaptive optimization algorithm. It must be implemented in the master nodes to manage the most appropriate allocation, reallocation or release of slave nodes resources in runtime of requests, according to the Recommended SRT and costs model already shown. This will maximize the SLA success probability.

Figure 4-2 presents the SiclopDB framework architecture as a new solution to efficient query processing on large databases available in the cloud environment. It integrates adaptive/dynamic re-optimization techniques by performing distributed queries in several steps, where each step concurrently executes a dynamic execution strategy at runtime of the queries, which will be presented in the next section.

Each component of the framework uses adaptive strategies during request runtime. Their costs are based on the $\boldsymbol{PV_{Ri}}$, $\boldsymbol{VRSRT_i}$ and $\boldsymbol{TCC_{Ri}}$ parameters defined in previous section and the SRT time is the $\boldsymbol{RSRT_{Ri}}$ presented in Chapter 3. The following presents the main components of the SiclopDB framework.

Dynamic Query Optimizer (DQO): It is used to construct an optimized query plan with objective of minimizing costs and maximizing the probability of success. The main difference for traditional optimizers is the construction of the query plan considering the SRT time restriction. For this purpose, that RSRT times agreed must be sufficient to execute the user's requests, observing the technological limits of the service provider.

Dynamic Query Scheduler (DQS): It is used to schedule the execution of distributed query plans. This component optimizes dynamically the queries at runtime, which is based on Service Response Time and the variation of resources utilized to process the query (for instance, average CPU utilization, available memory and estimated rates to processing of each slave node). Indeed, the queries submitted to DQS will be executed in the "best slave nodes". In this work, the definition of a "best nodes group" corresponds the group of slave nodes that possibly meets the Recommended SRT for a given request.

Dynamic Query Monitoring (DQM): Given an optimized and scheduled query plan, the aim of this component is to monitor the query execution. The monitoring verifies, periodically, the probability of a query to be executed in SRT time restriction. Therefore, the DQM reevaluates periodically all subqueries execution plans at runtime to check the possibility of SRT violation, whether the possibility is low, the query continues its execution, otherwise, the query will be re-optimized. The probability is estimated according to DBMS costs of slave nodes, slave nodes configurations, the query plan and a statistical table of metadata with Recommended SRT. The metadata serves as a cache, it stores information of previously executed queries on the provider (for instance, $\boldsymbol{SRT_{Ri}}$, $\boldsymbol{TCC_{Ri}}$ and $\boldsymbol{V_i}$). The metadata aids to reduce the computing overhead to calculate an estimated time to execute a query. Furthermore, the metadata will be automatically populated by the framework according to its use.

In the case that the request can not be executed before SRT time, the framework must calculate the execution time nearer to SRT time and the cloud provider must inform the penalties to be paid to customers. In this case, the traditional adaptive optimization algorithm will be executed because at this moment the fastest response time becomes more important than the SRT time. The next section presents in detail the partitioning and monitoring adaptive strategies, dynamic provisioning and the algorithms implemented in each of these components.

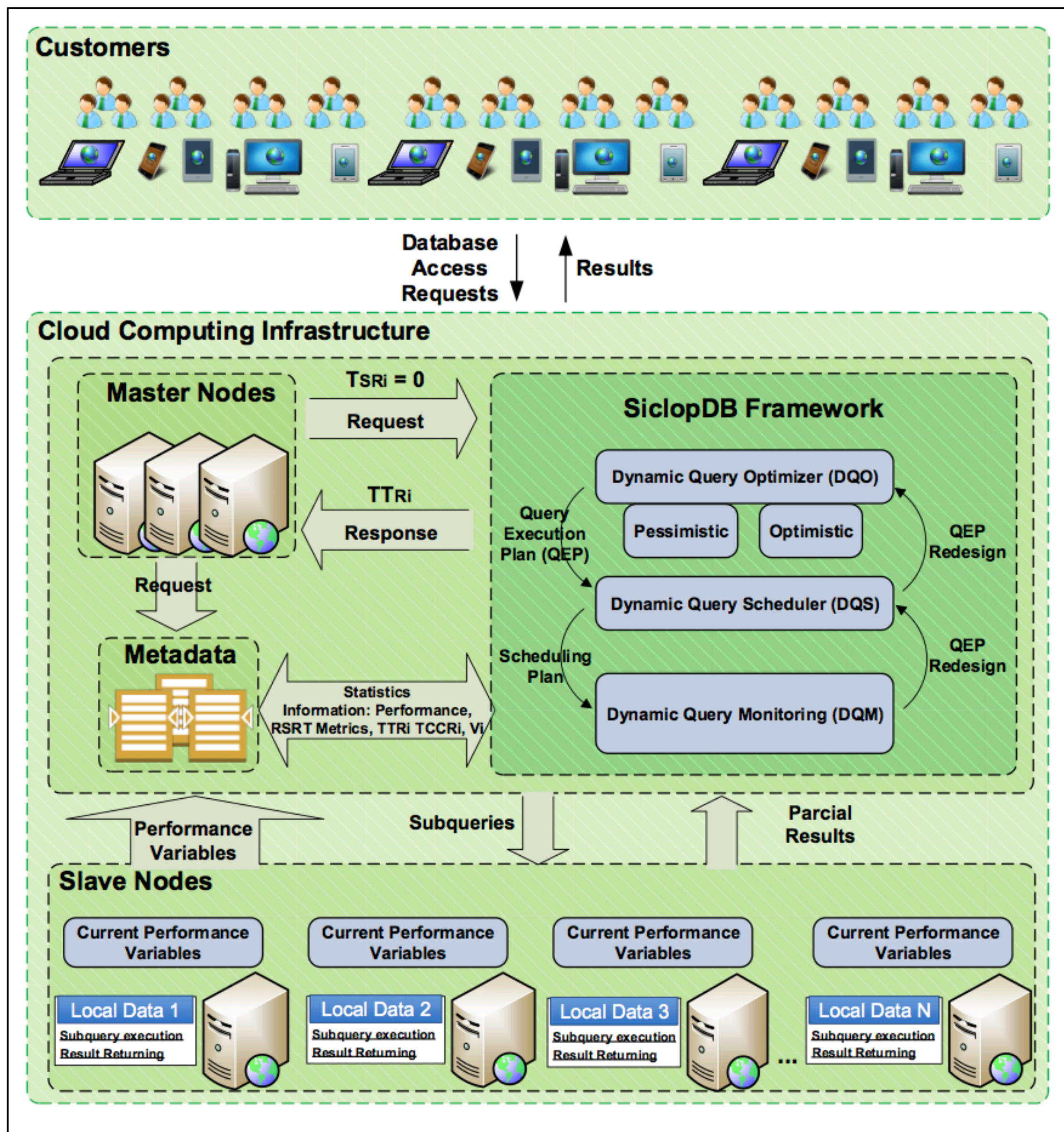


Figure 4-2. SiclopDB Framework Architecture.

4.4 SiclopDB framework – components

4.4.1 MetaData and performance

It is worth noting that before the effective execution of a request, it is replicated to the metadata server. The metadata main objective is to extract process and store information about the request that will be useful to its execution. Furthermore, the metadata monitors the real-time performance of each slave node with the aim to estimate query execution time. The following presents the main information of metadata:

(i) **Request Costs**: To estimate the cost of a request, this work uses the EXPLAIN

command that shows the query plan chosen by the DBMS optimizer. The query plan or query execution plan is the sequence of operations DBMS performs to run a request. The values obtained does not represent the correct estimated cost if the query is too complex, but it serves as a basis for estimating the request performance (*PostgreSQL 9.3.9 Documentation*, 2015; Riggs, Ciolli, Krosing, & Bartolini, 2015). This command returns the variables: cost, rows and width. The cost estimates are measured in units of disk I/O. An operator that reads a single block of 8.192 bytes (8K) from the disk has a cost of one unit. CPU time is also measured in disk I/O units, but usually as a fraction. For example, the amount of CPU time required to process a single tuple is assumed to be 1/100th (0,01) of a single disk I/O. Finally, the rows variable corresponds the number of tuples to be returned of a request and the width variable corresponds the quantity of bytes of each returned tuple. Therefore, the total cost is the sum of the quantity of disk pages to access the data plus the quantity of returned rows times 0,01, i.e. $\boldsymbol{cost = disk_pages \ + rows \times 0,01}$.

(ii) **Request Types (Range, Aggregation, Joins, Union, Grouping and Nesting Operators)**: As defined in Chapter 3, the requests executed in SiclopDB are classified between three types, according to complexity level: (1) type 1 requests represent the select-range and/or select-aggregation requests; (2) type 2 requests represent the database access requests that uses one or more of the following operators: equi join, cross join, inner join, left outer join, right outer join or full outer join; and finally, (3) type 3 requests that uses aggregation, joins, union, grouping and/or nesting operators. The result of classification is used to trace a request-profile that will be used by other requests in search of similar characteristics. Therefore, the explain command of DBMS can be used to obtain this information.

(iii) **Probability of SRT Violation**: Based on the requests of the similar characteristics that executed on the provider, it is calculated the probability of Recommended SRT violation. Let $\boldsymbol{PV_{Ri}}$ be the percentage of times that the response time of similar requests was bigger than Recommended SRT. If $\boldsymbol{PV_{Ri}}$ exceeds 50%, the query plan will take on a pessimistic approach, which consists to use more computational cost to decrease the probability of SRT violation. If $\boldsymbol{PV_{Ri}}$ does not exceeds 50%, the query plan of $\boldsymbol{R_i}$ will take an optimistic approach, which consists to use enough

computational cost to execute $\boldsymbol{R_i}$. Section 4.4.3 explains the use of these approaches.

(iv) **Performance Monitoring**: To get the current performance of a slave node the *iostat* (Layton, 2015; *System Analysis and Tuning Guide*, 2015) and *mpstat* tools (Russell & Cohn, 2012; *System Analysis and Tuning Guide*, 2015) were used. The *iostat* tool was used to check the disk saturation (Input/Output requests) and *mpstat* tool writes to standard output activities for each available processor core. These tools have many variables for monitoring the system performance (CPU utilization and device utilization report). However, this work used the variables *util*, *iowait* and *idle*. They are very important to identify problems of CPU and device saturation. *util* variable shows percentage of CPU time during which I/O requests were issued to the device (bandwidth utilization for the device). Device saturation occurs when this value is close to 100%. *iowait* variable shows the percentage of time that the CPU or CPUs were idle during which the system had an outstanding disk I/O request. *idle* variable shows the percentage of time that the CPU or CPUs were idle and the system did not have an outstanding disk I/O request. In metadata these values for each slave node are updated and stored at regular intervals. In SiclopDB, a slave node can be available or unavailable to execute a request. Therefore, the metadata analyzes the use of primary device bandwidth of each slave node through the *util* variable. Whether this percentage is above 80%, the slave node is unavailable for executing requests, because there is a high risk of not meet the expectations of query response. Following, the *idle* and *iowait* variables verify the average idle time and iowait for each CPU core. In the case, whether all CPU cores are below 10% (*idle*) and above 80% (*iowait*), the slave node is unavailable. Otherwise, the slave node is available to execute requests. The *iowait* depends on the number of CPU cores. A high *iowait* is an indicator of storage bottlenecks but not an indicator of storage saturation. This way, it was also used the *idle* variable. Finally, the information of availability or unavailability of a slave node is important to reduce the search overhead by slave nodes to execute a request, since that search overhead can be costly in infrastructures with many slave nodes.

4.4.2 Dynamic query optimizer (DQO)

This component is responsible to manage requests execution plan, based on the Recommended SRT and its type. In summary, Figure 4-3 shows the flowchart of possible execution plans to perform

a request.

For type 1 requests, in the initial provisioning, the request is partitioned and its subqueries are distributed according to the current performance of each slave node in order to have an execution plan that ensures the Recommended SRT. For this, it will use the metadata variables presented in previous section and partitioning strategies presented in Section 4.4.3.

During the execution of each partition, the monitoring checks the elapsed time and take one of two ways: (1) Estimating non SRT violation: the execution of other subqueries continues because the Recommended SRT is equal or greater than the elapsed time plus the remaining time (sum of estimated times of subqueries obtained from the initial provisioning. (2) Estimate SRT violation: the elapsed time plus the remaining time is greater than the Recommended SRT. Then, SiclopDB merges the remaining subqueries and executes a new provisioning. The Recommended SRT becomes the Recommended SRT minus elapsed time.

For type 2 requests, it is initially executed the partitioning of the request according to its simple nested loops (equi joins partitioning in SiclopDB) and if exists, its predicates. Then, each subquery is executed according to the type 1 requests. After processing all sub-queries, the result is unified in accordance with its joins.

Type 3 requests can be executed using a pessimistic or optimistic approach. The pessimistic approach is used when the $\boldsymbol{PV_{Ri}}$ of similar requests is greater than 50% and the optimistic approach when the $\boldsymbol{PV_{Ri}}$ is less than or equal to 50%.

Type 3 requests do not use monitoring nor adaptive partitioning during query execution. In the optimistic approach a greedy strategy is used, which choses only a slave node with sufficient capacity to execute the request in SRT time. In the pessimistic approach, the request is replicated to a set of slave nodes that can execute the request in SRT Time. The first slave node to execute the request signals the other nodes to abort their execution.

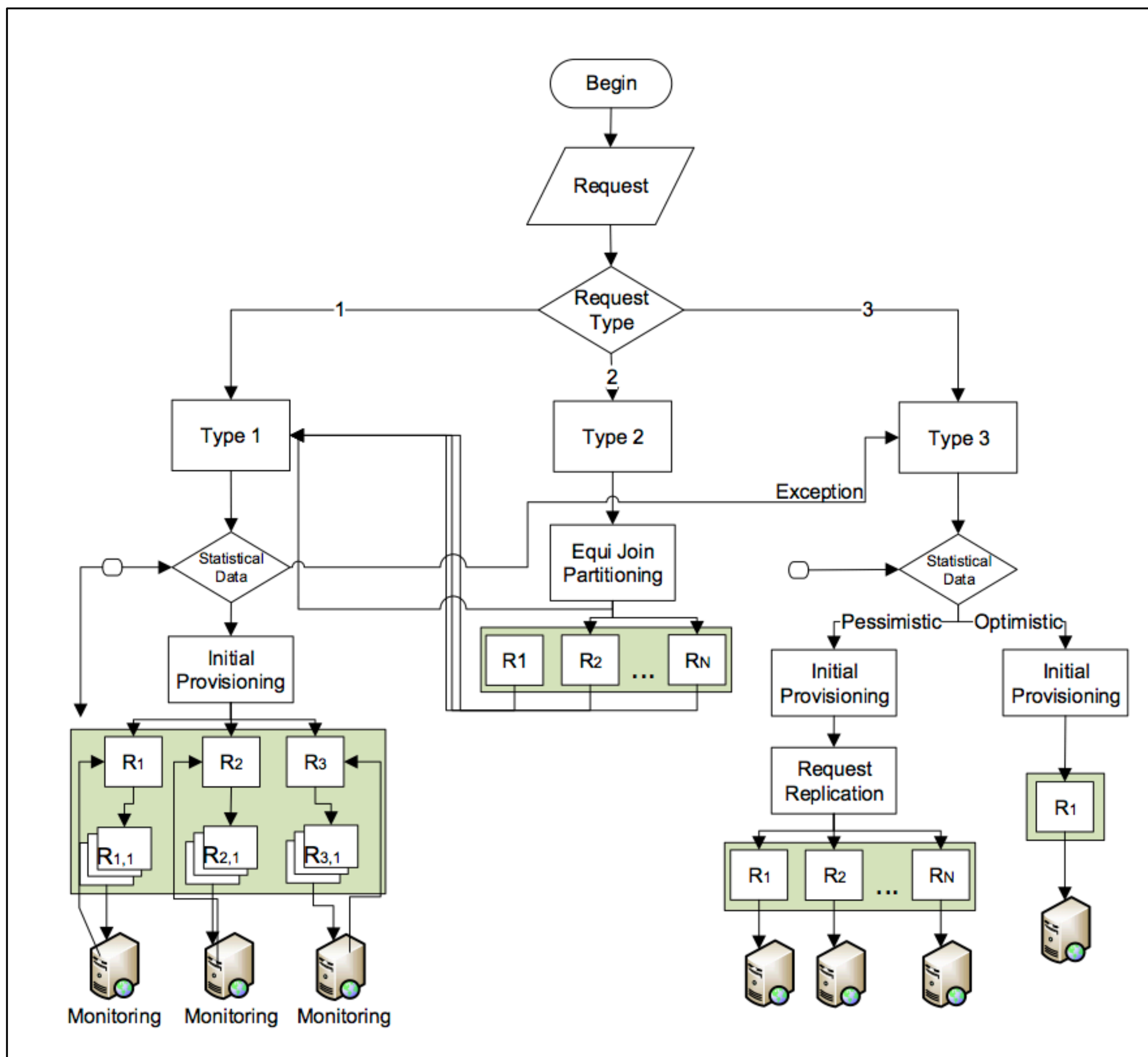


Figure 4-3. Flowchart of query processing in SiclopDB framework.

4.4.3 Dynamic query scheduler (DQS)

The DQS component is used to schedule the execution of distributed query plans. Thus, it distributes the partitions of a request to each slave node available based on its performance. This becomes a complex task in dynamic cloud environments where the performance of each machine can be different in time. This way, with the performance of slave nodes and database costs it is possible to estimate the expected performance of a slave node executing a partition. Consequently, it is possible to determine the appropriate number of partitions to split the request.

Let $\boldsymbol{T2R_{SN}}$ the *Tuple Read Rate*, the estimated time in seconds for a slave node to process a quantity of tuples:

$$\boldsymbol{T2R_{SN}} = \frac{\boldsymbol{rows \times 0,1}}{\boldsymbol{cost \times Svctm}} \quad (4.7)$$

where $\boldsymbol{rows}$ corresponds the number of tuples to be returned of a request, $\boldsymbol{cost}$ is estimated in units of disk I/O and $\boldsymbol{Svctm}$ the average service time (in seconds) for I/O requests that were issued to the device of a slave node. This last parameter can be obtained through the *iostat* tool.

To better understanding, consider a request R with Recommended SRT received by a cloud provider:

```
Select * // ← R
From Table T;
```

Consider that Recommended SRT is 100 seconds and through the explain command we have the cost = 368 and rows = 12.000. Moreover, consider that SN_1 is an available slave node and it has svctm = 13 milliseconds. Thus, $\boldsymbol{T2R_{SN1}}$ of SN_1 presents read rate of 250 tuples/second. Thus, SN1 meets the Recommended SRT because it was estimated that SN_1 in 100 seconds could process 25.000 tuples.

It is worth noting that the equation does not considers CPU overhead as well as the use of DBMS cache. However, it presents an estimate used only in the initial provisioning. Thus, at query processing, the $\boldsymbol{T2R_{SN}}$ is calculated by dividing the number of rows retrieved ($\boldsymbol{RT}$) by the time to retrieve them ($\boldsymbol{TRT}$).

$$\boldsymbol{T2R_{SN}} = \frac{\boldsymbol{RT}}{\boldsymbol{TRT}} \quad (4.8)$$

For complex queries, the strategy is similar to select-range queries. However, the rows variable is obtained by summing the number of accessed tuples by each query plan operator. Even if more than one operator uses these tuples and/or if these tuples are not part of the result. As well as select-range queries, this work considers that all access to a tuple block (on disk or temporary data pagination) is an I/O cost.

It is worth noting that this estimate does not consider the CPU overhead. However, the overhead of temporary data pagination is considered, since it does not distinguish the repetition of tuples during each step of the query execution plan.

Therefore, if we have the current speed of tuples read rate per second of a slave node, it is possible to partition a request in accordance with the estimated time to execute the request on each node. In

order to not violate the SRT, the sum of the times for each partition to execute a subquery, according to the times of each Slave Node (SN), it has to be less than the Recommended SRT.

$$\boldsymbol{RSRT_{Ri} \geq T2R_{SN1} + T2R_{SN2} + \cdots + T2R_{SNk}} \qquad (4.9)$$

In this work, the partitioning strategy depends on the type of request and we consider that all tables are clustered by primary key. Following, it will be presented the strategies implemented in the DQS component for each request type.

Type 1 Request: Assume that a cloud provider receives the following select-range request R with Recommended SRT:

```
SELECT * // ← R
FROM table T
WHERE T.pk >= 1000 and T.pk < 5000;
such that pk is the primary key of table T.
```

Considering that primary key values of T are sequential, without gaps between values, then we can extract rows = 4.000 tuples. Besides, consider that Recommended SRT is 100 seconds and that initial provisioning is a single slave node (SN_1) such that the current moment $\boldsymbol{T2R_{SN1}}$ = 20 tuples/sec.

Consequently, the initial provisioning using only SN_1 will bring a penalty to be paid by the provider because it was estimated that SN_1 in 100 seconds will process in 2.000 tuples. In this case, it is necessary to allocate a new slave node (SN_2) to help out. Assume that $\boldsymbol{T2R_{SN2}}$ = 10 tuples/sec then only 1.000 tuples can be processed in 100 seconds. Then, a new slave node (SN_3) is required to process the request. Then, consider $\boldsymbol{T2R_{SN3}}$ = 10 tuples/sec.

At this point, it is possible that three slave nodes are sufficient to process R and ensure the Recommended SRT. R is rewritten in three subqueries: R_1, R_2 and R_3, the first one is executed in SN_1, the second one in SN_2 and the third one in SN_3, respectively. Note that in this case a virtual partitioning is used (i.e. we partition using the predicate of the primary key) to divide R in R_1, R_2 and R_3.

```
SELECT * // ← R1
FROM table T
WHERE T.pk >= 1000 and T.pk < 3000;
```

```
SELECT * // ← R2
FROM table T
WHERE T.pk >= 3000 and T.pk < 4000;
```

```
SELECT * // ← R3
FROM table T
WHERE T.pk >= 4000 and T.pk < 5000;
```

Using only three slave nodes does not guarantee that the quality defined in SRT will be met, because the cloud environment is unstable and the performance of the nodes can change during the queries execution. Therefore, it is indispensable to use a proactive approach based on statistical data in metadata. For this, the DQM component is used. DQM partitions the queries, in such a way that the performance of the nodes can be monitored at a frequency that allows other nodes to be added when necessary in order to ensure the Recommended SRT.

An important issue is the monitoring frequency. If too frequent, the original query would have to be partitioned into many subqueries. Thus, the overload added could prejudice more than help. If monitoring is infrequent, it may be difficult to make corrections in a timely manner and avoid possible penalties.

The partitioning process in DQM uses historical data about the request containing information about how long it was necessary to process similar requests (same type of request), including the number of partitions used. From this information, it is possible to efficiently monitor the request execution. The next section presents the strategies of monitoring.

Consider that, for example, for similar requests 2 partitions were used for each partition of the initial provisioning. Then, R_1 is partitioned in two requests:

```
SELECT * // ← R1,1
FROM table T
WHERE T.pk >= 1000 and T.pk < 2000;
```

```
SELECT * // ← R1;2
FROM table T
WHERE T.pk >= 2000 and T.pk < 3000;
```

When $R_{1,1}$ is done, we have the first opportunity to monitor the query execution performance in a non-intrusive way. Consider that 70 seconds were spent to execute $R_{1;1}$. This means that the performance $\boldsymbol{T2R_{SN1}}$ was below of predicted, which leads to a completion time with the expected processing of the next subquery of 140 seconds. However, this value is above the Recommended SRT. Thus, it starts a revision of the initial provisioning that allows for the SRT to be satisfied. Before reviewing, the remaining partitions will be merged in a single query.

In this case, a solution is to relocate the remaining subquery to another slave node. Consider a slave node (SN_4) such that $\boldsymbol{T2R_{SN4}}$ = 30. Thus, all the 1.000 remaining tuples can be read by SN_4 in 30 seconds in the best-case scenario, and that does not lead to a violation of Recommended SRT. To monitor the request execution, the query is partitioned in two, each of the following way. This partitioning strategy is presented in section 4.4.4:

```
SELECT * // ← R1,2,1
FROM table T
WHERE T.pk >= 2000 and T.pk < 2500;
```

```
SELECT * // ← R1,2,2
FROM table T
WHERE T.pk >= 2500 and T.pk < 3000;
```

Consider that the performance is stable and it is able to finish its workload on schedule. Thus, the same strategy can be applied in the processing of R_2 in SN_2 and R_3 in SN_3. This partitioning method using the primary key as the partitioning attribute is the same for similar select-range requests and similar requests with aggregation.

Consider a new scenario, now with the increased performance of the slave node. For example, suppose R_2 request was partitioned into, for example, 2 subqueries. Imagine that after the first subquery ends and the monitoring starts, it is discovered it has executed in a shorter time than expected, possibly because of some other processes in the slave node that finished, and then SN_2 can finish processing R_2 before planned. Thus, this time off can be used to process some requests that SN_1 can not run. These queries can be allocated to SN_2 in order to satisfy the Recommended SRT. Although this is a simple example and with some assumptions, the solution given by this work deals with scenarios of reduced and increased performance of slave nodes.

It is important to note that all monitoring and setting is made in a non-intrusive way, i.e. our solution does not depend on technology used by the provider. For example, database version, operating system etc. Therefore, the slave nodes and their respective DBMSs do not require any changes to be used by framework.

Now consider that a cloud provider receives the following request R with Recommended SRT:

```
SELECT * // ← R
FROM table T;
```

Also considering that values of the primary key of T are sequential, without gaps between the values, then we extract rows = 8.000 tuples. Also, consider that Recommended SRT is 100 seconds and that the initial provisioning is a single slave node (SN_1) such that now $\boldsymbol{T2R_{SN1}}$ = 20 tuples/sec.

Consequently, as in the previous example, it will bring a penalty to be paid by the provider. Thus, a new slave node (SN_2) is allocated to help. Consider that $\boldsymbol{T2R_{SN2}}$ = 20 tuples/sec then, as in SN_1, 2.000 tuples can be processed in 100 seconds. Then a new slave node (SN_3) is required to process the query. Then, consider $\boldsymbol{T2R_{SN3}}$ = 40 tuples/sec.

Thus, it is possible that those three slave nodes are sufficient to process R and ensure the Recommended SRT. R is rewritten into three subqueries: R_1, R_2 and R_3, the first one is executed in the first SN_1, the second one in SN_2 and the third one in SN_3, respectively. As in the previous example, a virtual partition is created. However, the request is rewritten adding a range predicate on the table's primary key (Vmin and Vmax primary key) as shown below:

```
SELECT * // ← R1
FROM table T
WHERE T.pk >= Vmin and T.pk < Vmin + 2000;

SELECT * // ← R2
FROM table T
WHERE T.pk >= Vmin + 2000 and T.pk < Vmin + 4000;

SELECT * // ← R3
FROM table T
WHERE T.pk >= Vmin + 4000 and T.pk < Vmax + 1;
```

such that Vmin and Vmax is the minimum and the maximum value of the primary key of Table T, respectively.

After the query is rewritten, we use the partitioning methodology described previously. The monitoring and provisioning of slave nodes to process the rewritten query is made the same way.

Now consider that a cloud provider receives the following request R_1 or R_2 with Recommended SRT:

```
SELECT * // ← R1
FROM table T
WHERE T.pk > <<value>>;
```

such that pk is the primary key of table T.

or

```
SELECT * // ← R2
FROM table T
WHERE T.pk < <<value>>;
```

such that pk is the primary key of table T.

For this example, the request is rewritten adding a range predicate on the table's primary key obtaining the queries below:

```
SELECT * // ← R1
FROM table T
WHERE T.pk > <<value>> and T.pk < Vmax + 1;
```

or

SELECT * // ← R_2
FROM table T
WHERE T.pk >= Vmin and T.pk < <<value>> + 1;

Therefore, after the query is rewritten, partitioning strategies described previously are used. The monitoring and provisioning of slave nodes to process the rewritten query is made the same way.

Now consider that a cloud provider receives the following request R with Recommended SRT:

SELECT * // ← R
FROM table T
WHERE T.attr > <<value>> and T.attr < <<value>>;
such that attr is not a primary key of table T.

Different from the previous cases in which an index can be trivially used in execution of each partition, in this case the query plan of each partition is a linear scan on the table. Creating an index for each attribute present in the request predicate is not viable. Therefore, the proposed solution is to rewrite the request the same way to the previous solutions, however, using the primary key for partitioning, as shown below:

SELECT * // ← R
FROM table T
WHERE T.attr > <<value>> and T.attr < <<value>> and T.pk >= Vmin and T.pk < Vmax+1;

After the request is rewritten, the partitioning methodology in the primary key is used the same way as well as the monitoring and provisioning to process the rewritten query. It is worth noting that if all previous cases did not use the primary key, they would be rewritten in a similar way to this strategy.

Now consider that a cloud provider receives the following request R with Recommended SRT:

SELECT * // ← R
FROM table T
WHERE T.attr = <<value>>;
such that attr is not the primary key of table T.

In this case is worth noting that R is not a select-range request. However, it is not as complex as type 3 requests. Hence, R is an exception and the strategy is different from the previous ones. The strategy is, in the initial provisioning, to seek the set of slave node with $\boldsymbol{T2R}$ enough to process the request that ensures the Recommended SRT. Besides, consider that Recommended SRT is 100 seconds and total rows in a linear scan on the table is 8.000 tuples. If the *attr* attribute was a primary key and there was no composite primary key in table T, rows assume the value 1.

Now consider three slave nodes, SN_1, SN_2 and SN_3, with $\boldsymbol{T2R_{SN1}}$ = 400 tuples/sec, $\boldsymbol{T2R_{SN2}}$ = 20 tuples/sec, $\boldsymbol{T2R_{SN3}}$= 100 tuples/sec, respectively. Following the optimistic approach, the algorithm does a search in the slave nodes and executes the query in first slave node sufficient in such a way to execute R within the Recommended SRT. Thus, in this case, $\boldsymbol{T2R_{SN1}}$ is chosen. If the pessimistic approach of the algorithm is active, R will be executed in half of slave nodes with the highest $\boldsymbol{T2R}$ that ensures the SRT. The first slave node that finishes the request execution throws a signal to others slave nodes to abort their execution. Consequently, it reduces the risk of penalty to be paid by the provider. These strategies are also used for Type 3 requests.

Monitoring the slave node to process this type of request is made after its processing and then the metadata is updated with success or failure of request execution. Thus, as in type 3 requests, this exception of type 1 request does not use monitoring nor adaptive partitioning during query execution.

For requests with aggregation operators, it is added a predicate on the table's primary key, being used as a partitioning attribute. If the operation is distributive such as SUM, COUNT, MIN or MAX, it can be easily rewritten analogously to the previous examples (i.e. a linear scan predicate of the table is added).

Consider for example that a cloud provider receives the following request R with Recommended SRT:

```
SELECT Dist_Oper(*) // ← R
FROM table T;
```

Then the request is rewritten as follows:

```
SELECT Dist_Oper(*) // ← R
FROM table T
WHERE T.pk >= Vmin and T.pk <Vmax + 1;
such that pk is the primary key of table T.
```

Considering that, values of the primary key of T are sequential, without gaps between the values. After the query is rewritten, we use the partitioning strategies described previously. The monitoring and provisioning of slave nodes to process the rewritten query is made the same way.

Whether the request presents an algebraic aggregation operator, such as AVG, the result of the original query can not be easily obtained by means of partitions. With that, we transform the algebraic function into distributive functions, as for instance AVG, it is possible to transform into

SUM and COUNT, so that the result of the original request. Consider for example that a cloud provider receives the following request R with Recommended SRT:

```
SELECT AVG(T.attr) // ← R
FROM table T;
```

Then the request is rewritten as follows queries:

```
SELECT SUM(T.attr) into VSum // ← R1
FROM table T;
SELECT COUNT(T.attr) into VCou // ← R2
FROM table T;
```

Then R_1 and R_2 are rewritten again as the previous example. Like this:

```
SELECT SUM(*) // ← R1
FROM table T
WHERE T.pk >= Vmin and T.pk <Vmax + 1;
such that pk is the primary key of table T.
SELECT COUNT(*) // ← R2
FROM table T
WHERE T.pk >= Vmin and T.pk <Vmax + 1;
such that pk is the primary key of table T.
```

Therefore, after the query is rewritten, it is used the partitioning strategies already described. The monitoring and provisioning of slave nodes to process the rewritten query is made the same way.

Type 2 Request: For requests with joins, DQS rewrites the query, separating all tables of FROM clause. Currently, the component allows only equi joins requests. Consider that a cloud provider receives the following trivial select-join request R with Recommended SRT:

```
SELECT * // ← R
FROM table T1, table T2
WHERE T1.fk = T2.pk;
such that T1.fk the foreign key referenced by the primary key T2.pk.
```

The request R is rewritten in two subqueries, R_1 and R_2:

```
SELECT * // ← R1
FROM table T1;
SELECT * // ← R2
FROM table T2;
```

In this case, R_1 and R_2 will be executed utilizing strategies of type 1 requests. Thus, R_1 and R_2 are rewritten as follows:

```
SELECT * // ← R1
```

```
FROM table T1
WHERE T1.pk >= Vmin and T1.pk <Vmax+1;
```

```
SELECT * // ← R2
FROM table T2
WHERE T2.pk>=Vmin and T2.pk<Vmax+1;
```

Thus, it uses the partitioning methodology described for type 1 requests, as well as the monitoring and provisioning of slave nodes to process the rewritten queries. After the execution of all partitions, the slave node that executed R_1 makes the join to present the result. Therefore, it uses a similar algorithm to nested loops join algorithm. The R_1 is chosen as the outer table, or the driving table. The other table is called the inner table. For each row in the outer table, the algorithm finds all rows in the inner table that satisfy the join condition. Finally, it combines the data in each pair of rows that satisfy the join condition and returns the resulting rows.

Now consider that a cloud provider receives the following select-join request R with Recommended SRT:

```
SELECT * // ← R
FROM table T1, table T2
WHERE T1.fk = T2.pk and T1.pk = <<value>>;
such that T1.fk the foreign key referenced by the primary key T2.pk,
```

The request R is rewritten in two subqueries, R_1 and R_2 as follows:

```
SELECT * // ← R1
FROM table T1
WHERE T1.pk = <<value>>;
```

```
SELECT * // ← R2
FROM table T2;
```

One more time, R_1 and R_2 will be executed using strategies of type 1 requests. Thus, R_2 is rewritten as follows:

```
SELECT * // ← R2
FROM table T2
WHERE T2.pk>=Vmin and T1.pk<Vmax+1;
```

Thus, we use the partitioning methodology described previously and the monitoring and provisioning of slave nodes to process the rewritten query. Finally, after the execution of all partitions the slave node that executed R_1 makes the join to present the result.

Type 3 Request: For complex requests and others not shown here, SiclopDB adopts the strategy of seeking the set of available slave nodes with $\boldsymbol{T2R}$ enough to process the request that ensures the

Recommended SRT. Therefore, this type of request does not use monitoring nor adaptive partitioning during query execution.

Consider the following request R with Recommended SRT is 100 seconds and rows = 200.000 tuples.

```
SELECT column1, column2, column3 // ← R
FROM table1
   INNER JOIN table2
      ON table1.key_column = table2.key_column
   INNER JOIN table3
      ON table2.key_column = table3.key_column
   INNER JOIN table4
      ON table3.key_column = table4.key_column
WHERE table1.column = <<value1>>
   AND table2.column IN(<<value2>>, <<value3>>)
   AND table3.column IN(<<value4>>, <<value5>>, <<value6>>)
   AND SUBSTRING(table2.Name,1,4) IN (<<value7>>, <<value8>>)
GROUP BY table1.column1, table2.column
ORDER BY table1.column2, table1.column3;
```

In the optimistic approach, the greedy strategy is adopted, in which only one slave node executes the request and it is expected that ensures the Recommended SRT. Now consider three slave nodes, SN_1, SN_2 and SN_3, with $\boldsymbol{T2R_{SN1}}$ = 4.000 tuples/sec, $\boldsymbol{T2R_{SN2}}$ = 2.000 tuples/sec, $\boldsymbol{T2R_{SN3}}$ = 1.000 tuples/sec, respectively. In this case, the algorithm using greedy strategy chooses SN_1 because it was the first and enough in such a way to execute R within the Recommended SRT.

In pessimistic approach, the algorithm strategy is to choose half the number of slave nodes available with the highest $\boldsymbol{T2R}$ to execute request R. Consider four available slave nodes, SN_1, SN_2, SN_3, SN_4, with $\boldsymbol{T2R_{SN1}}$ = 4.000 tuples/sec, $\boldsymbol{T2R_{SN2}}$ = 2.000 tuples/sec, $\boldsymbol{T2R_{SN3}}$ = 1.000 tuples/sec and $\boldsymbol{T2R_{SN4}}$ = 1.500 tuples/sec, respectively. Then, the algorithm replicates the request R for SN_1 and SN_2, in such a way that least one can ensure the Recommended SRT.

In the worst-case scenario, if there are no slave nodes that meets the Recommended SRT, the closest node to meet the Recommended SRT in terms of $\boldsymbol{T2R}$ is selected. Monitoring the slave node to process this type of request is made after its processing, when it is checked for violation or not of Recommended SRT and metadata updates its information.

Finally, the summary of DQS component algorithm is shown below. As presented, for each type of request is used a strategy of partitioning and execution. After its execution, the request result is presented to the customer and to the provider it is presented the request information, such as SRT

violation, elapsed time of request etc. The information of SRT violation is important for the provider to understand the reasons of the violation and to make decisions to reduce the problem.

```
DQS ALGORITHM (R, TR, ET): RETURN RESULT
---------------------------------------------------------------------
– ET; //Elapsed Time = RSRT - ET.
– R; //Request
– TR; //Type of Request
– METADATA; //Metadata Class
– SLAVE_NODE[0..i]; //Available Slave Nodes
---------------------------------------------------------------------
21. BEGIN
22.     SWITCH(TR)
23.     CASE 1: //Type 1 Request
24.             IF (R.hasPredicate("WHERE T.pk = <<value>>;")) //Exception
25.                 DQS(R,3,ET);
26.             ELSE
27.                 Partition[0..i] = METADATA.getSelectedSlaveNode(R, SLAVE_NODE[0..i]);
28.                 FOR EACH Partition DO
29.                     RESULT += DQM(Partition, ET, 1, SLAVE_NODE[j]);
30.                 ENDFOR
31.                 RETURN RESULT;
32.             ENDIF
33.             BREAK;
34.      CASE 2: //Type 2 Request
35.             Partition[0..i] = PartitionEquiJoin(R);
36.             FOR EACH Partition DO
37.                 SubResult [0..i] = DQS (Partition,1,ET);
38.             ENDFOR
39.             RETURN JOIN(SubResult);
40.             BREAK;
41.      CASE 3: //Type 3 Request
42.             SelectedSlaveNodes[0..i]=METADATA.getSelectedSlaveNodes(R,SLAVE_NODE);//all
    nodes > ET
43.             IF (METADATA.getProbability(R) == OPTIMISTIC)
44.                 RESULT = DQM(R,ET,3,SelectedSlaveNodes[i]);
45.             ELSE //All slave nodes satisfy the ET
46.                 RESULT = DQM(R,ET,3,SelectedSlaveNodes[0..i/2]);
47.             ENDIF
48.             RETURN RESULT;
49.             BREAK;
50.      ENDSWITCH
51.      IF (ET > RSRT)
52.             METADATA.setViolation(TRUE);
53.      ENDIF
54. END
```

4.4.4 Dynamic query monitoring (DQM)

Given an optimized and scheduled query plan, the aim of this component is to monitor the query execution. As shown in Figure 4-3, the monitoring verifies, periodically, the possibility of a query to

be executed before a Recommended SRT. Therefore, the DQM component reevaluates each subquery at runtime and checks the possibility of SRT violation, whether it is low, the query continues its execution, otherwise, the query will be re-optimized in DQS component.

The monitoring will check the request execution progress. Whether the performance of slave node decreases, the system can try recovering and meeting the recommended SRT or if the performance of slave node increases, the system can use that to its advantage. Therefore, monitoring is adaptive with non-regular intervals, because the framework uses a strategy is based on following variables: CPU, memory and processing and reading percentage in DBMS of each slave node used by request. Thus, this work considers that slave nodes can have different performance.

The challenge of the monitoring algorithm is to define the best period to monitor, i.e., the time between consecutive probes. It should not be too small, since original queries would be partitioned into many subqueries. Thus, the overload added can prejudice more than help. Moreover, it should not be too large, because if that happens, it may be difficult to make corrections in a timely manner and avoid possible penalties.

DQM uses historical data of similar requests to establish the most efficient number of partitions for monitoring. Thus, the algorithm checks the request selectivity and the current performance of the first slave node in the initial provisioning. When there is no statistical data, by default, if the request selectivity is less than 10.000 tuples, the DQM will fragment the request within 2 partitions. If it is between 10.000 and 100.000 tuples, the DQM will fragment the request up to 4 partitions. If the selectivity is greater than 100.000 tuples, the DQM will fragment the request up to 8 partitions.

When there is statistical data, the number of partitions and the Recommended SRT used in the execution of similar requests is checked in metadata. Thus, the number of partitions for monitoring is chosen based on the similarity of the request (selectivity) and Recommended SRT. It is important to note that the operations will be realized in the metadata and will be available at the moment that is required by the request.

The summary of DQM component algorithm is shown below. As presented, type 1 and 3 requests use different strategies. For type 1 requests, the DQM uses monitoring and adaptive query processing and for type 3 requests, it does not use adaptive query processing, it uses a greedy algorithm in optimistic approach and the fastest execution in set of slave nodes in pessimistic approach.

```
DQM ALGORITHM (R, ET, TR, SLAVENODES): RETURN RESULT
---------------------------------------------------------------
– R; //Request.
– ET; //Elapsed Time: RSRT - ET
– TR; //Type of Request.
– SLAVENODES; //Slave Node to execute R.
---------------------------------------------------------------
1.  BEGIN
2.      SWITCH(TR)
3.      CASE 1: //Type 1 Request
4.              Partition[0..i] = Metadata.Partitioning(R);
5.              FOR EACH Partition DO
6.              IF((RESULT+=EXECUTE(Partition,SLAVENODES[0])).getElapsedTime()>T2R)
7.                  DQS (MERGE(Partition[j..i]),1, ET);
8.              ENDIF
9.              ENDFOR
10.             BREAK;
11.     CASE 3: //Type 3 Request
12.             //optimistic approach: SLAVENODES.getLength() returns 1.
13.             RESULT=EXECUTE(R,SLAVENODES[0..i]));
14.             BREAK;
15.     ENDSWITCH
16. RETURN RESULT;
17. END
```

4.5 Conclusion

This chapter presented solutions to efficient query processing of different types: select-range and select-aggregation queries (type 1 requests), select-equi-join queries (type 2 requests) and complex queries (type 3 requests). The strategies for adaptive processing were implemented and discussed in each component of SiclopDB framework.

It is important to note that all solutions (partitioning, monitoring and settings) are made in a non-intrusive way, i.e. slave nodes and their respective DBMSs do not require any changes to be used. Furthermore, these solutions are based on the costs of SLA violation and the computational cost model proposed in this work. To validate our solution, the next chapter presents the experimental results of these strategies with a large volume of data, machines and queries in the cloud.

Chapter 5 – Experimental evaluation: validation and results

5.1 Introduction

This chapter presents the experiments of a case study using the strategies of query processing presented in Chapter 4. To better understanding, this chapter is organized as follows:

> **5.2 Experimental rnvironment**: presents the environment where the experiments were executed.
>
> **5.3 Methodology**: presents the methodology of the experiments.
>
> **5.4 Requests used**: shows the requests used to the experiments.
>
> **5.5 Results and analysis**: presents the results obtained as well as its analysis.
>
> **5.6 Conclusion**: presents the final considerations of this chapter.

5.2 Experimental environment

The strategies presented in Chapter 4 were implemented in the SiclopDB framework using the Java language and concurrent programming with threads and an API based on OpenMP - Open Multi-Processing (Bull & Kambites, 2000). It was deployed in the Amazon EC2 cloud infrastructure using small instances (homogeneous environment). However, the performance of each VM (Virtual Machine) may vary over time. Due to the limitations of Amazon, 20 VMs were used, each with an Intel Xeon Processor with turbo up to 3.3GHz, 1.7 GB of main memory and 160 GB of disk storage.

It was created an AMI (Amazon Machine Image) of a VM with the database. This image allows to startup new VMs quickly. The Amazon EBS (Elastic Block Store) was used to store the AMI. Therefore, the startup time and instantiation of VM as well as the time of network authentication and database connection were not considered in experiments.

Each VM runs the Ubuntu 12.04 operating system and PostgreSQL 9.3 DBMS. This work focuses on OLAP applications with very large and complex database. Thus, a TPC-DS like benchmark was used to generate a database of approximately 13 GB, fully replicated in all VMs. Therefore, the database generated represents the customer data.

5.3 Methodology

The experiments aim at showing the efficiency of the query processing strategies proposed in this work. This way, it will check the ability to avoid penalties associated with SRT violation and the elasticity of the algorithm, according to the number of VMs allocated when processing queries.

Figure 5-1 shows step-by-step the methodology of the experiments. The first step consists in the definition and classification of the queries that will be used in the experiments. So, the queries are classified into one of three types of requests as defined in Chapter 3. Then, for each type of request, a set of queries workloads will be executed on SiclopDB framework. Finally, results will be analyzed checking the penalties, workloads statistics and metadata.

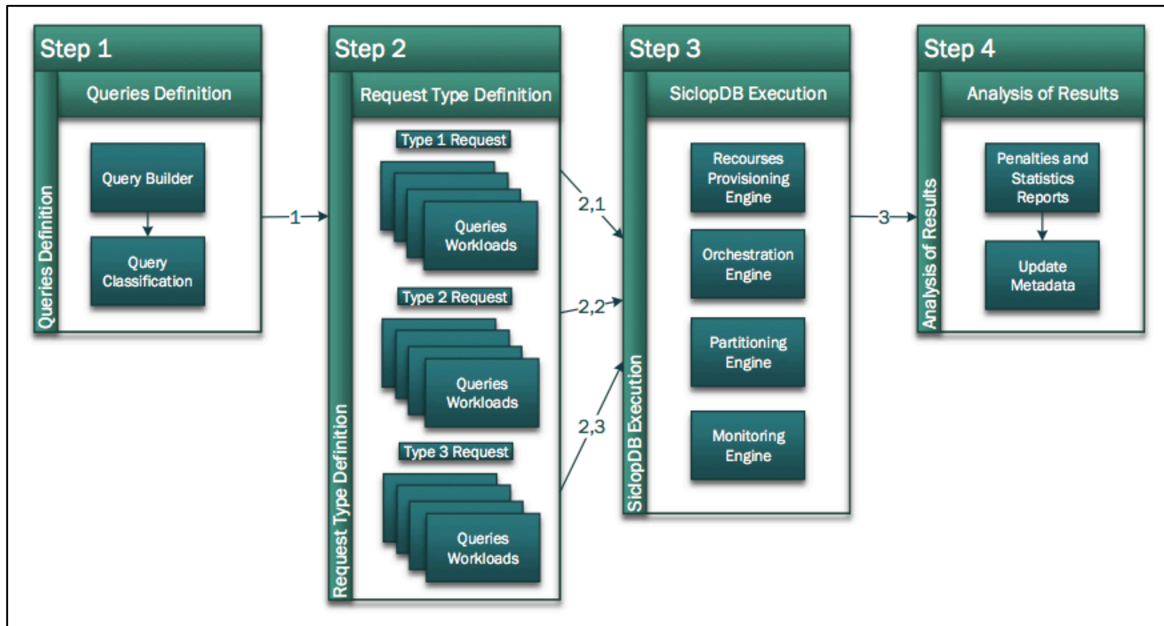


Figure 5-1. Methodology of experiments of SiclopDB framework.

For type 1 and type 2 requests, the experiments consisted in stressing the system using 10 workloads, each workload having 10 queries of the same type. For type 3 requests, as the strategy is predictive and queries are complex, 5 workloads were used, each workload having 5 queries of the same type. Finally, the experiments were performed using 10 workloads and each workload having 10 queries of several types of requests.

To know the minimum amount of required machines is a complex task. Therefore, previous tests were performed using a fixed number of VMs according to the strategy presented in Chapter 3. Thus, the minimum number of machines was found for the workload of the experiments. However, if new workloads arrive to the system, it will be necessary to perform extensive experiments again (as shown in Chapter 3) to obtain a new configuration of the service provider.

The arrival time of the queries workloads was disposed in two ways: (i) uniform distribution: each workload arriving at intervals of 30 seconds and (ii) uniformly varied distribution (non-uniform distribution): each workload arriving at random time intervals between 10 and 60 seconds. This

distribution is closer to real environments, since the unpredictability of workloads arriving to the system and performance variation are characteristics of cloud environments.

In this work, different values of Recommended SRT were used, from the most restricted to the most relaxed. After definition and classification of the queries to be used in the experiments, the values of Recommended SRT were obtained from tests following the methodology presented in Chapter 3. In order to find more restricted as well as more relaxed values of Recommended SRT, it was adapted according to SRT value. Therefore, minus 20% of SRT value corresponds the most restricted value of Recommended SRT and more 20% of SRT value corresponds the most relaxed value of Recommended SRT. For example, if the Recommended SRT is 100 seconds, experiments were performed also considering Recommended SRT of 80 seconds, and finally considering Recommended SRT of 120 seconds.

Seeking for more accurate results for each type of request, experiments were repeated 10 times. Finally, to eliminate any possible interference between successive experiments, in particular, effects of other queries already executed, the OS cache was deleted and the DBMS has been restarted before executing the queries workloads again.

For each experiment, the number of virtual machines used are observed in accordance with time. To calculate the computational cost, it is enough to observe the number of virtual machines used by each query. Finally, the query runtime is measured according to the strategies described in Chapter 4.

5.4 Used requests

This section presents some requests used in the case study. Type 1 requests are select-range and/or select-aggregation requests. They have approximately 300,000 tuples of selectivity and uses the *catalog_sales* table of TPC-DS. For the select-range queries whose predicate is on non-key attribute, it was used the *cs_bill_hdemo_sk* attribute and to queries whose predicate is on the key attribute, it was used *cs_item_sk* attribute of the same table.

The experiments involving the select-aggregation queries use the same select-range queries. However, the SELECT clause was modified. Finally, values interval for *cs_item_sk* and *cs_bill_hdemo_sk* attributes of each query was generated randomly from the values stored in the database. Following are examples of type 1 requests used in the experiments. All type 1 requests used in the experiments are listed in Annex A1.

```
select * from catalog_sales where cs_item_sk >= 1 and cs_item_sk < 300000;
select * from catalog_sales where cs_bill_hdemo_sk > 150000;
select * from catalog_sales where cs_item_sk > 600000;
select SUM(cs_bill_hdemo_sk) from catalog_sales;
select COUNT(cs_bill_hdemo_sk) from catalog_sales;
```

Type 2 requests are select-joins requests. The selectivity of these requests varied between 1,000 and 30,000 tuples and it uses different tables of TPC-DS benchmark. Besides, equi-joins predicates are represented according to the number of tables in the FROM clause. Following some examples of type 2 requests used in the experiments. The mostly used type 2 requests in the experiments are listed in Annex A2.

```
select  *
from store_sales,household_demographics,time_dim, store
where ss_sold_time_sk = time_dim.t_time_sk
        and ss_hdemo_sk = household_demographics.hd_demo_sk
        and ss_store_sk = s_store_sk;

select  ss_item_sk
from store_sales, time_dim
where ss_sold_time_sk = t_time_sk;

select  store_sales.*
from store_sales, customer_demographics, date_dim, item, promotion
where ss_sold_date_sk = d_date_sk and
         ss_item_sk = i_item_sk and
         ss_cdemo_sk = cd_demo_sk and
         ss_promo_sk = p_promo_sk and
         cd_gender = 'M';
```

Type 3 requests are select-sets-grouping-nesting requests and, optional select-aggregation and select-joins. They present very complex queries plans and its selectivity is between 150,000 and 200,000 tuples. It uses at least 10 different tables of TPC-DS. Following some examples of type 3 requests used in the experiments. The most type 3 requests used in the experiments are listed in Annex A3.

```
select  i_brand_id brand_id, i_brand brand, i_manufact_id, i_manufact, sum(ss_ext_sales_price)
ext_price
from date_dim, store_sales, item,customer,customer_address,store
where d_date_sk = ss_sold_date_sk   and ss_item_sk = i_item_sk
         and i_manager_id=13   and d_moy=11
         and d_year=2001   and ss_customer_sk = c_customer_sk
         and c_current_addr_sk = ca_address_sk   and substr(ca_zip,1,5) <> substr(s_zip,1,5)
         and ss_store_sk = s_store_sk
 group by i_brand,i_brand_id,i_manufact_id,i_manufact
 order by ext_price desc,i_brand,i_brand_id,i_manufact_id,i_manufact;
```

```
select  i_item_id,
        avg(ss_quantity) agg1,
        avg(ss_list_price) agg2,
        avg(ss_coupon_amt) agg3,
        avg(ss_sales_price) agg4
 from store_sales, customer_demographics, date_dim, item, promotion
 where ss_sold_date_sk = d_date_sk and
       ss_item_sk = i_item_sk and
       ss_cdemo_sk = cd_demo_sk and
       ss_promo_sk = p_promo_sk and
       cd_gender = 'M' and
       cd_marital_status = 'M' and
       cd_education_status = '4 yr Degree' and
       (p_channel_email = 'N' or p_channel_event = 'N') and
       d_year = 2001
 group by i_item_id
 order by i_item_id


with wss as
 (select d_week_seq,
        ss_store_sk,
        sum(case when (d_day_name='Sunday') then ss_sales_price else null end) sun_sales,
        sum(case when (d_day_name='Monday') then ss_sales_price else null end) mon_sales,
        sum(case when (d_day_name='Tuesday') then ss_sales_price else  null end) tue_sales,
        sum(case when (d_day_name='Wednesday') then ss_sales_price else null end) wed_sales,
        sum(case when (d_day_name='Thursday') then ss_sales_price else null end) thu_sales,
        sum(case when (d_day_name='Friday') then ss_sales_price else null end) fri_sales,
        sum(case when (d_day_name='Saturday') then ss_sales_price else null end) sat_sales
 from store_sales,date_dim
 where d_date_sk = ss_sold_date_sk
 group by d_week_seq,ss_store_sk
 )
  select  s_store_name1,s_store_id1,d_week_seq1
       ,sun_sales1/sun_sales2,mon_sales1/mon_sales2
       ,tue_sales1/tue_sales2,wed_sales1/wed_sales2,thu_sales1/thu_sales2
       ,fri_sales1/fri_sales2,sat_sales1/sat_sales2
 from
 (select s_store_name s_store_name1,wss.d_week_seq d_week_seq1
        ,s_store_id s_store_id1,sun_sales sun_sales1
        ,mon_sales mon_sales1,tue_sales tue_sales1
        ,wed_sales wed_sales1,thu_sales thu_sales1
        ,fri_sales fri_sales1,sat_sales sat_sales1
  from wss,store,date_dim d
  where d.d_week_seq = wss.d_week_seq and
        ss_store_sk = s_store_sk and
        d_month_seq between 1200 and 1200 + 11) y,
 (select s_store_name s_store_name2,wss.d_week_seq d_week_seq2
```

```
        ,s_store_id s_store_id2,sun_sales sun_sales2
        ,mon_sales mon_sales2,tue_sales tue_sales2
        ,wed_sales wed_sales2,thu_sales thu_sales2
        ,fri_sales fri_sales2,sat_sales sat_sales2
  from wss,store,date_dim d
  where d.d_week_seq = wss.d_week_seq and
        ss_store_sk = s_store_sk and
        d_month_seq between 1200+ 12 and 1200 + 23) x
 where s_store_id1=s_store_id2
   and d_week_seq1=d_week_seq2-52
 order by s_store_name1,s_store_id1,d_week_seq1;
```

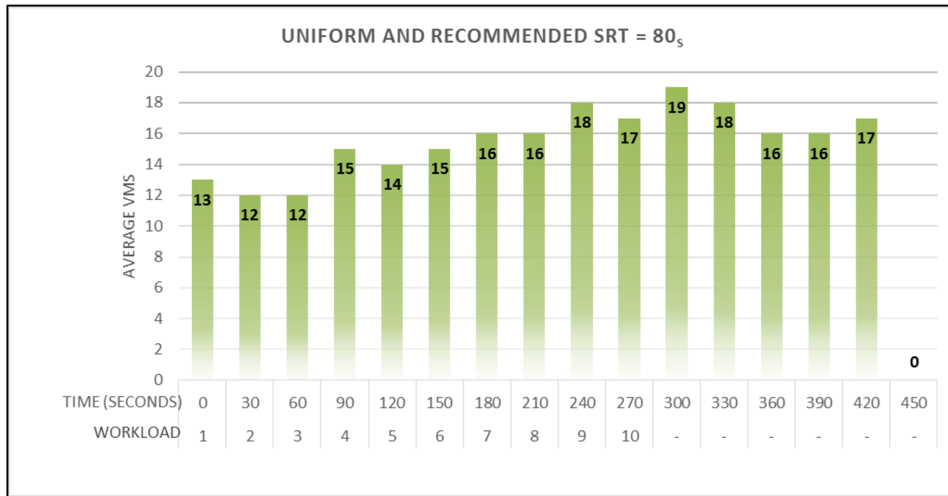
5.5 Results and analysis

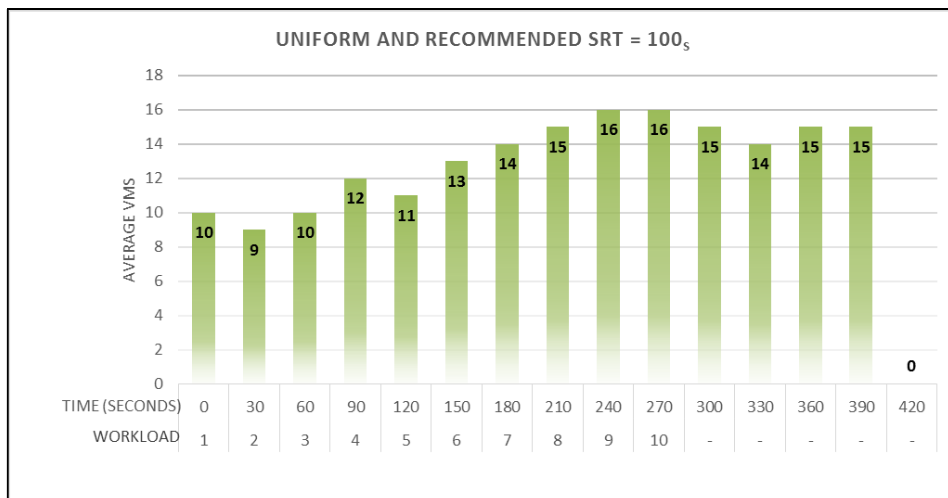
To execute the experiments, a VM was chosen arbitrarily to be the master node and consequently the other nodes become slave nodes. SiclopDB framework was deployed in the master node and the others were deployed with the TPC-DS benchmark database. Following are presented the results of experiments for each type of request.

5.5.1 Type 1 requests

For type 1 requests, experiments were divided into select-range queries and select-aggregation queries. Figures 5.2 and 5.3 show, respectively, experiments of select-range queries with the arrival of workloads following the uniform distribution and with the arrival of workloads following the non-uniform distribution. The graphs present the number of VMs used by time in seconds and the Recommended SRTs used were 80, 100 and 120 seconds. These values were obtained according to experiments realized following the methodology in Chapter 3. Finally, the queries predicate may be on a non-key attribute or on a key attribute.

It is important to emphasize at this point that when the attribute is not a primary key, our strategy scans all tuples of the table, i.e. all tuples are checked to verify whether they satisfy the predicate. Thus, this type of queries requires more processing time than the select-range that have a predicate on key attribute. Consequently, this causes the increase of VM computational cost, since the response time is higher.





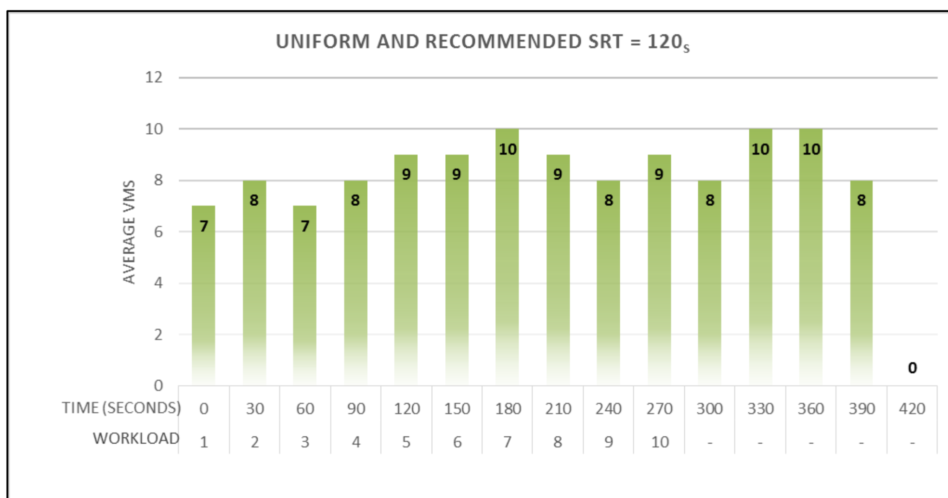


Figure 5-2. Type 1 Requests (Select-Range): average virtual machines used for workloads uniformly arriving every 30 seconds for the Recommended SRTs: 80, 100 and 120 seconds.

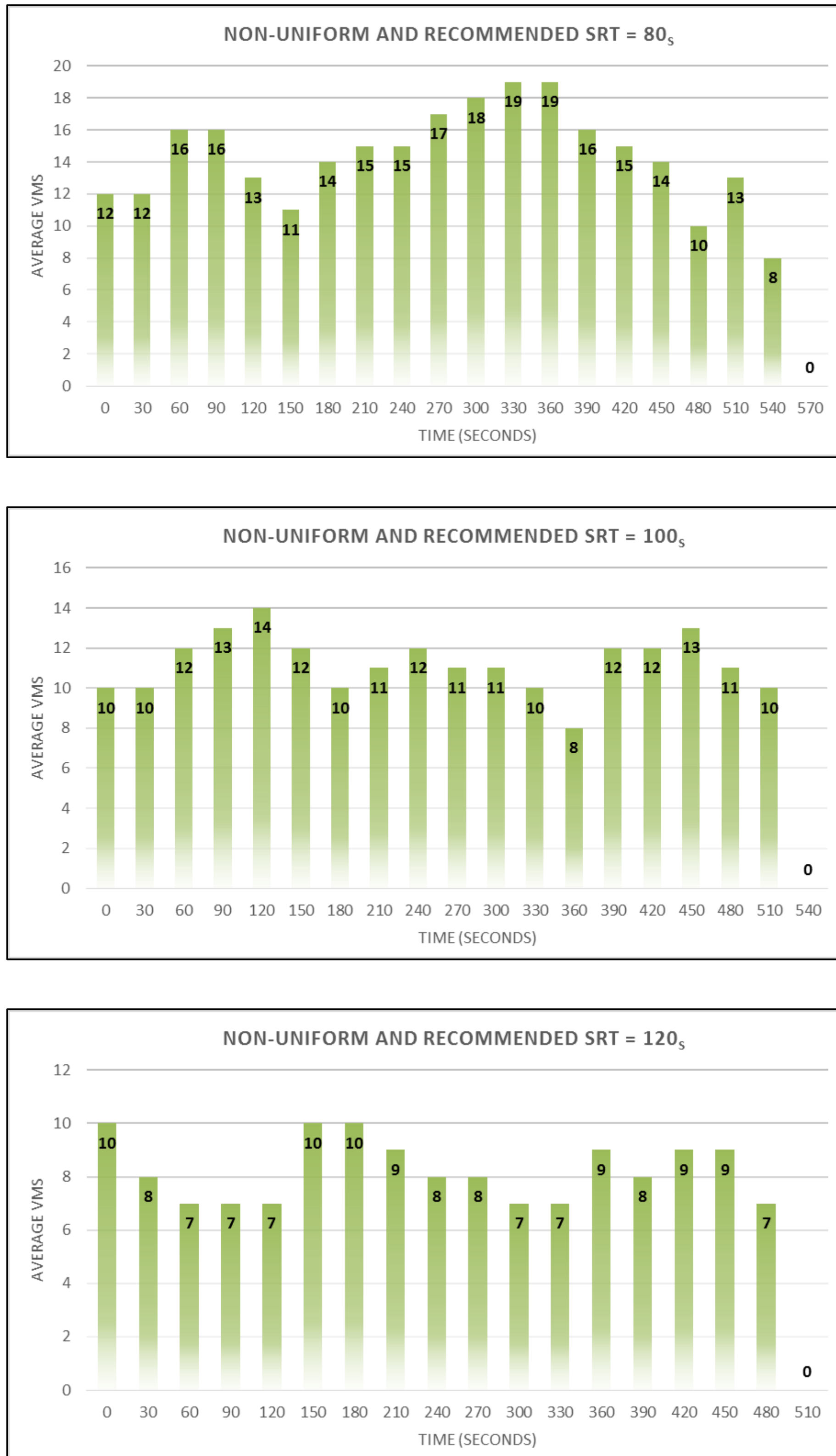
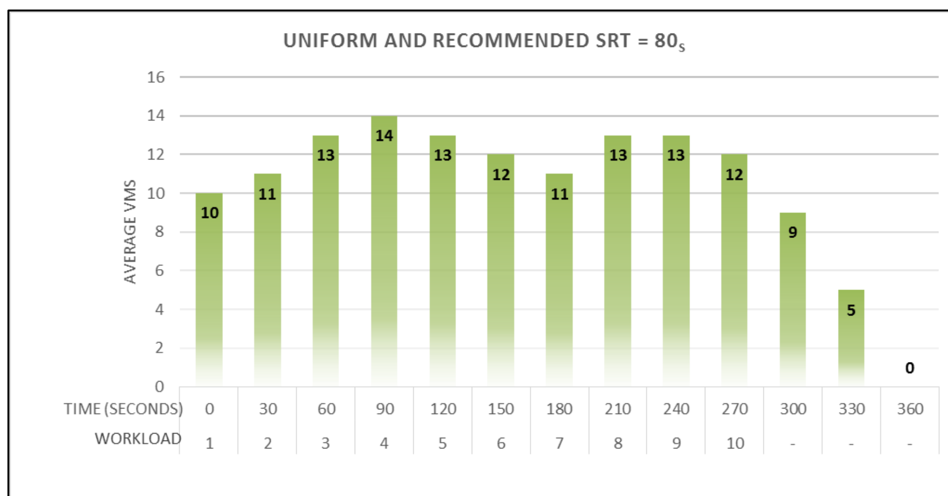


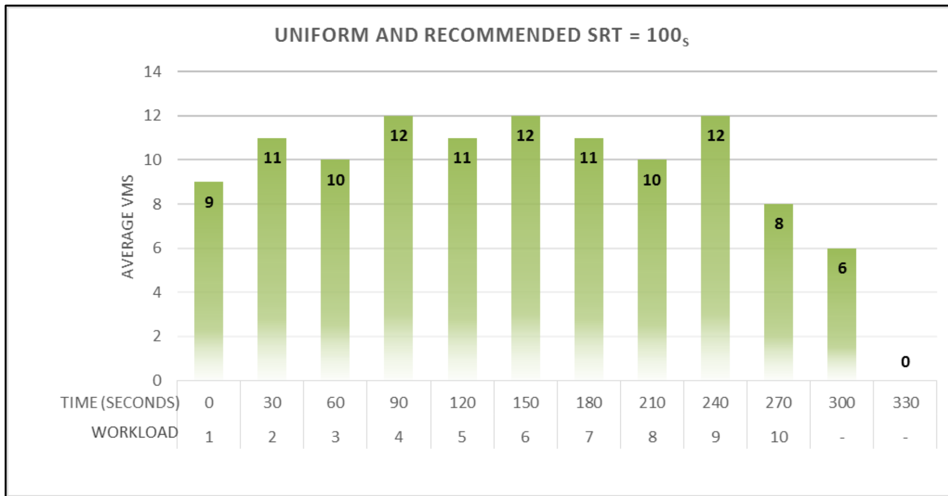
Figure 5-3. Type 1 Requests (Select-Range): average virtual machines used for workloads randomly arriving between 10 and 60 seconds for the Recommended SRTs: 80, 100 and 120 seconds.

According to the results, it can be seen the increase and decrease of the workloads on the system and the elasticity on the number of virtual machines allocated to execute the queries. We can also see that in Figure 5.2 and 5.3 the SiclopDB used almost all VMs available in the most restricted Recommended SRT, whose limit was almost reached in ninth and tenth workload. If the limit was reached, to reduce the provider penalties, this problem could be solved recommending the customer to acquire more VMs and/or to making new experiments to update the Recommended SRT.

When the SRT is more restricted, the computational cost is higher or equal to the computational cost of the most relaxed SRT, this happens to avoid penalties. Moreover, the computational cost is higher when the workloads arrive at random times (non-uniform distribution) if compared to uniform distribution. We believe that the system may not recover quickly when there is an unexpected overload of the resources, and seeking quick reaction to execute the queries, the algorithm allocates more VMs to execute the workload in SRT time. Consequently, the computational cost increases.

The results of experiments utilizing only select-aggregation queries are shown in Figures 5.4 and 5.5. Figure 5.4 presents experiments with uniform distribution of workloads and Figure 5.5 shows the experiments with non-uniform distribution. As for the select-range queries, the Recommended SRT was varied, the most restricted SRT was 80 seconds and the most relaxed SRT was 120 seconds.





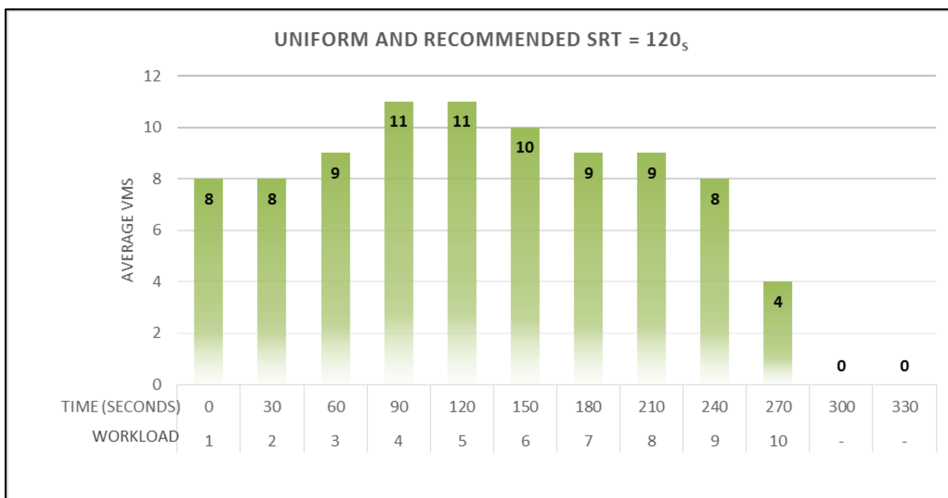
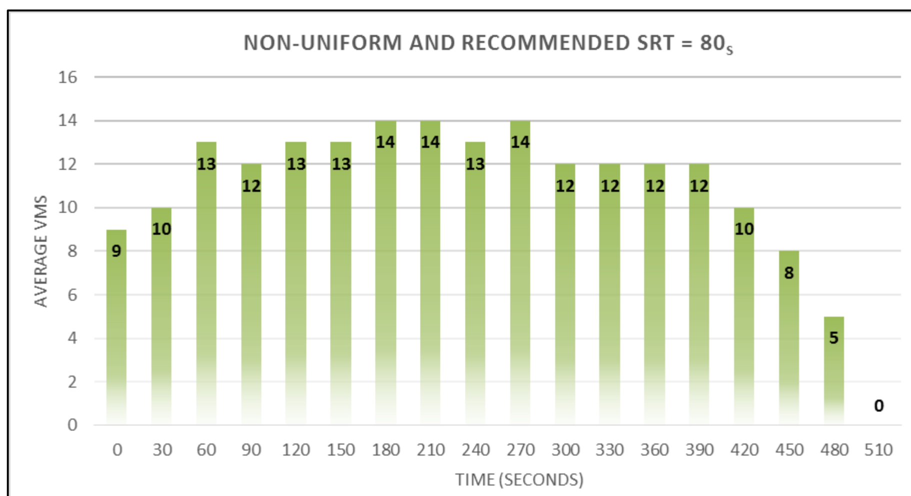


Figure 5-4. Type 1 Requests (Select-Aggregation): average virtual machines used for workloads uniformly arriving every 30 seconds for the Recommended SRTs: 80, 100 and 120 seconds.



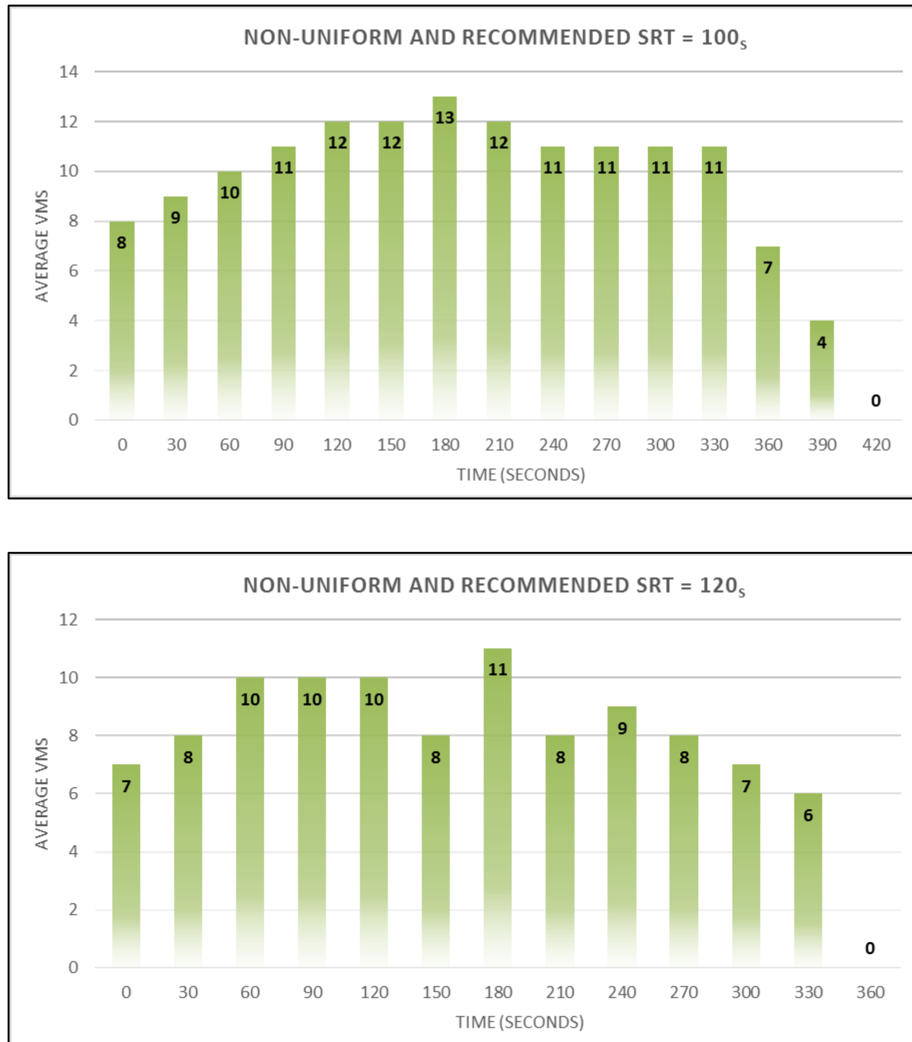


Figure 5-5. Type 1 Requests (Select-Aggregation): average virtual machines used for workloads randomly arriving between 10 and 60 seconds for the Recommended SRTs: 80, 100 and 120 seconds.

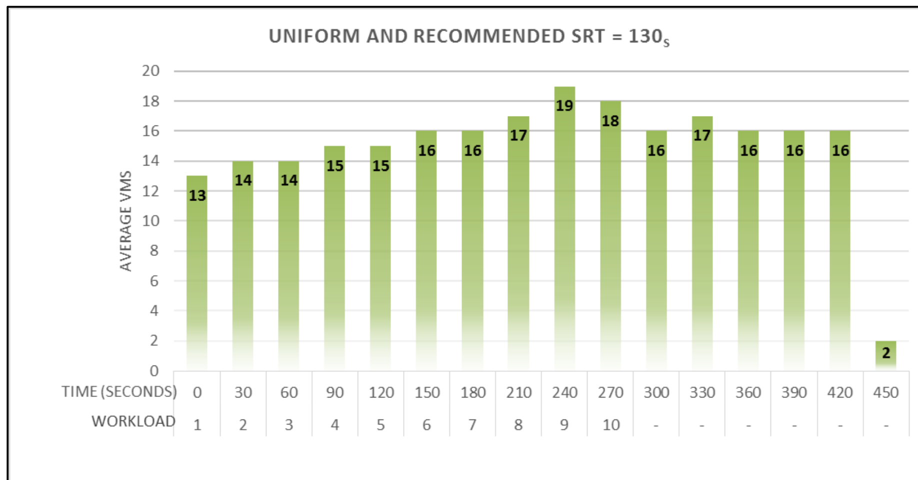
We can see that results show a similar behavior with the select-range queries. This is due to the strategy used to perform this type of query, which is similar to select-range queries. Moreover, for all experiments, the workloads are of the same type. However, several factors that are out of control, for example, other processes running on the physical machine hosting the VM can contribute to performance variation.

Comparing both experiments (select-range and select-aggregation queries) it can be noticed that select-aggregation queries present faster response time than the select-range queries. This is due to the amount of data to be retrieved by the SELECT clause. The select-range queries used in the experiments recover all attributes of a table (SELECT *), i.e. a large volume of data and according to (Elmasri & Navathe, 2010) more I/O operations are performed. Finally, select-aggregation queries need to retrieve only the aggregated value of an attribute.

5.5.2 Type 2 requests

For type 2 requests, experiments were realized in queries with or without predicate as shown in section 5.4. As shown in the previous subsection, the following graphs show the number of VMs used by the time in seconds. According to the proposed strategy in this work, it has obtained similar results to type 1 requests, because each fragment of a query is executed using the same strategy. Therefore, the primary difference between these types (type 1 and type 2 requests) is the query partitioning and merge of their results.

Figures 5.6 and 5.7 show, respectively, the experiments with the arrival of workloads following the uniform distribution and with the arrival of workloads following the non-uniform distribution. As in previous experiments, the Recommended SRT was varied, the most restricted SRT was 130 seconds and the most relaxed SRT was 180 seconds



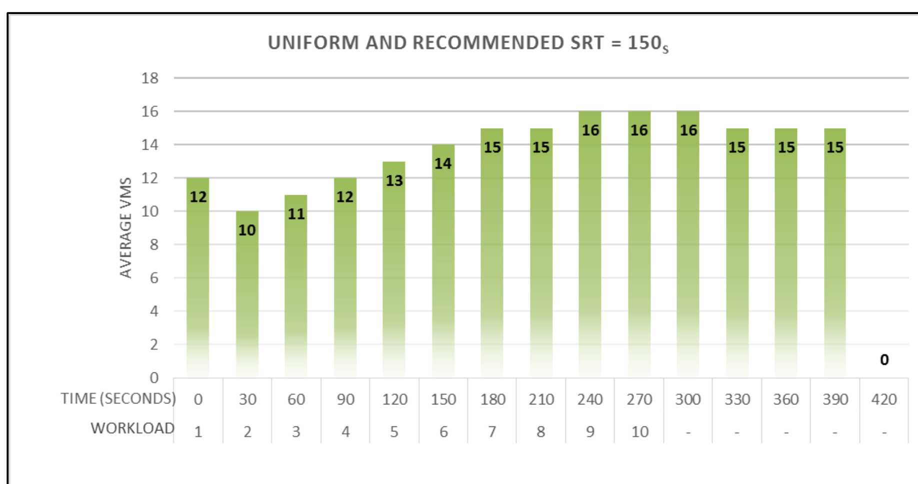
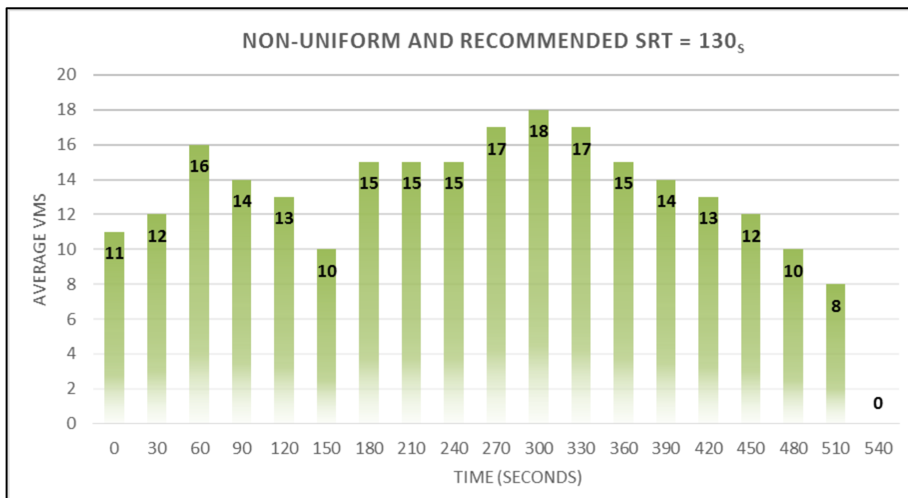
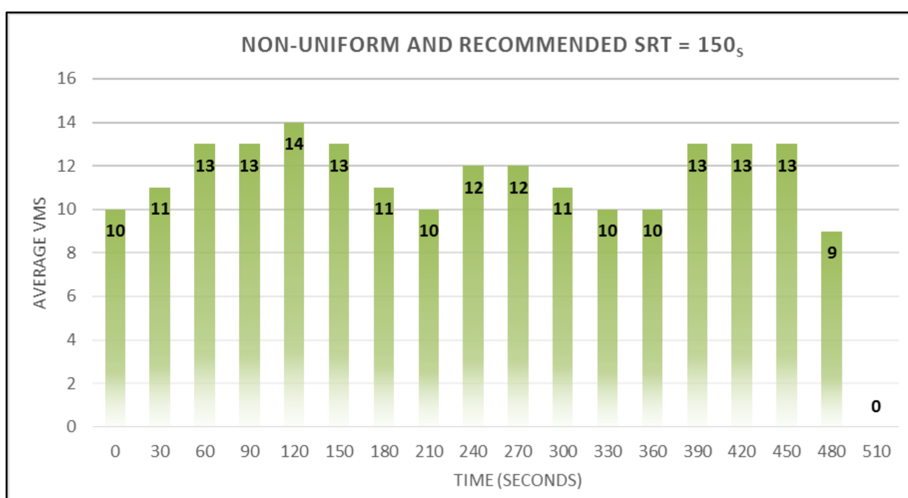


Figure 5-6. Type 2 Requests: average virtual machines used with workloads uniformly arriving every 30 seconds for the Recommended SRTs: 130, 150 and 180 seconds.





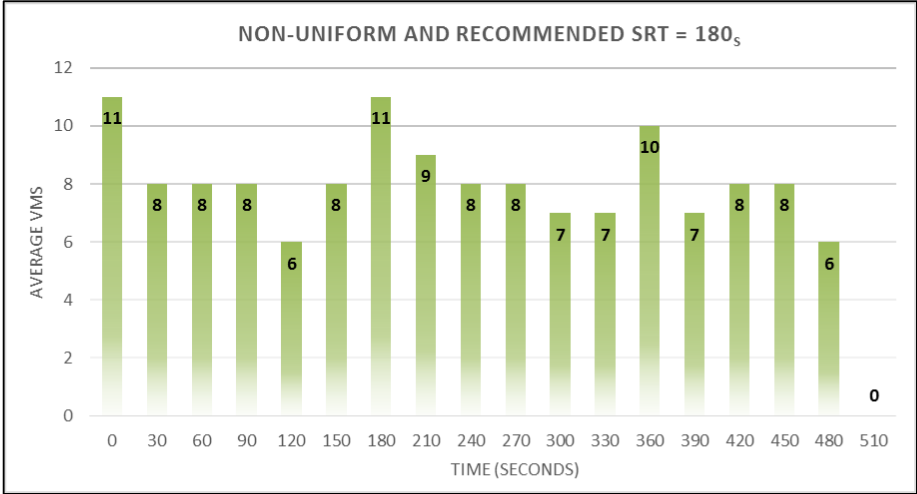


Figure 5-7. Type 2 Requests: average virtual machines used with workloads randomly arriving between 10 and 60 seconds for the Recommended SRTs: 130, 150 and 180 seconds.

In this work, the partitioning and merge of a query occurs in the VM Leader, which is a VM responsible for partitioning the query, processing joins and applying merge of their results. The VM Leader was chosen automatically and corresponds to the VM that receives the first fragment of a query.

Each fragment of a query receives a FID (Fragment Identifier) that is used to control and correctly merge the results. The partitioning time of queries was not considered because the reduced complexity of the queries used in the experiments. However, it was observed that the merging of the results causes a higher time to execute queries, approximately 10% more than the select-range requests. Finally, it is important to observe that in the most restricted SRT, the ninth workload reached the limit of the infrastructure service provider.

5.5.3 Type 3 requests

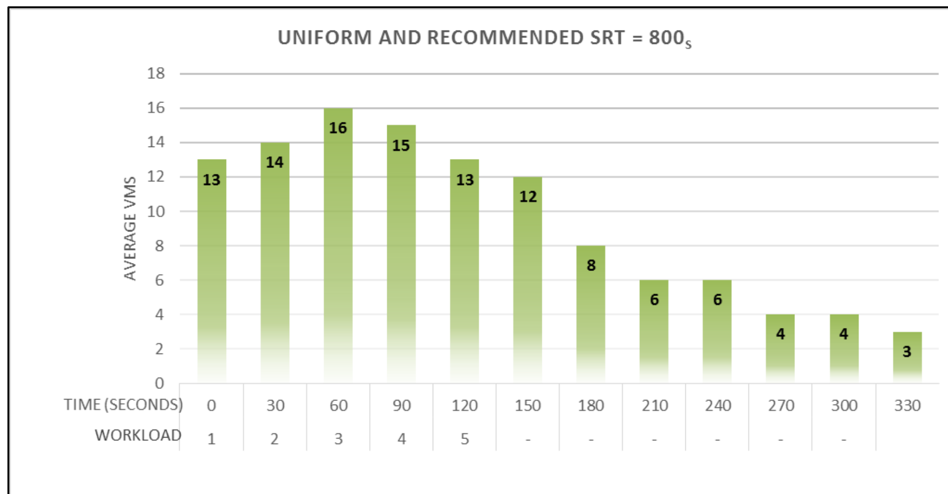
For type 3 requests, the experiments were realized with complex queries obtained from the TPC-DS Benchmark. As shown in previous sessions, the following graphs show the number of VMs allocated by time in seconds and the Recommended SRT was varied, the most restricted SRT was 800 seconds and the most relaxed SRT was 1200 seconds. However, due to the complexity of these queries and the limit of VMs available in the service provider, only 5 workloads were used, each having 5 complex queries.

According to proposed strategy of this work, the experiments stressed the system searching a VM that could execute successfully a query in SRT time (optimistic approach) or executing a query over

a set of VMs that one VM could execute successfully the query in SRT time (pessimistic approach). Therefore, it is not used monitoring nor adaptive partitioning during query execution.

Figures 5.8 and 5.9 show, respectively, the results of experiments following the uniform distribution and non-uniform distribution of workloads. We can observe that due to the strategy used in this work a large number of VMs are used since the first query workload. In addition, in accordance to the strategy presented in Chapter 4, the algorithm chooses through the metadata the optimistic or pessimistic strategy for executing a query and after its execution the metadata are updated.

However, we believe that the decrease in the use of virtual machines after the third workload happened due to the algorithm starting to use more often the optimistic approach. Consequently, the queries were being executed successfully. Moreover, it can be observed that due to the complexity and selectivity of the queries, there is a greater overhead for the ending its results.



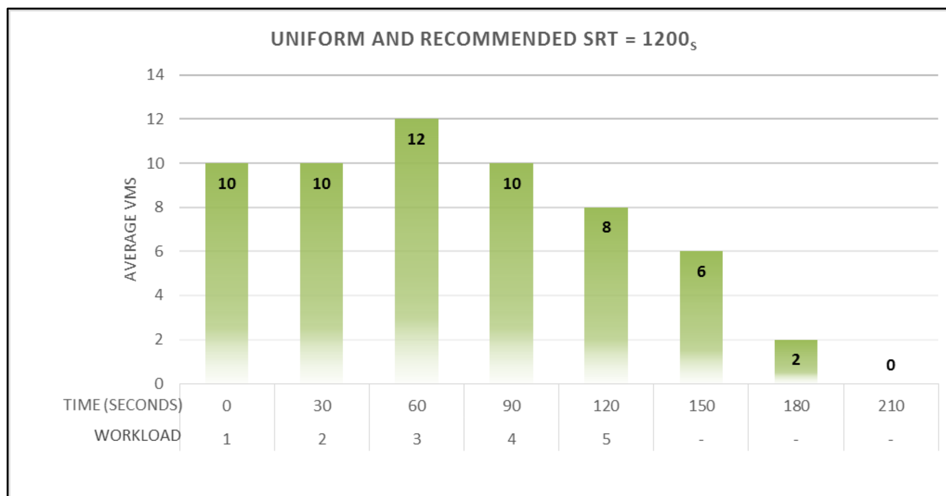
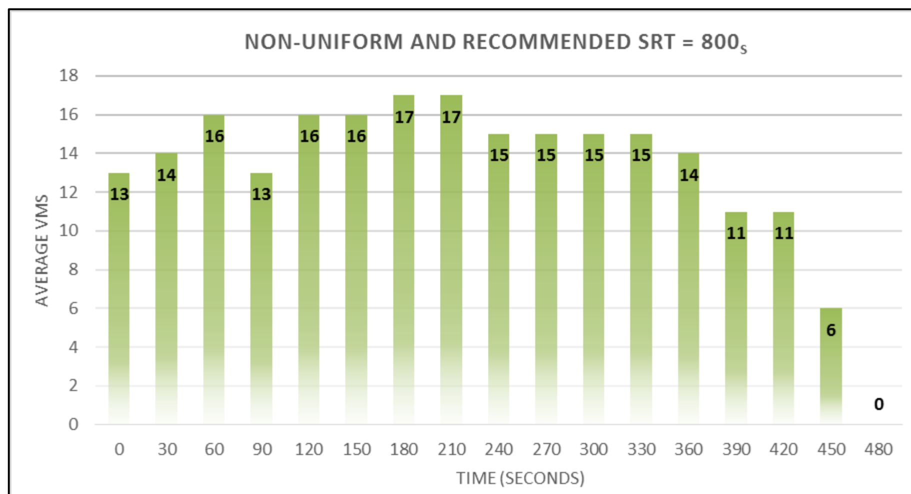


Figure 5-8. Type 3 Requests: average virtual machines used with workloads uniformly arriving every 30 seconds for the Recommended SRTs: 800, 1000 and 1200 seconds.



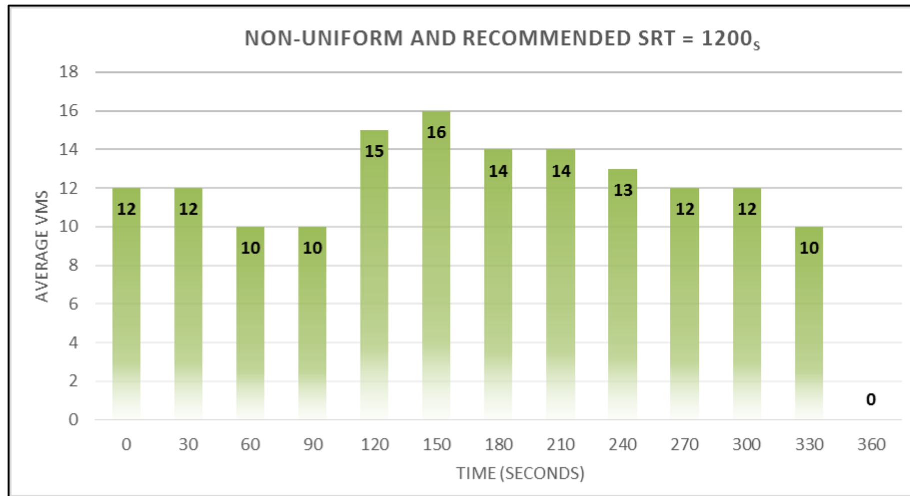
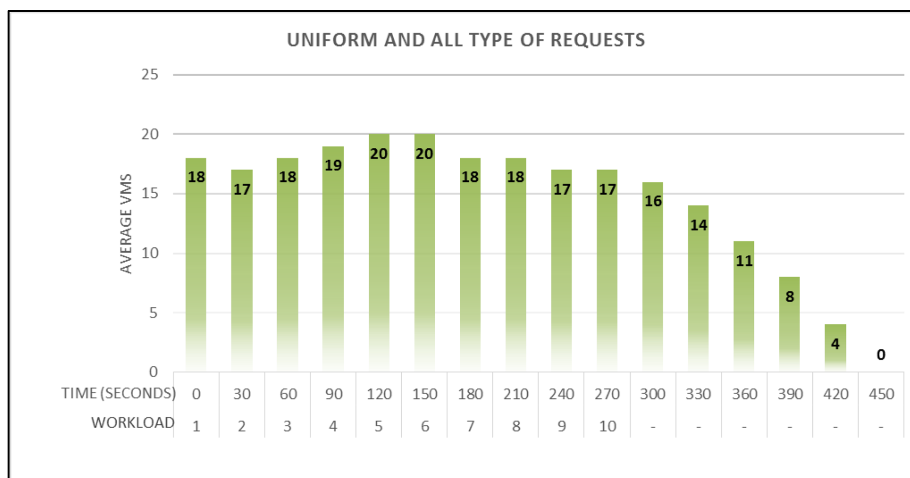


Figure 5-9. Type 3 Requests: average virtual machines used with workloads randomly arriving between 10 and 60 seconds for the Recommended SRTs: 800, 100 and 1200 seconds.

5.5.4 All type of requests

Finally, experiments were realized using all requests types over the same queries workload. According to the strategies proposed in this thesis, similar results to the previous ones were obtained. The main overhead is the algorithm having to classify each query to be executed (type 1, 2 or 3 of request). After classifying the query, the query is executed according to the already mentioned strategies.

Figure 5.10 shows, respectively, the experiments with the arrival of workloads following uniform distribution and non-uniform distribution. The graphs show the number of VMs used by the time in seconds. However, unlike previous experiments, each query after classification has a different Recommended SRT, according to their type of request.



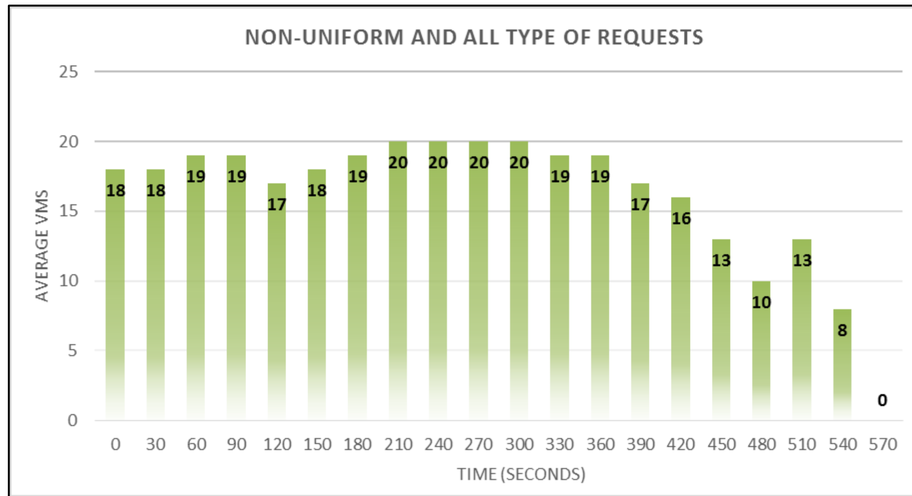


Figure 5-10. All Type Requests: average virtual machines used with workloads uniformly arriving every 30 seconds and randomly arriving between 10 and 60 seconds.

For several moments, it can be seen that the limit of the provider's infrastructure is reached; however, it has not been exceeded. Thus, it can be seen the increase and decrease in workloads due to elasticity in the number of allocated virtual machines to execute all queries. It is important to observe that no penalty occurred with all queries. However, if new workloads arrive to the system, it will be necessary to perform extensive experiments again (as shown in Chapter 3) to obtain a new configuration of service provider.

5.6 Conclusion

This chapter presented the experiments of the proposed strategies in Chapter 4. Given the increase and decrease of the workloads, it can be seen the elasticity in the number of virtual machines allocated by the methods proposed in this thesis to execute queries.

Furthermore, results show that the solution reacts to the resources variation of the environment and to different sizes of workloads. The solution ensures that the Recommended SRT is satisfied in a non-intrusive and automatic way. Finally, our proposal was effective to avoid the penalties in the execution of queries and the Recommended SRT was satisfied in all experiments without incurring penalties.

Chapter 6 – Conclusion

6.1 Final considerations

This chapter presents the main conclusions of this thesis. They are closely related to the objectives, contributions and solutions that have been presented along this work. Part of this thesis has been evaluated by the community in (Costa et al., 2015, 2016; Costa & Sousa, 2013).

This work presented a new solution to efficient query processing on large databases available in a cloud environment. It integrated adaptive re-optimization at runtime of the query and their costs are based on the SRT (Service Response Time) QoS parameter. For this, it was firstly proposed a model that allows the cloud service provider and its customers to establish an appropriate SLA relative to SRT performance of their applications available in the cloud. After, it was presented a new partitioning and monitoring strategies for adaptive processing of different types of queries (database access requests). Moreover, a dynamic provisioning strategy and its algorithm were presented. Finally, to validate this work, the strategies were implemented in the SiclopDB framework and the experiments were evaluated in Amazon EC2 cloud infrastructure.

Chapter 2 presented researches, concepts and technologies related to the object of study of this doctoral thesis (Objective 1 of the thesis). From the user's point of view, the SRT parameter is considered one of the main QoS performance parameters. However, major cloud providers have ignored or inappropriately treated the SRT parameter in SLA due to its complexity.

Furthermore, we can observe that most works in the literature focus on shorter query execution time and in the prediction of resources that will used by a query through the system current context. These works may not be suitable in unpredictable environments related to the availability of resources. Other related works focus on adaptive query processing. However, they present limitations of elasticity and/or scalability in their algorithms, the absence of adaptive monitoring query processing and/or use of intrusive solutions.

Chapter 3 presented one of the main contributions of this thesis (Objective 2 of the thesis). We propose a model that allows the cloud service provider and its customers to establish an appropriate SLA relative to SRT performance of their applications running in the cloud. The proposed model is a non-intrusive solution and can be applied when companies plan to migrate their applications, OLAP or not, to cloud services providers, with the goal to allocate computational resources on demand, to ensure the quality of service in terms of SRT.

Finally, the proposed model was evaluated in the Amazon EC2 cloud infrastructure using small instances and a TPC-DS (*Tpc Benchmark™ Ds*, 2012) like benchmark. It was used for generating an

OLAP database, considering that some cloud computing platforms support SQL queries directly or indirectly, this makes the proposed solution suitable for these kind of problems.

Chapter 4 presented the main contribution of this work (Objective 3 of the thesis), a new solution to efficient query processing on large databases available in a cloud environment. It is restricted to relational database requests. This way, it presented solutions to efficient processing of different queries: select-range and select-aggregation queries (type 1 requests), select-equi-join queries (type 2 requests) and complex queries (type 3 requests). Finally, the partitioning and monitoring strategies and a dynamic provisioning strategy were discussed and implemented for each component of the SiclopDB framework.

It is important to emphasize that this work focuses on OLAP applications because in this kind of environment, adaptive processing produces positive effects at query runtime. Furthermore, it is important to note that all solutions (partitioning, monitoring and settings) are made in a non-intrusive way, i.e. slave nodes and their respective DBMSs do not require any changes to be used. Finally, these solutions were based on the costs of SLA violation and the computational cost model proposed in this work.

To validate our solution, chapter 5 presented the experiments of the proposed strategies of Chapter 4. For this, the SiclopDB framework was evaluated in Amazon EC2 cloud infrastructure using small instances and the TPC-DS like benchmark was used for generating the OLAP database.

Given the increase and decrease of the workloads, it could be seen the elasticity in the number of virtual machines allocated to execute queries. Furthermore, results shown that the solution reacts to the variation of the environment with different of workloads. Finally, our proposal was effective to avoid the penalties in the execution of queries and the Recommended SRT was satisfied in all experiments without incurring penalties.

6.2 Future work

As future work, we will deploy our proposed model, beyond Amazon, in an Azure and Google Cloud Platform, using similar VMs. After, we will compare the response time between the different public cloud providers. Moreover, other future work consists to use specialized systems for the automatic classification of applications according to the request types as well as to the automatic analysis of results. Other work comprises to replace DOS and COS tools by others benchmark tools, for example, *pgbench* tool that allows a greater variation of performance parameters.

Furthermore, we intend to improve adaptive strategies to incorporate/allow different query types. For example, allowing SQL predicates more complex and reduction network data traffic. Consequently, we expect to carry out more experiments and modify the presented cost models. Finally, we intend to improve the cost model involving other SLA parameters, such as resiliency, throughput and efficiency, since they are important measures to evaluate the performance of services in cloud infrastructures.

References

Abadi, D. J. (2010). DataManagement in the Cloud: Limitations and Opportunities. *Bulletin of the IEEE Computer Society Technical Commitee on Data Engineering*, *132*, 1–10. http://doi.org/10.1007/978-1-4419-0176-7

Alrifai, M., & Risse, T. (2009). Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition. In *Proceedings of the 18th International Conference on World Wide Web* (pp. 881–890). New York, NY, USA: ACM. http://doi.org/10.1145/1526709.1526828

Alves, D., Bizarro, P., & Marques, P. (2011). Deadline Queries: Leveraging the Cloud to Produce On-Time Results. In *2011 IEEE 4th International Conference on Cloud Computing* (pp. 171–178). IEEE. http://doi.org/10.1109/CLOUD.2011.12

Amazon Web Services. (2015). *Auto Scaling: Developer Guide*. Retrieved from http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/as-dg.pdf

Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., ... Xu, M. (2005). *Web Services Agreement Specification (WS-Agreement)*. Retrieved from http://www.ggf.org/Public_Comment_Docs/Documents/Oct-2005/WS-AgreementSpecificationDraft050920.pdf

AWS EC2 Service Level Agreement. (2015). Retrieved June 15, 2015, from http://aws.amazon.com/ec2-sla

AWS S3 Service Level Agreement. (2015). Retrieved June 15, 2015, from http://aws.amazon.com/s3-sla

Bruno, N., Chaudhuri, S., & Ramamurthy, R. (2009). Power Hints for Query Optimization. In *2009 IEEE 25th International Conference on Data Engineering* (pp. 469–480). Shanghai, China: IEEE. http://doi.org/10.1109/ICDE.2009.68

Bull, J. M., & Kambites, M. E. (2000). JOMP–an OpenMP-like Interface for Java. In *Proceedings of the ACM 2000 Conference on Java Grande* (pp. 44–53). New York, New York, USA: ACM Press. http://doi.org/10.1145/337449.337466

Canfora, G., di Penta, M., Esposito, R., & Villani, M. L. (2005). An approach for QoS-aware service composition based on genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation - GECCO '05* (p. 1069). New York, New York, USA: ACM Press. http://doi.org/10.1145/1068009.1068189

Cervino, J., Kalyvianaki, E., Salvachua, J., & Pietzuch, P. (2012). Adaptive Provisioning of Stream Processing Systems in the Cloud. In *2012 IEEE 28th International Conference on Data Engineering Workshops* (pp. 295–301). IEEE. http://doi.org/10.1109/ICDEW.2012.40

Chi, Y., Moon, H. J., Hacigümüş, H., & Tatemura, J. (2011). SLA-tree. In *Proceedings of the 14th International Conference on Extending Database Technology - EDBT/ICDT '11* (p. 129). New York, New York, USA: ACM Press. http://doi.org/10.1145/1951365.1951383

Coelho da Silva, T. L., Nascimento, M. A., de Macêdo, J. A. F., Sousa, F. R. C., & Machado, J. C. (2012). Towards non-intrusive elastic query processing in the cloud. In *Proceedings of the fourth international workshop on Cloud data management - CloudDB '12* (p. 9). New York, New York, USA: ACM Press. http://doi.org/10.1145/2390021.2390024

Coelho da Silva, T. L., Nascimento, M. A., de Macêdo, J. A. F., Sousa, F. R. C., & Machado, J. C. (2013). Non-Intrusive Elastic Query Processing in the Cloud. *Journal of Computer Science and Technology*, *28*(6), 932–947. http://doi.org/10.1007/s11390-013-1389-2

Corradini, F., Polzonetti, A., Re, B., & Tesei, L. (2008). Quality of service in e-government underlines the role of information usability. *International Journal of Information Quality*, *2*(2), 133. http://doi.org/10.1504/IJIQ.2008.022960

Costa, C. M., Leite, C. R. M., & Sousa, A. L. (2015). Service Response Time Measurement Model of Service Level Agreements in Cloud Environment. In *2015 {IEEE} International Conference on Smart City/SocialCom/SustainCom, SmartCity 2015, Chengdu, China, December 19-21, 2015* (pp. 969–974). {IEEE} Computer Society. http://doi.org/10.1109/SmartCity.2015.196

Costa, C. M., Leite, C. R. M., & Sousa, A. L. (2016). Efficient SQL Adaptive Query Processing in Cloud Databases Systems. In *2016 IEEE Conference on Evolving and Adaptive Intelligent Systems (IEEE EAIS 2016)* (pp. 114–121). Natal, RN, Brazil: {IEEE} Computer Society.

Costa, C. M., & Sousa, A. L. (2013). Adaptive Query Processing in Cloud Database Systems. In *Third International Conference on Cloud and Green Computing (CGC)* (pp. 201–202). http://doi.org/10.1109/CGC.2013.39

Coutinho, E. F., de Carvalho Sousa, F. R., Rego, P. A. L., Gomes, D. G., & de Souza, J. N. (2015). Elasticity in cloud computing: a survey. *Annals of Telecommunications - Annales Des Télécommunications*, *70*(7-8), 289–309. http://doi.org/10.1007/s12243-014-0450-7

Curino, C., Jones, E. P. C., Madden, S., & Balakrishnan, H. (2011). Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 international conference on Management of data - SIGMOD '11* (p. 313). New York, New York, USA: ACM Press. http://doi.org/10.1145/1989323.1989357

Das, S., Agarwal, S., Agrawal, D., & El Abbadi, A. (2013). ElasTraS. *ACM Transactions on Database Systems*, *38*(1), 1–45. http://doi.org/10.1145/2445583.2445588

Dean, J., & Ghemawat, S. (2008a). MapReduce: Simplified Data Processing on Large Clusters. *Magazine Communications of the ACM*, *51*(1), 107–113. http://doi.org/10114513274521327492

Dean, J., & Ghemawat, S. (2008b). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, *51*(1), 107. http://doi.org/10.1145/1327452.1327492

Deshpande, A., Ives, Z., & Raman, V. (2007). Adaptive Query Processing. *Foundations and Trends® in Databases*, *1*(1), 1–140. http://doi.org/10.1561/1900000001

Elmasri, R., & Navathe, S. (2010). *Fundamentals of Database Systems* (6th ed.). USA: Addison-Wesley Publishing Company.

Emeakaroha, V. C., Netto, M. A. S., Calheiros, R. N., Brandic, I., Buyya, R., & de Rose, C. A. F. (2012). Towards autonomic detection of SLA violations in Cloud infrastructures. *Future Generation Computer Systems*, *28*(7), 1017–1029. http://doi.org/10.1016/j.future.2011.08.018

Florescu, D., & Kossmann, D. (2009). Rethinking cost and performance of database systems. *ACM SIGMOD Record*, *38*(1), 43. http://doi.org/10.1145/1558334.1558339

Foster, I., & Kesselman, C. (2003). *The Grid 2: Blueprint for a New Computing Infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Garg, S. K., Versteeg, S., & Buyya, R. (2011). SMICloud: A Framework for Comparing and Ranking Cloud Services. In *4th IEEE International Conference on Utility and Cloud Computing* (pp. 210–218). http://doi.org/10.1109/UCC.2011.36

Garg, S. K., Versteeg, S., & Buyya, R. (2013). A framework for ranking of cloud computing services. *Future Generation Computer Systems*, *29*(4), 1012–1023. http://doi.org/10.1016/j.future.2012.06.006

Goiri, Í., Julià, F., Fitó, J. O., Macías, M., & Guitart, J. (2012). Supporting CPU-based guarantees in cloud SLAs via resource-level QoS metrics. *Future Generation Computer Systems*, *28*(8), 1295–1302. http://doi.org/10.1016/j.future.2011.11.004

Gounaris, A., Paton, N. W., Fernandes, A. A. A., & Sakellariou, R. (2002). Adaptive Query Processing: A Survey. In *British National Conference on Databases (BNCOD'02)* (pp. 11–25). London, UK, UK. http://doi.org/10.1007/3-540-45495-0_2

Guitart, J., Carrera, D., Beltran, V., Torres, J., & Ayguadé, E. (2008). Dynamic CPU provisioning for self-managed secure web applications in SMP hosting platforms. *Computer Networks*, *52*(7), 1390–1409. http://doi.org/10.1016/j.comnet.2007.12.009

Iqbal, W., Dailey, M., & Carrera, D. (2009). SLA-Driven Adaptive Resource Management for Web Applications on a Heterogeneous Compute Cloud. In *1st International Conference on Cloud Computing (CloudCom '09)* (pp. 243–253). http://doi.org/10.1007/978-3-642-10665-1_22

Keller, A., & Ludwig, H. (2003). No Title. *Journal of Network and Systems Management*, *11*(1), 57–81. http://doi.org/10.1023/A:1022445108617

Kimball, R., & Ross, M. (2013). *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling* (3rd ed.). Wiley. Retrieved from http://www.amazon.com/Data-Warehouse-Toolkit-Definitive-Dimensional/dp/1118530802/ref=tmm_pap_title_0?_encoding=UTF8&qid=&sr=

Kllapi, H., Sitaridi, E., Tsangaris, M. M., & Ioannidis, Y. (2011). Schedule optimization for data processing flows on the cloud. In *Proceedings of the 2011 international conference on Management of data - SIGMOD '11* (p. 289). New York, New York, USA: ACM Press. http://doi.org/10.1145/1989323.1989355

Larkin, B., & Rose, M. (2015). *2015 ITA Cloud Computing Top Markets Report*. *International Trade Administration*. Retrieved from http://trade.gov/topmarkets/pdf/Cloud_Computing_Top_Markets_Report.pdf

Layton, J. (2015). Monitoring Storage Devices with iostat. *ADMIN Magazine Headquarters Linux New Media, USA, LCC.* Retrieved from http://www.admin-magazine.com/HPC/Articles/Monitoring-Storage-with-iostat

Lee, Y. C., Wang, C., Zomaya, A. Y., & Zhou, B. B. (2010). Profit-Driven Service Request Scheduling in Clouds. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (pp. 15–24). Washington, DC, USA: IEEE. http://doi.org/10.1109/CCGRID.2010.83

Liang, Z., Zou, H., Guo, J., Yang, F., & Lin, R. (2013). Selecting Web Service for Multi-user Based on Multi-QoS Prediction. In *Services Computing (SCC), 2013 IEEE International Conference on* (pp. 551–558). http://doi.org/10.1109/SCC.2013.35

Mangard, S., & Poschmann, A. Y. (Eds.). (2015). *Constructive Side-Channel Analysis and Secure Design* (Vol. 9064). Cham: Springer International Publishing. http://doi.org/10.1007/978-3-319-21476-4

Mian, R., Martin, P., & Vazquez-Poletti, J. L. (2013). Provisioning data analytic workloads in a cloud. *Future Generation Computer Systems*, *29*(6), 1452–1458. http://doi.org/10.1016/j.future.2012.01.008

Morton, K., Balazinska, M., & Grossman, D. (2010). ParaTimer: A Progress Indicator for MapReduce DAGs. In *Proceedings of the 2010 international conference on Management of data - SIGMOD '10* (p. 507). New York, New York, USA: ACM Press. http://doi.org/10.1145/1807167.1807223

Morton, K., Friesen, A., Balazinska, M., & Grossman, D. (2010). Estimating the progress of MapReduce pipelines. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)* (pp. 681–684). IEEE. http://doi.org/10.1109/ICDE.2010.5447919

Naskos, A., Stachtiari, E., Gounaris, A., Katsaros, P., Tsoumakos, D., Konstantinou, I., & Sioutas, S. (2015). Dependable Horizontal Scaling Based on Probabilistic Model Checking. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (pp. 31–40). IEEE. http://doi.org/10.1109/CCGrid.2015.91

Padhy, R. P., Patra, M. R., & Satapathy, S. C. (2012). SLAs in Cloud Systems : The Business Perspective. *International Journal of Computer Science and Technology*, *8491*, 481–488.

Patel, P., Ranabahu, A., & Sheth, A. (2009). Service Level Agreement in Cloud Computing. In *Cloud Workshops at OOPSLA09* (pp. 1–10). Retrieved from http://knoesis.wright.edu/library/download/OOPSLA_cloud_wsla_v3.pdf

PostgreSQL 9.3.9 Documentation. (2015). University of Berkeley. California, USA. Retrieved from https://www.postgresql.org/docs/9.3/static/index.html

Ray, R. (2012). Cloud Computing: Are Small Businesses Embracing The Technology? *Business Insider*, 2. Retrieved from http://www.businessinsider.com/cloud-computing-are-small-businesses-embracing-the-technology-2012-3

Riggs, S., Ciolli, G., Krosing, H., & Bartolini, G. (2015). *PostgreSQL 9 Administration Cookbook* (2nd ed.). Packt Publishing - ebooks Account.

Rogers, J., Papaemmanouil, O., & Cetintemel, U. (2010). A generic auto-provisioning framework for cloud databases. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)* (pp. 63–68). IEEE. http://doi.org/10.1109/ICDEW.2010.5452746

Russell, J., & Cohn, R. (2012). *Mpstat*. Book on Demand Ltd. Retrieved from http://www.linuxcommand.org/man_pages/mpstat1.html

Sanderson, D. (2012). *Programming Google App Engine:* (2nd ed.). O'Reilly Media | Google Press. Retrieved from cloud.google.com/appengine

Schad, J., Dittrich, J., & Quiané-Ruiz, J.-A. (2010). Runtime measurements in the cloud. *Proceedings of the VLDB Endowment*, *3*(1-2), 460–471. http://doi.org/10.14778/1920841.1920902

Selinger, P. G. (1979). Access Path Selection in a Relational DBMS. In *SIGMOD* (pp. 23–24). Boston, USA.

Sharma, U., Shenoy, P., Sahu, S., & Shaikh, A. (2010). *Kingfisher: A system for elastic cost-aware provisioning in the cloud. Technical Report UM-CS-2010-005*. Retrieved from http://web.cs.umass.edu/publication/docs/2010/UM-CS-2010-005.pdf

Sharma, U., Shenoy, P., Sahu, S., & Shaikh, A. (2011). A Cost-Aware Elasticity Provisioning System for the Cloud. In *31st International Conference on Distributed Computing Systems (ICDCS'11)*

(pp. 559–570). Washigntion, DC, USA: IEEE Computer Society. http://doi.org/10.1109/ICDCS.2011.59

Siegel, J., & Perdue, J. (2012). Cloud services measures for global use: The Service Measurement Index (SMI). *Annual SRII Global Conference, SRII*, 411–415. http://doi.org/10.1109/SRII.2012.51

System Analysis and Tuning Guide. (2015). USA. Retrieved from https://www.suse.com/documentation/sles-12/pdfdoc/book_sle_tuning/book_sle_tuning.pdf

Tpc Benchmark™ Ds. (2012). Retrieved from http://www.tpc.org/tpcds/spec/tpcds_1.1.0.pdf

Vaulx, F. D., Simmon, E., & Bohn, R. (2015). *Cloud Computing Service Metrics Description. NIST - National Institute of Standards and Technology - Special Publicaiton* (Vol. 500). USA.

Vigfusson, Y., Silberstein, A., Cooper, B. F., & Fonseca, R. (2009). Adaptively parallelizing distributed range queries. In *Proceedings of the VLDB Endowment* (Vol. 2, pp. 682–693). VLDB Endowment. http://doi.org/10.14778/1687627.1687705

Wu, L., & Buyya, R. (2010). Service Level Agreement (SLA) in Utility Computing Systems. In *Performance and Dependability in Service Computing* (Vol. abs/1010.2, pp. 1–25). IGI Global. http://doi.org/10.4018/978-1-60960-794-4.ch001

Wu, L., Garg, S. K., & Buyya, R. (2011). SLA-Based Resource Allocation for Software as a Service Provider (SaaS) in Cloud Computing Environments. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (pp. 195–204). IEEE. http://doi.org/10.1109/CCGrid.2011.51

Yin, S., Hameurlain, A., & Morvan, F. (2015). Robust Query Optimization Methods With Respect to Estimation Errors. *ACM SIGMOD Record*, *44*(3), 25–36. http://doi.org/10.1145/2854006.2854012

Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., & Sheng, Q. Z. (2003). Quality Driven Web Services Composition. In *Proceedings of the 12th International Conference on World Wide Web* (pp. 411–421). New York, NY, USA: ACM. http://doi.org/10.1145/775152.775211

Zeng, L., Benatallah, B., H.H. Ngu, A., Dumas, M., Kalagnanam, J., & Chang, H. (2004). QoS-Aware Middleware for Web Services Composition. *IEEE Trans. Softw. Eng.*, *30*(5), 311–327. http://doi.org/10.1109/TSE.2004.11

Zhang, L., & Ardagna, D. (2004). SLA based profit optimization in autonomic computing systems. In *Proceedings of the 2nd international conference on Service oriented computing - ICSOC '04* (p. 173). New York, New York, USA: ACM Press. http://doi.org/10.1145/1035167.1035193

Zhao, J., Hu, X., & Meng, X. (2010). ESQP. In *Proceedings of the second international workshop on Cloud data management - CloudDB '10* (p. 1). New York, New York, USA: ACM Press. http://doi.org/10.1145/1871929.1871931

Zheng, Z., Ma, H., Lyu, M. R., & King, I. (2009). WSRec: A Collaborative Filtering Based Web Service Recommender System. In *Web Services, 2009. ICWS 2009. IEEE International Conference on* (pp. 437–444). http://doi.org/10.1109/ICWS.2009.30

Zheng, Z., Zhang, Y., & Lyu, M. R. (2010). Distributed QoS Evaluation for Real-World Web Services. In *Web Services (ICWS), 2010 IEEE International Conference on* (pp. 83–90). http://doi.org/10.1109/ICWS.2010.10

Zhou, A., Wang, S., Zheng, Z., Hsu, C.-H., Lyu, M., & Yang, F. (2014). On Cloud Service Reliability

Enhancement with Optimal Resource Usage. *IEEE Transactions on Cloud Computing*, *PP*(99), 1–1. http://doi.org/10.1109/TCC.2014.2369421

Annex

Annex A1 – Type 1 requests

1. select * from catalog_sales where cs_item_sk >= 1 and cs_item_sk < 300000;
2. select * from catalog_sales where cs_bill_hdemo_sk > 150000;
3. select * from catalog_sales where cs_item_sk >= 300001 and cs_item_sk < 600000;
4. select * from catalog_sales where cs_item_sk > 600000;
5. select * from catalog_sales where cs_item_sk >= 600001 and cs_item_sk < 900000;
6. select * from catalog_sales where cs_item_sk >= 900001 and cs_item_sk < 1200000;
7. select * from catalog_sales where cs_item_sk >= 1200001 and cs_item_sk < 1500000;
8. select * from catalog_sales where cs_item_sk = 900000;
9. select * from catalog_sales where cs_item_sk >= 1500001 and cs_item_sk < 1800000;
10. select cs_bill_hdemo_sk from catalog_sales;
11. select * from catalog_sales where cs_item_sk >= 1800001 and cs_item_sk < 2100000;
12. select * from catalog_sales where cs_bill_hdemo_sk >= 1800001 and cs_bill_hdemo_sk < 2100000;
13. select * from catalog_sales where cs_item_sk >= 2100000 and cs_item_sk < 2400000;
14. select * from catalog_sales where cs_bill_hdemo_sk >= 2100001 and cs_bill_hdemo_sk < 2400000;
15. select * from catalog_sales where cs_item_sk >= 2400001 and cs_item_sk < 2700000;
16. select * from catalog_sales where cs_bill_hdemo_sk >= 2400001 and cs_bill_hdemo_sk < 2700000;
17. select * from catalog_sales where cs_item_sk >= 2700001 and cs_item_sk < 3000000;
18. select * from catalog_sales where cs_bill_hdemo_sk >= 2700001 and cs_bill_hdemo_sk < 3000000;
19. select SUM(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 1 and cs_item_sk <

300000;

20. select SUM(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk > 150000;
21. select SUM(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 300001 and cs_item_sk < 600000;
22. select SUM(cs_item_sk) from catalog_sales where cs_item_sk > 600000;
23. select SUM(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 600001 and cs_item_sk < 900000;
24. select SUM(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 900001 and cs_item_sk < 1200000;
25. select SUM(cs_item_sk) from catalog_sales where cs_item_sk >= 1200001 and cs_item_sk < 1500000;
26. select SUM(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk = 900000;
27. select SUM(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 1500001 and cs_item_sk < 1800000;
28. select SUM(cs_bill_hdemo_sk) from catalog_sales;
29. select SUM(cs_item_sk) from catalog_sales where cs_item_sk >= 1800001 and cs_item_sk < 2100000;
30. select SUM(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 1800001 and cs_bill_hdemo_sk < 2100000;
31. select SUM(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 2100000 and cs_item_sk < 2400000;
32. select SUM(cs_bill_hdemo_sk) from catalog_sales where cs_bill_hdemo_sk >= 2100001 and cs_bill_hdemo_sk < 2400000;
33. select SUM(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 2400001 and cs_item_sk < 2700000;
34. select SUM(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 2400001 and cs_bill_hdemo_sk < 2700000;
35. select SUM(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 2700001 and

cs_item_sk < 3000000;

36. select SUM(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 2700001 and cs_bill_hdemo_sk < 3000000;

37. select AVG(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 1 and cs_item_sk < 300000;

38. select AVG(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk > 150000;

39. select AVG(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 300001 and cs_item_sk < 600000;

40. select AVG(cs_item_sk) from catalog_sales where cs_item_sk > 600000;

41. select AVG(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 600001 and cs_item_sk < 900000;

42. select AVG(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 900001 and cs_item_sk < 1200000;

43. select AVG(cs_item_sk) from catalog_sales where cs_item_sk >= 1200001 and cs_item_sk < 1500000;

44. select AVG(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk = 900000;

45. select AVG(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 1500001 and cs_item_sk < 1800000;

46. select AVG(cs_bill_hdemo_sk) from catalog_sales;

47. select AVG(cs_item_sk) from catalog_sales where cs_item_sk >= 1800001 and cs_item_sk < 2100000;

48. select AVG(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 1800001 and cs_bill_hdemo_sk < 2100000;

49. select AVG(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 2100000 and cs_item_sk < 2400000;

50. select AVG(cs_bill_hdemo_sk) from catalog_sales where cs_bill_hdemo_sk >= 2100001 and cs_bill_hdemo_sk < 2400000;

51. select AVG(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 2400001 and

cs_item_sk < 2700000;

52. select AVG(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 2400001 and cs_bill_hdemo_sk < 2700000;

53. select AVG(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 2700001 and cs_item_sk < 3000000;

54. select AVG(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 2700001 and cs_bill_hdemo_sk < 3000000;

55. select COUNT(*) from catalog_sales where cs_item_sk >= 1 and cs_item_sk < 300000;

56. select COUNT(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk > 150000;

57. select COUNT(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 300001 and cs_item_sk < 600000;

58. select COUNT(*) from catalog_sales where cs_item_sk > 600000;

59. select COUNT(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 600001 and cs_item_sk < 900000;

60. select COUNT(*) from catalog_sales where cs_item_sk >= 900001 and cs_item_sk < 1200000;

61. select COUNT(cs_item_sk) from catalog_sales where cs_item_sk >= 1200001 and cs_item_sk < 1500000;

62. select COUNT(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk = 900000;

63. select COUNT(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 1500001 and cs_item_sk < 1800000;

64. select COUNT(cs_bill_hdemo_sk) from catalog_sales;

65. select COUNT(cs_item_sk) from catalog_sales where cs_item_sk >= 1800001 and cs_item_sk < 2100000;

66. select COUNT(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 1800001 and cs_bill_hdemo_sk < 2100000;

67. select COUNT(*) from catalog_sales where cs_item_sk >= 2100000 and cs_item_sk < 2400000;

68. select COUNT(cs_bill_hdemo_sk) from catalog_sales where cs_bill_hdemo_sk >= 2100001 and

cs_bill_hdemo_sk < 2400000;

69. select COUNT(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 2400001 and cs_item_sk < 2700000;

70. select COUNT(*) from catalog_sales where cs_bill_hdemo_sk >= 2400001 and cs_bill_hdemo_sk < 2700000;

71. select COUNT(*) from catalog_sales where cs_item_sk >= 2700001 and cs_item_sk < 3000000;

72. select COUNT(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 2700001 and cs_bill_hdemo_sk < 3000000;

73. select MIN(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 1 and cs_item_sk < 300000;

74. select MIN(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk > 150000;

75. select MIN(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 300001 and cs_item_sk < 600000;

76. select MIN(cs_item_sk) from catalog_sales where cs_item_sk > 600000;

77. select MIN(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 600001 and cs_item_sk < 900000;

78. select MIN(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 900001 and cs_item_sk < 1200000;

79. select MIN(cs_item_sk) from catalog_sales where cs_item_sk >= 1200001 and cs_item_sk < 1500000;

80. select MIN(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk = 900000;

81. select MIN(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 1500001 and cs_item_sk < 1800000;

82. select MIN(cs_bill_hdemo_sk) from catalog_sales;

83. select MIN(cs_item_sk) from catalog_sales where cs_item_sk >= 1800001 and cs_item_sk < 2100000;

84. select MIN(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 1800001 and cs_bill_hdemo_sk < 2100000;

85. select MIN(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 2100000 and cs_item_sk < 2400000;
86. select MIN(cs_bill_hdemo_sk) from catalog_sales where cs_bill_hdemo_sk >= 2100001 and cs_bill_hdemo_sk < 2400000;
87. select MIN(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 2400001 and cs_item_sk < 2700000;
88. select MIN(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 2400001 and cs_bill_hdemo_sk < 2700000;
89. select MIN(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 2700001 and cs_item_sk < 3000000;
90. select MIN(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 2700001 and cs_bill_hdemo_sk < 3000000;
91. select MAX(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 1 and cs_item_sk < 300000;
92. select MAX(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk > 150000;
93. select MAX(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 300001 and cs_item_sk < 600000;
94. select MAX(cs_item_sk) from catalog_sales where cs_item_sk > 600000;
95. select MAX(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 600001 and cs_item_sk < 900000;
96. select MAX(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 900001 and cs_item_sk < 1200000;
97. select MAX(cs_item_sk) from catalog_sales where cs_item_sk >= 1200001 and cs_item_sk < 1500000;
98. select MAX(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk = 900000;
99. select MAX(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 1500001 and cs_item_sk < 1800000;
100. select MAX(cs_bill_hdemo_sk) from catalog_sales;

101. select MAX(cs_item_sk) from catalog_sales where cs_item_sk >= 1800001 and cs_item_sk < 2100000;

102. select MAX(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 1800001 and cs_bill_hdemo_sk < 2100000;

103. select MAX(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 2100000 and cs_item_sk < 2400000;

104. select MAX(cs_bill_hdemo_sk) from catalog_sales where cs_bill_hdemo_sk >= 2100001 and cs_bill_hdemo_sk < 2400000;

105. select MAX(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 2400001 and cs_item_sk < 2700000;

106. select MAX(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 2400001 and cs_bill_hdemo_sk < 2700000;

107. select MAX(cs_bill_hdemo_sk) from catalog_sales where cs_item_sk >= 2700001 and cs_item_sk < 3000000;

108. select MAX(cs_item_sk) from catalog_sales where cs_bill_hdemo_sk >= 2700001 and cs_bill_hdemo_sk < 3000000;

Annex A2 – Type 2 requests

1.
```
select  *
from store_sales,household_demographics,time_dim, store
where ss_sold_time_sk = time_dim.t_time_sk
and ss_hdemo_sk = household_demographics.hd_demo_sk
and ss_store_sk = s_store_sk;
```

2.
```
select  ss_item_sk
from store_sales, time_dim
where ss_sold_time_sk = t_time_sk;
```

3.
```
select  store_sales.*
from store_sales, customer_demographics, date_dim, item, promotion
where ss_sold_date_sk = d_date_sk and
        ss_item_sk = i_item_sk and
        ss_cdemo_sk = cd_demo_sk and
        ss_promo_sk = p_promo_sk and
        cd_gender = 'M';
```

4.
```
select  *
from store_sales,household_demographics,time_dim
where ss_sold_time_sk = time_dim.t_time_sk
and ss_hdemo_sk = household_demographics.hd_demo_sk;
```

5.
```
select  ss_item_sk
from store_sales, time_dim
where ss_sold_time_sk = t_time_sk and ss_item_sk > 100000;
```

6.
```
select  store_sales.*
from store_sales, customer_demographics, date_dim, item, promotion
where ss_sold_date_sk = d_date_sk and
        ss_item_sk = i_item_sk and
        ss_cdemo_sk = cd_demo_sk and
        ss_promo_sk = p_promo_sk and
        cd_gender = 'F';
```

7.
```
select  count(store.s_store_name)
   from store_sales,household_demographics,time_dim, store
   where ss_sold_time_sk = time_dim.t_time_sk
           and ss_hdemo_sk = household_demographics.hd_demo_sk
           and ss_store_sk = s_store_sk
           and time_dim.t_hour = 8
           and time_dim.t_minute >= 30
```

```
        and household_demographics.hd_dep_count = 5
```

8. select i_item_id, avg(ss_quantity) agg1, avg(ss_list_price) agg2, avg(ss_coupon_amt) agg3, avg(ss_sales_price) agg4
```
from store_sales, customer_demographics, date_dim, item, promotion
where ss_sold_date_sk = d_date_sk and
    ss_item_sk = i_item_sk and
    ss_cdemo_sk = cd_demo_sk and
    ss_promo_sk = p_promo_sk;
```

9. select i_item_id, avg(ss_quantity)
```
from store_sales, customer_demographics, date_dim, item, promotion
where ss_sold_date_sk = d_date_sk and
    ss_item_sk = i_item_sk and
    ss_cdemo_sk = cd_demo_sk and
    ss_promo_sk = p_promo_sk and
    cd_gender = 'M' and
    cd_marital_status = 'M' and
    cd_education_status = '4 yr Degree' and
    (p_channel_email = 'N' or p_channel_event = 'N') and
    d_year = 2001;
```

10. select sum(cs_ext_discount_amt) as "excess discount amount"
```
from catalog_sales,item,date_dim
where i_manufact_id = 577
    and i_item_sk = cs_item_sk
    and d_date between '1998-03-18' and
    (cast('1998-03-18' as date) + 90)
    and d_date_sk = cs_sold_date_sk;
```

11. select 1.3 * avg(cs_ext_discount_amt)
```
from catalog_sales,date_dim
where cs_item_sk = i_item_sk
    and d_date between '1998-03-18' and
    (cast('1998-03-18' as date) + 90)
    and d_date_sk = cs_sold_date_sk
```

12. select *
```
from store_sales,date_dim
where c.c_customer_sk = ss_customer_sk and
    ss_sold_date_sk = d_date_sk and
    d_year = 2001
```

13. select cd_gender, cd_marital_status, cd_education_status, count(*) cnt1, cd_purchase_estimate, count(*) cnt2, cd_credit_rating, count(*) cnt3
```
from  customer c,customer_address ca,customer_demographics
where  c.c_current_addr_sk = ca.ca_address_sk and
```

```
cd_demo_sk = c.c_current_cdemo_sk;
```

14. select *
```
from web_sales,date_dim
where c.c_customer_sk = ws_bill_customer_sk and
    ws_sold_date_sk = d_date_sk and
    d_year = 2001 and
    d_moy between 2 and 2+2;
```

15. select *
```
from catalog_sales,date_dim
where c.c_customer_sk = cs_ship_customer_sk and
    cs_sold_date_sk = d_date_sk and
    d_year = 2001 and
    d_moy between 2 and 2+2;
```

16. select count(*)
```
from  customer c,customer_address ca,customer_demographics
where  c.c_current_addr_sk = ca.ca_address_sk and
    cd_demo_sk = c.c_current_cdemo_sk;
```

17. select ss_item_sk
```
from store_sales, time_dim
where ss_sold_time_sk = t_time_sk and ss_item_sk >= 100000 and ss_item_sk < 400000;
```

18. select ss_item_sk
```
from store_sales, time_dim
where ss_sold_time_sk = t_time_sk and ss_item_sk < 900000;
```

19. select count(*)
```
from store_sales,household_demographics,time_dim, store
where ss_sold_time_sk = time_dim.t_time_sk
    and ss_hdemo_sk = household_demographics.hd_demo_sk
    and ss_store_sk = s_store_sk
    and time_dim.t_minute >= 30
    and store.s_store_name = 'ese';
```

Annex A3 – Type 3 requests

1. select count(*)

```
from store_sales,household_demographics,time_dim, store
where ss_sold_time_sk = time_dim.t_time_sk
   and ss_hdemo_sk = household_demographics.hd_demo_sk
   and ss_store_sk = s_store_sk
   and time_dim.t_hour = 8
   and time_dim.t_minute >= 30
   and household_demographics.hd_dep_count = 5
   and store.s_store_name = 'ese'
order by count(*);
```

2. select i_item_id, avg(ss_quantity) agg1, avg(ss_list_price) agg2, avg(ss_coupon_amt) agg3, avg(ss_sales_price) agg4

```
 from store_sales, customer_demographics, date_dim, item, promotion
 where ss_sold_date_sk = d_date_sk and
       ss_item_sk = i_item_sk and
       ss_cdemo_sk = cd_demo_sk and
       ss_promo_sk = p_promo_sk and
       cd_gender = 'M' and
       cd_marital_status = 'M' and
       cd_education_status = '4 yr Degree' and
       (p_channel_email = 'N' or p_channel_event = 'N') and
       d_year = 2001
 group by i_item_id
 order by i_item_id;
```

3. select asceding.rnk, i1.i_product_name best_performing, i2.i_product_name worst_performing

```
from(select *
     from (select item_sk,rank() over (order by rank_col asc) rnk
           from (select ss_item_sk item_sk,avg(ss_net_profit) rank_col
                 from store_sales ss1
                 where ss_store_sk = 30
                 group by ss_item_sk
                 having avg(ss_net_profit) > 0.9*(select avg(ss_net_profit) rank_col
                                                  from store_sales
                                                  where ss_store_sk = 30
                                                    and ss_hdemo_sk is null
                                                  group by ss_store_sk))V1)V11
     where rnk  < 11) asceding,
    (select *
     from (select item_sk,rank() over (order by rank_col desc) rnk
           from (select ss_item_sk item_sk,avg(ss_net_profit) rank_col
                 from store_sales ss1
                 where ss_store_sk = 30
```

```
                 group by ss_item_sk
                 having avg(ss_net_profit) > 0.9*(select avg(ss_net_profit) rank_col
                                                  from store_sales
                                                  where ss_store_sk = 30
                                                    and ss_hdemo_sk is null
                                                  group by ss_store_sk))V2)V21
     where rnk  < 11) descending,
item i1,
item i2
where asceding.rnk = descending.rnk
  and i1.i_item_sk=asceding.item_sk
  and i2.i_item_sk=descending.item_sk
order by asceding.rnk;
```

4. select sum(cs_ext_discount_amt) as "excess discount amount"

```
from
   catalog_sales,item,date_dim
where i_manufact_id = 577
and i_item_sk = cs_item_sk
and d_date between '1998-03-18' and
        (cast('1998-03-18' as date) + 90)
and d_date_sk = cs_sold_date_sk
and cs_ext_discount_amt
     > (
         select 1.3 * avg(cs_ext_discount_amt)
         from catalog_sales,date_dim
         where cs_item_sk = i_item_sk
          and d_date between '1998-03-18' and
                             (cast('1998-03-18' as date) + 90)
          and d_date_sk = cs_sold_date_sk
      );
```

5. select cd_gender, cd_marital_status, cd_education_status, count(*) cnt1, cd_purchase_estimate,

```
  count(*) cnt2, cd_credit_rating, count(*) cnt3
 from  customer c,customer_address ca,customer_demographics
 where  c.c_current_addr_sk = ca.ca_address_sk and
  ca_state in ('KS','ND','WV') and
  cd_demo_sk = c.c_current_cdemo_sk and
  exists (select *
          from store_sales,date_dim
          where c.c_customer_sk = ss_customer_sk and
                ss_sold_date_sk = d_date_sk and
                d_year = 2001 and
                d_moy between 2 and 2+2) and
   (not exists (select *
            from web_sales,date_dim
```

```
          where c.c_customer_sk = ws_bill_customer_sk and
                ws_sold_date_sk = d_date_sk and
                d_year = 2001 and
                d_moy between 2 and 2+2) and
    not exists (select *
          from catalog_sales,date_dim
          where c.c_customer_sk = cs_ship_customer_sk and
                cs_sold_date_sk = d_date_sk and
                d_year = 2001 and
                d_moy between 2 and 2+2))
 group by cd_gender, cd_marital_status, cd_education_status, cd_purchase_estimate,
        cd_credit_rating
 order by cd_gender, cd_marital_status, cd_education_status, cd_purchase_estimate,
        cd_credit_rating;
```

6. WITH all_sales AS (

```
 SELECT d_year,i_brand_id,i_class_id,i_category_id,i_manufact_id
       ,SUM(sales_cnt) AS sales_cnt
       ,SUM(sales_amt) AS sales_amt
 FROM (SELECT d_year,i_brand_id,i_class_id,i_category_id,i_manufact_id
             ,cs_quantity - COALESCE(cr_return_quantity,0) AS sales_cnt
             ,cs_ext_sales_price - COALESCE(cr_return_amount,0.0) AS sales_amt
       FROM catalog_sales JOIN item ON i_item_sk=cs_item_sk
                          JOIN date_dim ON d_date_sk=cs_sold_date_sk
                          LEFT JOIN catalog_returns ON (cs_order_number=cr_order_number
                                                    AND cs_item_sk=cr_item_sk)
       WHERE i_category='Shoes'
       UNION
       SELECT d_year,i_brand_id,i_class_id,i_category_id,i_manufact_id
             ,ss_quantity - COALESCE(sr_return_quantity,0) AS sales_cnt
             ,ss_ext_sales_price - COALESCE(sr_return_amt,0.0) AS sales_amt
       FROM store_sales JOIN item ON i_item_sk=ss_item_sk
                        JOIN date_dim ON d_date_sk=ss_sold_date_sk
                        LEFT JOIN store_returns ON (ss_ticket_number=sr_ticket_number
                                                AND ss_item_sk=sr_item_sk)
       WHERE i_category='Shoes'
       UNION
       SELECT d_year,i_brand_id,i_class_id,i_category_id,i_manufact_id
             ,ws_quantity - COALESCE(wr_return_quantity,0) AS sales_cnt
             ,ws_ext_sales_price - COALESCE(wr_return_amt,0.0) AS sales_amt
       FROM web_sales JOIN item ON i_item_sk=ws_item_sk
                      JOIN date_dim ON d_date_sk=ws_sold_date_sk
                      LEFT JOIN web_returns ON (ws_order_number=wr_order_number
                                            AND ws_item_sk=wr_item_sk)
       WHERE i_category='Shoes') sales_detail
 GROUP BY d_year, i_brand_id, i_class_id, i_category_id, i_manufact_id)
 SELECT  prev_yr.d_year AS prev_year,curr_yr.d_year AS year,curr_yr.i_brand_id
                       ,curr_yr.i_class_id,curr_yr.i_category_id,curr_yr.i_manufact_id
```

```
                        ,prev_yr.sales_cnt AS prev_yr_cnt,curr_yr.sales_cnt AS curr_yr_cnt
                        ,curr_yr.sales_cnt-prev_yr.sales_cnt AS sales_cnt_diff
                        ,curr_yr.sales_amt-prev_yr.sales_amt AS sales_amt_diff
 FROM all_sales curr_yr, all_sales prev_yr
 WHERE curr_yr.i_brand_id=prev_yr.i_brand_id  AND curr_yr.i_class_id=prev_yr.i_class_id
   AND curr_yr.i_category_id=prev_yr.i_category_id AND curr_yr.i_manufact_id=prev_yr.i_manufact_id
   AND curr_yr.d_year=2000 AND prev_yr.d_year=2000-1
   AND CAST(curr_yr.sales_cnt AS DECIMAL(17,2))/CAST(prev_yr.sales_cnt AS DECIMAL(17,2))<0.9
 ORDER BY sales_cnt_diff;
```

7. with ssr as

```
 (select  s_store_id as store_id,
          sum(ss_ext_sales_price) as sales,
          sum(coalesce(sr_return_amt, 0)) as returns,
          sum(ss_net_profit - coalesce(sr_net_loss, 0)) as profit
  from store_sales left outer join store_returns on
         (ss_item_sk = sr_item_sk and ss_ticket_number = sr_ticket_number),
     date_dim, store, item,promotion
 where ss_sold_date_sk = d_date_sk
       and d_date between cast('2000-08-10' as date)
                  and (cast('2000-08-10' as date) +  30)
       and ss_store_sk = s_store_sk
       and ss_item_sk = i_item_sk
       and i_current_price > 50
       and ss_promo_sk = p_promo_sk
       and p_channel_tv = 'N'
 group by s_store_id)
 ,
 csr as
 (select  cp_catalog_page_id as catalog_page_id,
          sum(cs_ext_sales_price) as sales,
          sum(coalesce(cr_return_amount, 0)) as returns,
          sum(cs_net_profit - coalesce(cr_net_loss, 0)) as profit
  from catalog_sales left outer join catalog_returns on
         (cs_item_sk = cr_item_sk and cs_order_number = cr_order_number),
     date_dim, catalog_page, item, promotion
 where cs_sold_date_sk = d_date_sk
       and d_date between cast('2000-08-10' as date)
                  and (cast('2000-08-10' as date) +  30)
        and cs_catalog_page_sk = cp_catalog_page_sk
       and cs_item_sk = i_item_sk
       and i_current_price > 50
       and cs_promo_sk = p_promo_sk
       and p_channel_tv = 'N'
group by cp_catalog_page_id)
 ,
 wsr as
```

```
(select  web_site_id,
         sum(ws_ext_sales_price) as sales,
         sum(coalesce(wr_return_amt, 0)) as returns,
         sum(ws_net_profit - coalesce(wr_net_loss, 0)) as profit
 from web_sales left outer join web_returns on
        (ws_item_sk = wr_item_sk and ws_order_number = wr_order_number),
    date_dim, web_site, item, promotion
 where ws_sold_date_sk = d_date_sk
      and d_date between cast('2000-08-10' as date)
                and (cast('2000-08-10' as date) +  30)
       and ws_web_site_sk = web_site_sk
      and ws_item_sk = i_item_sk
      and i_current_price > 50
      and ws_promo_sk = p_promo_sk
      and p_channel_tv = 'N'
group by web_site_id)
  select  channel, id, sum(sales) as sales, sum(returns) as returns, sum(profit) as profit
 from  (select 'store channel' as channel, 'store' || store_id as id, sales, returns, profit
 from   ssr
 union all
 select 'catalog channel' as channel, 'catalog_page' || catalog_page_id as id, sales, returns, profit
 from  csr
 union all
 select 'web channel' as channel, 'web_site' || web_site_id as id, sales, returns, profit
 from   wsr
 ) x
 group by channel, id
 order by channel,id;
```

8.
```
select  i_brand_id brand_id, i_brand brand, i_manufact_id, i_manufact,
          sum(ss_ext_sales_price) ext_price
 from date_dim, store_sales, item,customer,customer_address,store
 where d_date_sk = ss_sold_date_sk   and ss_item_sk = i_item_sk
   and i_manager_id=13   and d_moy=11
   and d_year=2001   and ss_customer_sk = c_customer_sk
   and c_current_addr_sk = ca_address_sk   and substr(ca_zip,1,5) <> substr(s_zip,1,5)
   and ss_store_sk = s_store_sk
 group by i_brand,i_brand_id,i_manufact_id,i_manufact
 order by ext_price desc,i_brand,i_brand_id,i_manufact_id,i_manufact;
```

9.
```
with ws as
  (select d_year AS ws_sold_year, ws_item_sk,
    ws_bill_customer_sk ws_customer_sk,
    sum(ws_quantity) ws_qty,
    sum(ws_wholesale_cost) ws_wc,
```

```
    sum(ws_sales_price) ws_sp
   from web_sales
   left join web_returns on wr_order_number=ws_order_number and ws_item_sk=wr_item_sk
   join date_dim on ws_sold_date_sk = d_date_sk
   where wr_order_number is null
   group by d_year, ws_item_sk, ws_bill_customer_sk
   ),
cs as
  (select d_year AS cs_sold_year, cs_item_sk,
    cs_bill_customer_sk cs_customer_sk,
    sum(cs_quantity) cs_qty,
    sum(cs_wholesale_cost) cs_wc,
    sum(cs_sales_price) cs_sp
   from catalog_sales
   left join catalog_returns on cr_order_number=cs_order_number and cs_item_sk=cr_item_sk
   join date_dim on cs_sold_date_sk = d_date_sk
   where cr_order_number is null
   group by d_year, cs_item_sk, cs_bill_customer_sk
   ),
ss as
  (select d_year AS ss_sold_year, ss_item_sk,
    ss_customer_sk,
    sum(ss_quantity) ss_qty,
    sum(ss_wholesale_cost) ss_wc,
    sum(ss_sales_price) ss_sp
   from store_sales
   left join store_returns on sr_ticket_number=ss_ticket_number and ss_item_sk=sr_item_sk
   join date_dim on ss_sold_date_sk = d_date_sk
   where sr_ticket_number is null
   group by d_year, ss_item_sk, ss_customer_sk
   )
 select
ss_sold_year, ss_item_sk, ss_customer_sk,
round(ss_qty/(coalesce(ws_qty+cs_qty,1)),2) ratio,
ss_qty store_qty, ss_wc store_wholesale_cost, ss_sp store_sales_price,
coalesce(ws_qty,0)+coalesce(cs_qty,0) other_chan_qty,
coalesce(ws_wc,0)+coalesce(cs_wc,0) other_chan_wholesale_cost,
coalesce(ws_sp,0)+coalesce(cs_sp,0) other_chan_sales_price
from ss
left join ws on (ws_sold_year=ss_sold_year and ws_item_sk=ss_item_sk and
ws_customer_sk=ss_customer_sk)
left join cs on (cs_sold_year=ss_sold_year and cs_item_sk=cs_item_sk and
cs_customer_sk=ss_customer_sk)
where coalesce(ws_qty,0)>0 and coalesce(cs_qty, 0)>0 and ss_sold_year=1999
order by
  ss_sold_year, ss_item_sk, ss_customer_sk,
  ss_qty desc, ss_wc desc, ss_sp desc,
  other_chan_qty,
```

```
 other_chan_wholesale_cost,
 other_chan_sales_price,
 round(ss_qty/(coalesce(ws_qty+cs_qty,1)),2);
```

10. with wss as

```
(select d_week_seq,
       ss_store_sk,
       sum(case when (d_day_name='Sunday') then ss_sales_price else null end) sun_sales,
       sum(case when (d_day_name='Monday') then ss_sales_price else null end) mon_sales,
       sum(case when (d_day_name='Tuesday') then ss_sales_price else  null end) tue_sales,
       sum(case when (d_day_name='Wednesday') then ss_sales_price else null end) wed_sales,
       sum(case when (d_day_name='Thursday') then ss_sales_price else null end) thu_sales,
       sum(case when (d_day_name='Friday') then ss_sales_price else null end) fri_sales,
       sum(case when (d_day_name='Saturday') then ss_sales_price else null end) sat_sales
 from store_sales,date_dim
 where d_date_sk = ss_sold_date_sk
 group by d_week_seq,ss_store_sk
 )  select  s_store_name1,s_store_id1,d_week_seq1
      ,sun_sales1/sun_sales2,mon_sales1/mon_sales2
      ,tue_sales1/tue_sales2,wed_sales1/wed_sales2,thu_sales1/thu_sales2
      ,fri_sales1/fri_sales2,sat_sales1/sat_sales2
 from (select s_store_name s_store_name1,wss.d_week_seq d_week_seq1
       ,s_store_id s_store_id1,sun_sales sun_sales1
       ,mon_sales mon_sales1,tue_sales tue_sales1
       ,wed_sales wed_sales1,thu_sales thu_sales1
       ,fri_sales fri_sales1,sat_sales sat_sales1
  from wss,store,date_dim d
  where d.d_week_seq = wss.d_week_seq and
       ss_store_sk = s_store_sk and
       d_month_seq between 1200 and 1200 + 11) y,
 (select s_store_name s_store_name2,wss.d_week_seq d_week_seq2
       ,s_store_id s_store_id2,sun_sales sun_sales2
       ,mon_sales mon_sales2,tue_sales tue_sales2
       ,wed_sales wed_sales2,thu_sales thu_sales2
       ,fri_sales fri_sales2,sat_sales sat_sales2
  from wss,store,date_dim d
  where d.d_week_seq = wss.d_week_seq and  ss_store_sk = s_store_sk and
       d_month_seq between 1200+ 12 and 1200 + 23) x
 where s_store_id1=s_store_id2
   and d_week_seq1=d_week_seq2-52
 order by s_store_name1,s_store_id1,d_week_seq1;
```

Annex A4 – paper 1 – (2013)

Adaptive Query Processing in Cloud Database Systems

Clayton Maciel Costa
HASLab / INESC TEC
Instituto Federal do Rio Grande do Norte / Univ. do Minho
Ipanguaçu, Brasil / Braga, Portugal
clayton.maciel@ifrn.edu.br

António Luís Sousa
HASLab / INESC TEC
Universidade do Minho
Braga, Portugal
als@di.uminho.pt

***Abstract* – In cloud environments, resources should be acquired and released automatically and quickly at runtime. Thereby, the implementation of traditional query optimization strategies in cloud platforms can have a poor performance, because they cannot predict future availability and/or release of resources. In such scenarios, adaptive query processing can adapt itself to the available resources to run queries and, consequently, present an acceptable performance in response to a query. However, traditional and adaptive query optimizers main objective is to reduce response time. Moreover, in the context of cloud computing, users and providers of services expect to get answers in time to guarantee the SLA. Therefore, we propose a framework that uses adaptive query processing based on heuristic rules and cost of failing the SLA. It will be implemented on structured data, considering that some cloud computing platforms support SQL queries directly or indirectly, which makes this problem relevant.**

Keywords—cloud computing; database systems; adaptive query processing

I. INTRODUCTION

In the cloud computing model, the cloud providers have to optimize their profits while servicing several clients. This is obtained recurring to some level of abstraction (virtualization) according to the type of service, such as: storage, processing, bandwidth and active user accounts. To guarantee the quality of service (QoS - Quality of Service) there are SLA (Service Level Agreement) associated to the service delivery. The SLA is a contract formalized between a cloud service provider and its customers that define the level of service expected from the service provider. SLAs are output-based in that their purpose is specifically to define what the customer will receive. The SLA provides several metrics on the levels of availability, functionality, performance, penalties, billing etc [1, 2, 3]. In this work, we use the Service Response Time SLA metric, which is the total time between the instance the query is presented to the system and the time it completes its execution in the system.

Following this context, adaptive query processing has the ability to dynamically and automatically allocate or release resources (elasticity of resources) during the query runtime. This technique is very important when statistical information about the services available may be minimal and the availability of physical resources may change. This is a typical scenario of cloud environments. However, traditional and adaptive query optimzers' main objective is to reduce response time. Moreover, in the context of cloud computing, users and providers of services expect to get answers in time to guarantee the service SLA. Therefore, we propose a framework that uses adaptive query processing based on heuristic rules and the cost of failing the SLA. Figure 1 presents the Framework Architecture which uses dynamic re-optimization techniques. The Section II presents briefly related works and the Section III we explain in detail each component of Framework. Finally, Section IV shows the conclusions.

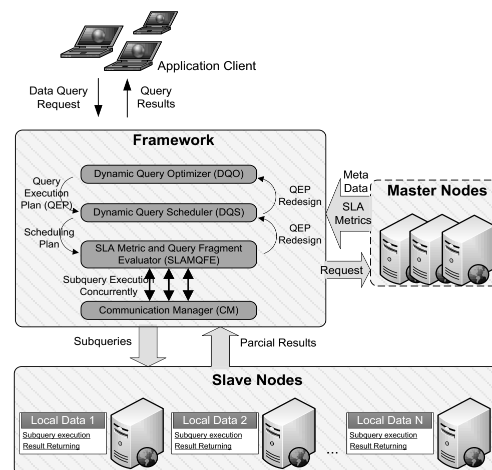


Fig. 1. Framework Architecture

II. RELATED WORK

There are several works related to efficient query processing in cloud database systems [3, 4, 5]. Most of these works provide the basis for new technologies of query processing/optimization. For instance, ESQP, an efficient SQL query processing algorithm using data replicas in cloud storage, it is based on traditional techniques of parallel/distributed DBMS, such as use of index and pipeline [4]. SLA-Tree improves the efficiency using scheduling (environment of multiple queries with different profiles to be executed),

[4]. SLA-Tree improves the efficiency using scheduling (environment of multiple queries with different profiles to be executed), dispatching (environment of several servers for one query to be execute), and capacity planning (current dynamic workload) [5]. These problems are very important to cloud database systems and they are based on classic techniques.

I. Framework

Our framework integrates adaptive/dynamic re-optimization techniques by performing distributed queries in several steps. Each component of the Framework (Figure 1) utilizes adaptive strategies applied at runtime of the query and their costs are based on the Service Response Time QoS parameter, defined by SMI-CSMIC consortium [1, 2]. The components are specified below:

Dynamic Query Optimizer (DQO): It is used to construct an optimized query plan based on Service Response Time. The main difference of traditional optimizers is to construct query plans considering the SLA time restriction. For this purpose, it is important to consider that the initial SLA time agreed must be sufficient, observing the technological limits of the service provider.

Dynamic Query Scheduler (DQS): It is used to schedule the execution of distributed query plans. This component optimizes dynamically the queries at runtime, which is based on Service Response Time and the variation of resources utilized to process the query (for instance, average CPU utilization, available memory and estimated rates to processing of each slave node). Indeed, the queries submitted to DQS will be processed in the "best hosts" among all available slave nodes. In this work, the definition of a "best hosts group" depends on system variables, such as: available resources, resources needed to meet SLA requirements and optimization objectives, which can relate directly with SLA requirements, for instance, minimizing costs and maximizing the probability of success, or can relate indirectly, for example, the better workload balancing.

SLA Metric and Query Fragment Evaluator (SLAMQFE): Given an optimized and scheduled query plan, the aim of this component is to monitor the query execution. In case of being necessary to make a re-optimizing, the component loads the query to DQS component. The monitoring verifies, periodically, the probability of a query to be executed before a SLA time restriction. Therefore, the SLAMQFE reevaluates periodically all queries execution plans at runtime to check the probability of violate the SLA, whether the probability is low, the query continues its execution, otherwise, the query will be re-optimized. The probability is estimated according to DBMS metadata of slave nodes, slave nodes configurations, the query plans and a statistical table with specific maximum service response time. The statistical table serves as a cache, as it is storing successful probabilities (queries that did not violate the SLA time) of previously executed queries. The table aids to reduce the computing overhead to calculate an estimated time to execute a query. Furthermore, it will be automatically populated by the Framework according to its use. The Table I presents the SLAMQFE component algorithm.

TABLE I. SLAMQFE Component Algorithm

```
SLAMQFE ALGORITHM (Q, TLSLA): RETURN Tr
    TLSLA; //SLA Time.
    Tr = 0; //Query Processing Total Time. Default = 0.
    Tini; //Query Processing Start Time.
    Tprop; //Average Time Estimated to Process Completely the Query Q.
    TabProp; //Statistical Table.
    SlaveNode[]; // Slave Nodes Available.
    Q; //Query Plan.
1.   BEGIN
2.      Tini = getCurrentTime();
3.      Tprop = SlaveNode[i..i+1].getSRT().comparedTo(execute(Q));
4.      IF (Tprop > TLSA) THEN
5.         SlaveNode[i..i+1] = new SlaveNode();  //New Slave Node Instances
6.         Schedule SubQueryPlan[j..j+1] for SlaveNode[i..i+1];
7.         Tr = Tr + (getCurrentTime() - Tini);
8.         Tr += (SLAMQFE(SubQueryPlan[j..j+1],(TLSA-Tr)));
9.      ELSE
10.        SlaveNode[i..i+1] = execute(Q); //Query Processing on Slave Node
11.        Tr = Tr + (getCurrentTime() - Tini);
12.        TabProp[k..k+1] = Q and Tr; //Statistical Table
13.     ENDIF
14.  RETURN Tr;
15.  END
```

.

In the case that the query cannot be executed before time SLA, the Framework must calculate the execution time nearer to SLA time and the cloud provider must inform to the customer, discussing the penalties. In this case, the adaptive optimization traditional algorithm will execute because at this moment the fastest response time becomes more important than the SLA time.

II. Conclusion

In this work, we present a Framework using adaptive techniques to efficient processing of queries in cloud database systems. Our solution is restricted to requests of database access and it based on the QoS parameters, formalized by SMI-CSMIC consortium [1, 2]. Furthermore, our approach has not restriction of elasticity and/or scalability of their algorithms and it is non-intrusive. In future work, we intend to realize experiments on large scale with large volume of data and queries in the cloud.

References

[1] "Cloud Service Measurement Index Consortium (CSMIC)". URL: http://www.cloudcommons.com/group/cloud-service-measurement-initiative-consortium/home.

[2] S. K. Garg, S. Versteeg and R. Buyya, "A framework for ranking of cloud computing services", Future Gener. Comput. Syst. 29, pp. 1012-1023. DOI=10.1016/j.future.2012.06.006, 2012.

[3] R. Buyya, S. K. Garg and R. N. Calheiros, "SLA-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions", Proceedings of the 2011 International Conference on Cloud and Service Computing (CSC '11). IEEE Computer Society, Washington, DC, USA, 1-10. DOI=10.1109/CSC.2011.6138522, 2011.

[4] J. Zhao, X. Hu and X. Meng, "ESQP: an efficient SQL query processing for cloud data management", Proceedings of the second international workshop on Cloud data management (CloudDB '10). ACM, New York, NY, USA, pp. 1-8. DOI=10.1145/1871929.1871931, 2010.

[5] Y. Chi, H. J. Moon, H. Hacig\&\#252;m\&\#252;\&\#351 and J. Tatemura, "SLA-tree: a framework for efficiently supporting SLA-based decisions in cloud computing", Proceedings of the 14th International Conference on Extending Database Technology (EDBT/ICDT '11), Anastasia Ailamaki, Sihem Amer-Yahia, Jignesh Pate, Tore Risch, Pierre Senellart, and Julia Stoyanovich (Eds.). ACM, New York, NY, USA, pp. 129-140. DOI=10.1145/1951365.1951383, 2011.

Annex A5 – paper 2 – (2015)

Service Response Time Measurement Model of Service Level Agreements in Cloud Environment

Clayton Maciel Costa
High-Assurance Software Lab / INESC TEC
Instituto Federal do Rio Grande do Norte / Universidade do Minho
Ipanguaçu, Brasil / Braga, Portugal
clayton.maciel@ifrn.edu.br

Cicília Raquel Maia Leite
Software Engineering Lab
Universidade do Estado do Rio Grande do Norte
Mossoró, Brasil
ciciliamaia@gmail.com

António Luís Sousa
High-Assurance Software Lab / INESC TEC
Universidade do Minho
Braga, Portugal
als@di.uminho.pt

***Abstract* – In cloud environments, resources should be acquired and released automatically and quickly at runtime. Therefore, ensuring the desired QoS is a great challenge for the cloud service provider. Moreover, it increases when we have large amount of data to be manipulated in this environment. Considering that, performance is an important requirement for most customers when they migrate their applications to the cloud. In this paper, we propose a model for measuring a Service Response Time estimated for different request types on large databases available in a cloud environment. This work allows the cloud service provider and its customers establish an appropriate SLA relative to performance expected of services available in the cloud. Finally, the model was evaluated in Amazon EC2 cloud infrastructure and the TPC-DS like benchmark was used for generating a database of structured data, considering that some cloud computing platforms support SQL queries directly or indirectly. This makes the proposed solution relevant for these kind of problems.**

Keywords-cloud computing; service level agreement; performance; service response time

I. INTRODUCTION

In the cloud computing model, the cloud providers have to optimize their profits while servicing several customers. This is obtained recurring to some level of abstraction (virtualization) according to the type of service, such as: storage, processing, bandwidth and active user accounts [1]. To ensure QoS (Quality of Service), there are SLA (Service Level Agreements) associated to the service delivery. The SLA is a formal contract defined between a cloud service provider and its customers that describe the level of service expected from provider. SLAs are output-based in that their purpose is specifically to define what the customers expect to receive. The SLA is composed of several metrics on the levels of availability, functionality, performance, penalties, billing, etc [1, 2, 9]. In this work, our focus is the SRT (Service Response Time) performance parameter of SLA, which corresponds to the total time between time that the request/query arrives to the provider and at the time it completes its execution in the system.

The performance metrics in the SLA, including SRT, is one of the most important requirements for most of customers when they migrate their applications to the cloud, as it relates to expectations of their applications in the cloud performance. From the user's point of view, this parameter is considered one of the main QoS parameters [4]. However, nowadays one can see that the major cloud providers like Amazon [5, 6] and Google [7] emphasizing guaranteeing of availability, CPU instance and cost measure. Therefore, the SRT parameter is not completely handled or inappropriately treated in SLA. Therefore, in order to ensure customer expectations relative to performance, cloud service providers have to understand how to incorporate suitably the SRT parameter in their SLA.

The measuring of SRT parameter in SLA is a very complex task because it depends on many system variables, such as request type, database model and current rate system performance. Furthermore, it is common in a cloud environment that the requests rate is highly unpredictable. Therefore, guaranteeing a specific response time for any level of request rate is regarded as a significant challenge to the paradigm of cloud computing. Moreover, the growth of data stored in the cloud makes this challenge ever harder.

Thereby, in this paper, we propose a model for estimating the SRT for different types of requests on large databases available in the cloud environment. Our propose is to allow the cloud service provider and its clients to establish an appropriate SLA relative to SRT performance of services available in the cloud. The proposed model is a non-intrusive solution and it can be applied when companies wish to migrate their applications, OLAP or not, to cloud services providers, with the goal to allocate computational resources on demand, to ensure the quality of service in terms of service response time. Finally, the model was evaluated utilizing Amazon EC2 cloud infrastructure small instances type and the TPC-DS [8] like benchmark was used only for generating an OLAP database of structured data, considering that some cloud computing platforms support SQL queries directly or indirectly, this makes the proposed solution suitable for these kind of problems.

This paper is organized as follows. Section II presents related works. Section III presents the SRT definition, its measurement model, and finally, their tools. Section IV shows the experiments of proposed model. Finally, Section V shows the conclusions and future works.

II. RELATED WORKS

In the context of SLA agreements, it is possible to identify two important research areas:

(i) QoS Parameters Definition: currently we have many cloud providers offering different prices, parameters and performance levels, even different services with when those providers offer similar services. In addition, several

infrastructure [2, 9]. Thereby, there is a wide range of different contracts with different SLA requirements. Thus, it becomes difficult for a customer to choose the most suitable provider to migrate their applications. For example, in [1, 2, 9] the authors are focused on the definition and measurement of SLA parameters in general. Then, they have main objective to define a global view of QoS parameters and their metrics used by cloud service providers. In context of this paper, the authors define the SRT how fast the service is ready for use. Moreover, they do not define the time of effective execution of any request. (ii) **QoS Parameters Ensure**: The provider must consider how to optimize the use of resources and how to preserve the QoS parameters that it must be guaranteed according to SLA. In this scenario, it is very important to consider the possibility of new requests and their priorities; even when running other tasks, the provider must use efficiently the resources to guarantee the requirements. For example, financial organizations usually require security and privacy QoS requirements, but for example, the availability QoS requirement, although important, it is not a priority of these organizations [9]. For example, in [3] proposes a resource-provisioning framework in a public cloud to execute requests in large amounts of data. This work proposes an SLA cost model and presents a provisioning method based on SLA time, predicting the best value to execute requests at any given time. In [4] presents an SLA-oriented resource manager focused on cloud computing and based on open source technology. It provides adaptive resource allocation and dynamic load balancing for Web applications in order to ensure a SLA. However, these works define the Service Response Time arbitrarily.

I. Service Response Time Measurement Model

To try to resolve the above challenges, it is necessary to define and to standardize the QoS parameters used by most cloud provider and finally, to provide a methodology to compare services of different cloud. Consequently, the customers can make a better selection of a cloud service provider [2, 9]; because, an appropriately selected service provider increases the probability that SLA requirements are guaranteed.

In this context, nowadays, many companies have migrated their applications and data to the cloud due to the benefits of this technology. From the user's point of view, the SRT metric is considered one of the main QoS parameters. However, the major cloud providers have ignored or inappropriately treated the SRT parameter in SLA due to its complexity.

This way, our proposal is an estimation of SRT parameter for customers who wish to migrate their applications to the cloud but have no idea of the performance offered by the cloud provider for its applications. Therefore, the main contribution of this work is to propose a model for obtaining the SRT, so it can be treated adequately in SLA contracts. Thereby, it is necessary to define formally, what is a Recommended SRT.

A. Definition

In this work, the SRT (Service Response Time) corresponds to the time that a service takes to execute effectively a request [2]. This way, the SRT of a service starts when a customer request is ready to execute and it finishes when the request executes effectively. Including, for example, startup time of virtual machine or wait of a fragment request, etc.

Let ***Ri*** a database access request in a cloud, where ***i*** represents one of the following request types: (i) select-range and/or select-aggregation, (ii) select-joins or (iii) select-sets-grouping-nesting. The **A**verage **S**ervice **R**esponse **T**ime of a request ***Ri*** (***ASRT***) executed by ***n*** physical/virtual machines is given by:

$$ASRT_{Ri} = \sum_{Ri} SRT_{Ri}/n \tag{1}$$

where SRT_{Ri} is the time between the moment a request ***Ri*** is ready to run and the service executes the request effectively.

Let A_{Ri} be a set of averages service response time for all type ***i*** requests, i.e. $A_{Ri} = \{ ASRT_{Ri}^{1}, ASRT_{Ri}^{2}, ASRT_{Ri}^{3}, \ldots ASRT_{Ri}^{K} \}$, where ***k*** is the quantity of type ***i*** requests. Let $A\ A_{Ri}$ be a set of half the size of A_{Ri} (***k/2***) with the highest averages of A_{Ri}.

Thus, the Recommended SRT ($RSRT_{Ri}$) for a set of Type ***i*** Requests deployed in the cloud is given by median of $A\ A_{Ri}$:

$$RSRT_{Ri} = \uparrow A \uparrow A_{Ri\frac{k}{4}+0.5} \quad for\ odd\ k \tag{2}$$

or

$$RSRT_{Ri} = \frac{\left\{\uparrow A \uparrow A_{Ri\frac{k}{4}} + \uparrow A \uparrow A_{Ri\frac{k}{4}+1}\right\}}{2} \quad for\ even\ k \tag{3}$$

It is worth noting that Recommended SRT presents a pessimistic estimate of response time, because it is based on requests that require more time to process, i.e. on median of the upper half that represents the highest requests response time.

The discussion of Recommended SRT occurs in SLA contract definition phase, in which evaluate several tasks of customers applications on the cloud service provider. The applications most used and complex are defined and selected. In this work, complex applications represent applications that use high load of system (large use of CPU and disk read/write).

B. SRT Measurement Model

According [11], a cloud computing platform is a cluster with hundreds or thousands of PCs (nodes) for data computing and storage. There are two types of nodes in the cluster: master nodes and slave nodes. Master nodes store metadata and they manage the all cluster slave nodes. The slave nodes store the data and their replicas for security.

In this context, the Figure 1 shows the steps to obtain the Recommended SRT of a cloud computing platform: **(1)** acquisition of customer applications; **(2)** selection and classification of applications according to the request types: (i) select-range and/or select-aggregation, (ii) select-joins or (iii) select-sets-grouping-nesting; **(3)** experiments of customer applications deployed on master nodes and slave nodes of cloud provider; and finally, **(4)** analysis of results, which is a Recommended SRT for each request type and system load.

It is worth noting that in contract level, the trust and validation of the results will depend mainly on good practice in step 2, because good selectivity of customer applications will reduce SLA violation.

In step 3, to assist the tests were implemented three tools and deployed in cloud provider, they are COS (CPU Overload

Simulator), DOS (Disk I/O Overload Simulator) and SRT Calculator. Following we detail each of them and their functions.

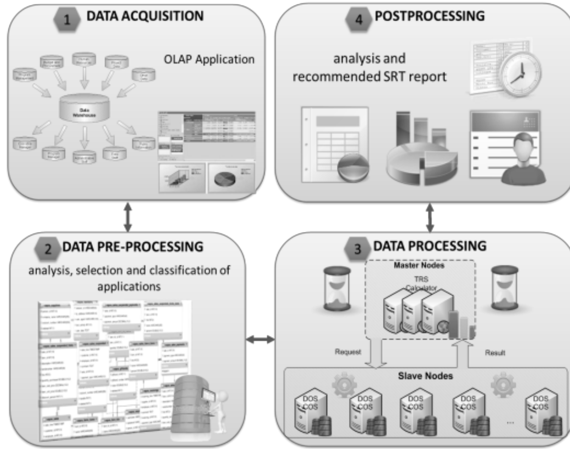


Figure. 1. Steps to obtain the Recommended SRT.

A. COS and DOS: Overload Simulator

The tools were deployed in slave nodes of cloud service provider and it serves, respectively, to simulate partial and total CPU overload, i.e. the overload can also be by the processor core and serves to simulate disk read/write overload.

The COS tool generates an overload of threads of similar execution priority of the processes running in operating system. Although the set of threads are running in the same process, if the COS tool executes itself more than once it will generate a set of threads in different processes causing large number of processes of equal priority competing for the processor. Thus, to overload the processor, the tool executes itself generating a large amount of processes, each having a large number of threads.

This tool allows serial execution by processor core, i.e. each core will be overloading by time. Thus, it allows configuring how many cores are overloaded. To analyze the CPU, the user can use the *sysstat* tool, in which checks the processor in real time. The *sysstat* tool is package with a collection of performance monitoring tools for major Linux distribution.

The DOS tool generates an overload of threads of database access requests with similar execution priority of the processes running in operating system. Unlike COS tool, DOS simply run once, generating a very large set of threads of equal priority in the same process, overloading the system and competing with any another database access request that arrives at the processor.

This tool allows also defining the quantity of threads to be generate, in which each one simulates a database access request in the machine. This way, the tool allows a wide variation in quantity of bytes of reading and writing from/to disk.

B. SRT Calculator

The SRT Calculator tool was deployed in master nodes of provider and it serves to execute the tests in the specified slave nodes.

The SRT Calculator computes a set the Recommended SRT as defined in Section *III – A* and it generates a parameterized report to be analyzed and discussed between the cloud service provider and it customer.

The report presents the Recommended SRT for each request type and overload variation in slave nodes, through the COS and DOS tools. Beyond, for each of request type, statistical parameters are generated from the set of the requests response times (seconds), such as arithmetic average, sample variance, standard deviation, mode and coefficient of variation. Therefore, a specialist can evaluate these parameters to validate the results. For example, a high standard deviation indicates that the data points are spread out over a wider range of values. Then, the result can not be reliable. For the better understanding, the summary of SRT Calculator algorithm is shown below:

```
Config_VM; //Configuration File of Physique/Virtual Machine (Slave Nodes).
REQUEST-TYPE[i]; //Requests Type File, i equals 1, 2 or 3.
1.  BEGIN
2.      SLAVE-NODE[i..n] = Config_VM;
3.      FOR EACH SLAVE-NODE DO
4.          FOR EACH REQUEST-TYPE DO
5.              ExecuteRequest(SLAVE-NODE[i], REQUEST-TYPE [i]) ;
6.          ENDFOR
7.          REPORT(REQUEST-TYPE);
8.      ENDFOR
9.      REPORT(ALL-REQUEST);
10.
11.     VOID REPORT(REQUEST)
12.         BEGIN
13.             Avegare(); //(nanosseg) -- (milisseg) -- (seg) -- (min)
14.             Sample Variance(); //(seg)
15.             Mode; //(seg)
16.             Coefficient of Variation(); ///(seg)
17.             Recommended SRT ();//(nanosseg) -- (milisseg) -- (seg) -- (min)
18.         END
19. END
```

To use the SRT Calculator is necessary to classify the customer applications in one of three requests type. In addition, a set of physical/virtual machines of the cloud must be selected to store customer applications. This way it is necessary to configure the following files: (1) network configuration file and database connection of slave nodes; (2) configuration file for requests with select-range and/or aggregating functions requests; (3) configuration file for requests with one or more joins; and finally, (4) configuration file for requests with set of operations, grouping and/or nesting. Next Section presents a case study of the proposed model using Amazon EC2 cloud infrastructure and TCP-DS to generate an OLAP database and requests.

II. EXPERIMENTAL EVALUATION

This Section presents a case study of the proposed model to obtain the Recommended SRT utilizing small instances of Amazon EC2 cloud infrastructure. First, we present the environment and the experiments methodology. Then, we show the requests used and finally, we present the results obtained as well as its analysis.

A. Experimental Environment

The tools (COS, DOS and SRT Calculator) were implemented in Java language using concurrent programming with threads and API based on OpenMP (Open Multi-

Processing) [10]. They were deployed in the Amazon EC2 cloud infrastructure in small instances (homogeneous environment). Due to the limitations of Amazon, it was used 16 VMs (Virtual Machines), each one with an Intel Xeon Processor with turbo up to 3.3GHz, 1.7 GB of main memory and 160 GB of disk storage.

It was created an AMI (Amazon Machine Image) of VM with the database. This image allows startup a new VM quickly. The Amazon EBS (Elastic Block Store) was used to storage the AMI. Therefore, the startup time and instantiation of VM, the time of network authentication and database connection were considers in experiments.

Each VM runs the Ubuntu 12.04 operating system and PostgreSQL 9.3 DBMS. This work focuses on OLAP applications with very large and complex database. Thus, the TPC-DS was used to generate a database of approximately 13 GB, fully replicated in each VM. Furthermore, 50 requests of several complexities were selected. Therefore, the database and the requests generated represents the customer applications.

A. Methodology

Figure 2 presents the methodology of experiments. As shown, in a VM, chosen arbitrarily, it was deployed the SRT Calculator tool (master node). It communicates with others VMs (slave nodes). Furthermore, the 50 requests obtained of TPC-DS were classified according to level of complexity between three types.

Thus, the SRT Calculator executes all request of each type in all VMs, varying the overload on the slave nodes through of COS and DOS tool (they were deployed in slave nodes).

The PP (Processor Performance) represents levels of overload of CPU. The DP (Disk Performance) consists of percentage level of overload in Megabytes of reading and/or writing on disk. To view the rate of CPU and disk usage, the *sysstat* and *dstat* tools were used.

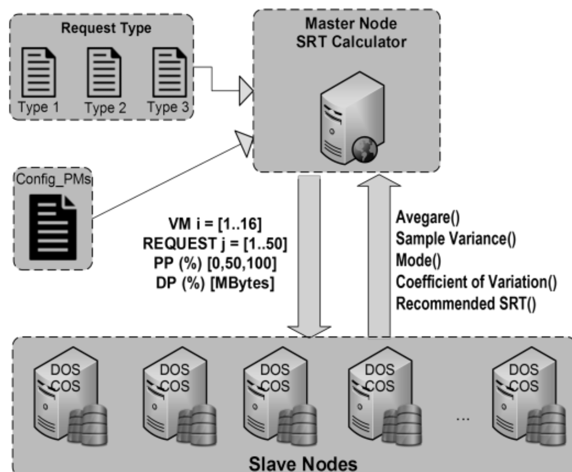


Figure. 1. Methodology of experiments to obtain the Recommended SRT.

B. Requests

The TPC-DS offers many database requests for experiments, which for this case study were selected 50 requests. The classification of each request was based on results of *explain analyze* command of the PostgreSQL DBMS.

Type 1 Requests are select-range and/or select-aggregation requests. They have approximately 140,000 tuples of selectivity and it uses the *catalog_sales* table of TPC-DS. Type 2 Requests are select-joins requests and optional select-aggregation functions. The selectivity of these requests varied between 1000 and 60,000 tuples and it uses at least 20 different tables of TPC-DS. Finally, Type 3 Requests are select-sets-grouping-nesting requests and, optional select-aggregation and select-joins. They present very complex query plan and its selectivity is between 100,000 and 200,000 tuples. It uses at least 20 different tables of database generated by TPC-DS Benchmark.

C. Results

The results were grouped by type of request and overload variation in slave nodes. So, to Type 1 Requests, the result of experiments on all VMs are presented in Figure 3. It shows the SRT averages to 50 requests executed on all VMs (all slave nodes) when they are not overloaded (current) and when they are with CPU and Disk Overloaded.

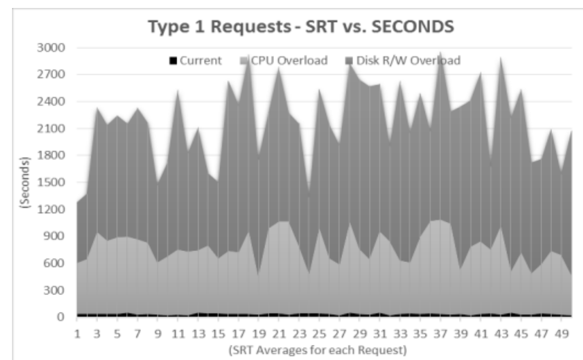


Figure. 2. SRT averages on all VMs for Type 1 Requests.

Therefore, based on definition of Recommended SRT and considering that the processor and disk not overloaded (*Current Status* in Figure 3) we have the following result:

::TYPE 1 REQUESTS::
Average: 34,46(s)
Sample Variance: 71,02897959
Standard Deviation: 8,42786922
Mode: 35
Coefficient of Variation: 24,45696233
Recommended SRT: 42(s)

Overload with COS tool (*CPU Overload* in Figure 3), the result is as follows:

::TYPE 1 REQUESTS::
Average: 741,3(s)
Sample Variance: 32053,03061
Standard Deviation: 179,0336019
Mode: 620
Coefficient of Variation: 24,15130202
Recommended SRT: 865(s)

Overload with DOS tool (*Disk R/W Overload* in Figure 3), the following values were found:

::TYPE 1 REQUESTS::
Average: 1402,16(s)
Sample Variance: 134948,0555
Standard Deviation: 367,3527671
Mode: 1450
Coefficient of Variation: 26,19906196
Recommended SRT: 1718(s)

To Type 2 Requests, the result of experiments in all VMs are presented in Figure 4. It shows the SRT averages to 50 requests executed on all VMs (all slave nodes) when they are not overloaded (current) and when they are with CPU and Disk Overloaded.

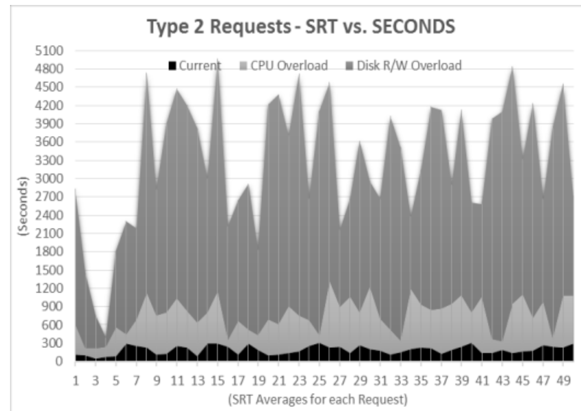


Figure. 1. SRT averages on all VMs for Type 2 Requests.

Therefore, based on definition of Recommended SRT and considering the processor and disk not overloaded (*Current Status* in Figure 4) we have the following result:

::TYPE 2 REQUESTS::
Average: 187,6(s)
Sample Variance: 5059,755102
Standard Deviation: 71,13195556
Mode: 288
Coefficient of Variation: 37,91682066
Recommended SRT: 242(s)

Overload with COS tool (*CPU Overload* in Figure 4), the result is as follows:

::TYPE 2 REQUESTS::
Average: 567,86(s)
Sample Variance: 76106,16367
Standard Deviation: 275,8734559
Mode: 127
Coefficient of Variation: 48,58124466
Recommended SRT: 783(s)

Overload with DOS tool (*Disk R/W Overload* in Figure 4), the following values were found:

::TYPE 2 REQUESTS::
Average: 2514,8(s)
Sample Variance: 977864,7347
Standard Deviation: 988,8704337
Mode: 2618
Coefficient of Variation: 39,32203093
Recommended SRT: 3455(s)

To Type 3 Requests, the result of experiments on all VMs are presented in Figure 5. It shows the SRT averages to 50 requests executed on all VMs (all slave nodes) when they are not overloaded (current) and when they are with CPU and Disk Overloaded.

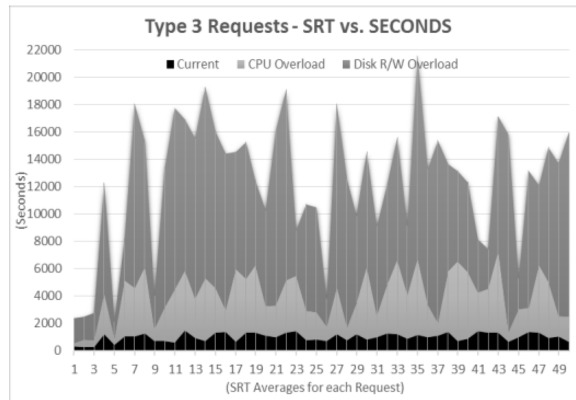


Figure. 2. SRT averages on all VMs for Type 3 Requests.

Therefore, based on definition of Recommended SRT and considering that processor and disk not overloaded (*Current Status* in Figure 5) we have the following result:

::TYPE 3 REQUESTS::
Average: 981,52(s)
Sample Variance: 106462,9486
Standard Deviation: 326,286605
Mode: 1001
Coefficient of Variation: 33,24299097
Recommended SRT: 1283(s)

Overload with COS tool (*CPU Overload* in Figure 5), the result is as follows:

::TYPE 3 REQUESTS::
Average: 3044,6(s)
Sample Variance: 2667600,653
Standard Deviation: 1633,279111
Mode: 2960
Coefficient of Variation: 53,64511301
Recommended SRT: 4431(s)

Overload with DOS tool (*Disk R/W Overload* in Figure 5), the following values were found:

::TYPE 3 REQUESTS::
Average: 8284,32(s)
Sample Variance: 16121155,85
Standard Deviation: 4015,11592
Mode: 8200
Coefficient of Variation: 48,46645133
Recommended SRT: 11391(s)

A. Analysis of Results

Figure 6 and Table I summarizes the results of Recommended SRT. According to results, the SRT was higher when CPU or disk were overwhelmed, mainly the disk, which caused also overload in CPU.

TABLE I. RECOMMENDED SRT RESULT

Request Type	Recommended SRT		
	Current	*CPU Overload*	*Disk R/W Overload*
1	42(s)	865(s)	1718(s)
2	242(s)	783(s)	3455(s)
3	1283(s)	4431(s)	11391(s)

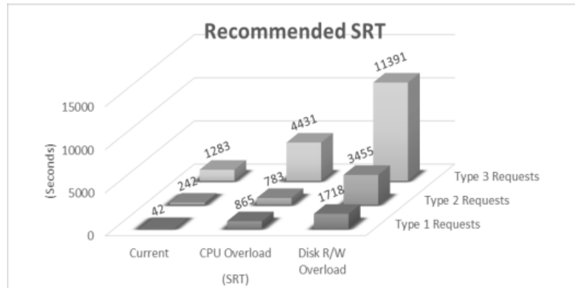


Figure. 1. Recommended SRT Result.

It is worth noting that the number of rows and columns returned from a request (query selectivity) increases significantly the total time of its execution. For example, Type 3 Requests have very high selectivity and therefore, they have higher Recommended SRT. Other example, even with overloaded CPU, Type 2 Requests have Recommended SRT smaller than Type 3 Requests with current CPU utilization.

In general, Type 1 and Type 2 Requests have smaller Recommended SRTs than Type 3 requests. However, Type 1 requests when overloaded CPU, its Recommended SRT is bigger than Type 2 Request because the selectivity of Type 1 Requests is greater than Type 2 Requests.

The results obtained provide the basis for negotiation between the cloud service provider and its customers establish an expected Service Response Time of their services. Furthermore, these values can be used by monitoring tools, when a limit value is achieved, the environment can react recovering or minimizing the consequences of SLA violation. For example, allocating, reallocating and/or releasing resources at run-time.

Therefore, a well-chosen provider brings benefits to both sides: the customer will have accurate information about the performance of their applications when they are performed in the cloud and the provider will reduce future penalties, as it has the provider's expected behavior after the tests.

I. CONCLUSION

In this paper, we presented a model that allows the cloud service provider and its customers establish an appropriate SLA relative to SRT performance of its applications available in the cloud.

The proposed model is a non-intrusive solution and can be applied when companies wish to migrate their applications, OLAP or not, to cloud services providers, with the goal to allocate computational resources on demand, to ensure the quality of service in terms of Service Response Time.

Finally, this work focuses on OLAP applications with very large and complex database, it was evaluated utilizing structured data of TPC-DS like Benchmark, considering that many cloud computing platforms support SQL requests directly or indirectly, this makes the proposed solution relevant for these kind of problems.

As future work, we will deploy our propose, beyond Amazon, in an Azure and Google Cloud Platform, using similar VM flavors, and then compare the response time between the different public cloud providers. Furthermore, other future work consists to use specialist systems to the automatic classification of applications according to the request types, as well as, to the automatic analysis of results. Other work comprises to replace DOS and COS tools by others benchmark tools for example, *pgbench* tool that allows a greater variation of performance parameters.

REFERENCES

[1] “Cloud Service Measurement Index Consortium (CSMIC)”. URL: http://www.cloudcommons.com/group/cloud-service-measurement-initiative-consortium/home.

[2] S. K. Garg, S. Versteeg and R. Buyya, “A framework for ranking of cloud computing services”, Future Gener. Comput. Syst. 29, 2012, pp. 1012-1023. DOI=10.1016/j.future.2012.06.006.

[3] R. Mian, P. Martin, and J. L. Vazquez-Poletti. 2013. Provisioning data analytic workloads in a cloud. Future Gener. Comput. Syst. 29, 2013, pp. 1452-1458. DOI=10.1016/j.future.2012.01.008 http://dx.doi.org/10.1016/j.future.2012.01.008

[4] W. Iqbal, M. Dailey, and D. Carrera. “SLA-Driven Adaptive Resource Management for Web Applications on a Heterogeneous Compute Cloud”, In Proceedings of the 1st International Conference on Cloud Computing (CloudCom '09), Martin Gilje Jaatun, Gansen Zhao, and Chunming Rong (Eds.). Springer-Verlag, Berlin, Heidelberg, 2009, pp. 243-253. DOI=10.1007/978-3-642-10665-1_22 http://dx.doi.org/10.1007/978-3-642-10665-1_22.

[5] AWS EC2 Service Level Agreement. URL: http://aws.amazon.com/ec2-sla.

[6] AWS S3 Service Level Agreement. URL: http://aws.amazon.com/s3-sla.

[7] D. Sanderson. “Programming Google App Engine” (2nd ed.), O'Reilly Media, 2012, p. 536, ISBN 978-1449398262. URL: cloud.google.com/appengine.

[8] TPC-DS. TPC-DS Benchmark. 2014. URL: http://www.tpc.org/tpcds/

[9] V. C. Emeakaroha, M. A. S. Netto, R. N. Calheiros, I. Brandic, R. Buyya, and C. A. F. De Rose. “Towards autonomic detection of SLA violations in Cloud infrastructures”, Future Gener. Comput. Syst. 28, 2012, pp. 1017-1029. DOI=10.1016/j.future.2011.08.018 http://dx.doi.org/10.1016/j.future.2011.08.018.

[10] “About the OpenMP ARB”. OpenMP.org. 2013-07-11. Retrieved 2013-08-14.”. URL: http://openmp.org/wp/about-openmp/.

[11] H. Kllapi, E. Sitaridi, M. M. Tsangaris and Y. Ioannidis. “Schedule optimization for data processing flows on the cloud”, In *Proceedings of the 2011 international conference on Management of data* (SIGMOD '11). ACM, New York, NY, USA, 2011, pp. 289-300. DOI=10.1145/1989323.1989355 http://doi.acm.org/10.1145/1989323.1989355.

Annex A6 – paper 3 – (2016)

Efficient SQL Adaptive Query Processing in Cloud Databases Systems

Clayton Maciel Costa
High-Assurance Software Lab / INESC TEC
Instituto Federal do Rio Grande do Norte / Universidade do Minho
Ipanguaçu, Brasil / Braga, Portugal
clayton.maciel@ifrn.edu.br

Cicília Raquel Maia Leite
Software Engineering Lab
Universidade do Estado do Rio Grande do Norte
Mossoró, Brasil
ciciliamaia@gmail.com

António Luís Sousa
High-Assurance Software Lab / INESC TEC
Universidade do Minho
Braga, Portugal
als@di.uminho.pt

***Abstract* – Nowadays, many companies have migrated their applications and data to the cloud. Among other benefits of this technology, the ability to answer quickly business requirements has been one of the main motivations. Thereby, in cloud environments, resources should be acquired and released automatically and quickly at runtime. This way, to ensure QoS, the major cloud providers emphasize ensuring of availability, CPU instance and cost measure in their SLAs (Service Level Agreements). However, the QoS performance are not completely handled or inappropriately treated in SLAs. Although from the user's point of view, it is considered one of the main QoS parameters. Therefore, the aim of this work consists in development of a solution to efficient query processing on large databases available in the cloud environments. It integrates adaptive re-optimization at query runtime and their costs are based on the SRT (Service Response Time) QoS performance parameter of SLA. Finally, the solution was evaluated in Amazon EC2 cloud infrastructure and the TPC-DS like benchmark was used for generating a database.**

Keywords* - *cloud computing; service level agreement; performance; service response time

I. INTRODUCTION

Nowadays, many companies have migrated their applications and data to the cloud due to the benefits of this technology. For example, the applications and data stored in the cloud can be accessed anywhere independent of local software platform. Another important benefit is the significant reduction of costs and time of experimentation and development when compared with local infrastructure because it eliminates the need of one or more physical servers in company increasing the space, minimizing the necessity of specialists for repairs.

In the cloud computing model, the cloud providers have to optimize their profits while servicing several customers. This is obtained recurring to some level of abstraction (virtualization) according to the type of service, such as: storage, processing, bandwidth and active user accounts [1]. To ensure QoS (Quality of Service), there are SLA (Service Level Agreements) associated to the service delivery. The SLA is a formal contract defined between a cloud service provider and its customers that describe the level of service expected from provider. SLAs are output-based in that their purpose is specifically to define what the customers expect to receive. The SLA is composed of several metrics on the levels of availability, functionality, performance, penalties, billing, etc [1]–[3]. This work focuses on the SRT (Service Response Time) performance parameter of SLA, which corresponds to the total time between time that the request/query arrives to the provider and at the time, it completes its execution in the system.

Following this context, adaptive query processing has the ability to dynamically and automatically allocate or release resources (elasticity of resources) during the query runtime. This technique is very important when statistical information about the services available may be minimal and the availability of physical resources may change. This is a typical scenario of cloud environments. However, traditional and adaptive query optimzers' main objective is to reduce response time. Moreover, in the context of cloud computing, users and providers of services expect to get answers in time to guarantee the service SLA.

The performance parameters of a SLA are the most important requirements for most customers when they decide to migrate their applications to the cloud. Because these parameters are directly related to the performance of their applications in the cloud. Therefore, from the user's point of view, they are considered one of the main QoS parameters [4]. Nowadays, one can see that the major cloud providers like Amazon [5] and Google [6] emphasizing availability, CPU instance and cost measure. Therefore, the SRT performance parameter is not completely handled or inappropriately treated in SLA.

The measuring of SRT parameter in SLA is a very complex task because it depends on many system variables, such as request type, database model and current rate system performance. Furthermore, it is common in a cloud environment that the requests rate is highly unpredictable. Therefore, guaranteeing a specific response time for any level of request rate is regarded as a significant challenge to the paradigm of cloud computing. Moreover, the growth of data stored in the cloud makes this challenge ever harder.

Therefore, the aim of this work consists in development of a solution to efficient query processing on large databases available in the cloud environment. It integrates adaptive re-optimization at runtime of the query and their costs are based on the SRT QoS parameter. Moreover, it is restricted to relational database access requests, it has not restriction of elasticity and/or scalability of their algorithms and a non-intrusive approach. Finally, it was evaluated utilizing Amazon EC2 cloud infrastructure small instances type and the TPC-DS [7] like

978-1-5090-2583-1/16/$31.00 ©2016 IEEE

2016 IEEE Conference on Evolving and Adaptive Intelligent Systems

benchmark was used for generating an OLAP database of structured data.

This paper is organized as follows. Section 2 presents related works. Section 3 presents the strategies for adaptive processing of different types of queries in cloud databases systems. Section 4 shows the experimental evaluation. Finally, Section 5 shows the conclusions and future works.

II. Related Works

Currently, several researches have been focused in search of techniques for efficient query processing in the cloud. As shown in Table I, the works in [8]–[17] do not use the strategy of monitoring during requests execution. The algorithm in [18] provides adaptive optimizing the response time of queries. The algorithm partitions and adaptively identifies the best level of parallelism for each query. The authors propose an adaptive provisioning algorithm for only select-range queries and consider variations in performance of VMs (Virtual Machines). However, it does not observe the SLA agreement and does not specify the frequency of the monitoring algorithm during query execution. The work in [4] presents an adaptive SLA-oriented resource manager. However, it only predicts the provisioning of resources and does not check DBMS variables for database access requests, addressing only the level of the application server layer. The approach in [19] uses the strategy of regular monitoring intervals during requests execution and therefore does not consider that VMs may have different performance. In addition, it limits its scope to single pipeline queries (queries without joins). The approach in [20] presents a non-intrusive framework for adaptive queries processing in database implanted in cloud environment. This work observes query response time of the SLA contract; makes adaptive monitoring considering the heterogeneous environment, and therefore, it considers that the VMs may have different performances. However, the scope is limited only to select-range queries.

This way, we can observe that most works in the literature focus on shorter execution time of a query and on the prediction of resources to be used for query through the current system context. These works may not be suitable in highly unpredictable environments on the availability of resources. In turn, others works emphasize on adaptive query processing. However, they present limitations of elasticity and/or scalability in their algorithms, the absence of adaptive monitoring query processing, use of intrusive solutions and/or use proprietary technology and do not use formalisms in defining the QoS parameters in their solutions and as a result, the same service may have different understanding among cloud service providers.

III. Adaptive Query Processing in the Cloud

This Section presents the solution for efficient adaptive processing of different types of queries (database access requests) in cloud environment. This way, it will presented the definition of a request and the strategies of execution for each type of request.

A. Requests

In computational context, a request corresponds a task to be executed by a Web Service sent by a customer who has access to this service. This work focuses on database access requests on OLAP applications in the cloud environment. A request message is a SQL query composed by one or more tables and it can be of different types.

TABLE I. Related Works

Related Work	Adaptive Query Processing	Based on SRT on the SLA contract	Restriction of Query	Provisioning or Release of Resources
[8], [21]	No	Yes	Not restricted	Provisioning of Resources
[9]	No	No	Not restricted	Provisioning of Resources
[18]	Yes	No	Select-range	Provisioning of Resources
[4]	Yes	Yes	Not applied	Provisioning of Resources
[10]	No	Yes	Not applied	Provisioning of Resources
[19]	Yes	Yes	Select-range	Provisioning and Release of Resources
[14]	No	Yes	Not restricted	Not applied
[12], [11]	No	Yes	Not restricted	Provisioning of Resources
[15]	No	No	Not restricted	Provisioning of Resources
[13]	No	Yes	Not restricted	Provisioning of Resources
[16]	No	No	Not restricted	Provisioning of Resources
[20]	Yes	Yes	Select-range	Provisioning and Release of Resources
[17]	No	Yes	Not restricted	Provisioning of Resources

Therefore, to better understanding of proposed solution, the requests were classified between three types, according to level of complexity: (i) **Type 1 Requests** represent the select-range and/or select-aggregation requests. Select-range are the database access requests that will return only tuples that are in a given range of a table. An index can be used to select the tuples. The range is used when a column, key or not, is compared with a constant using: =, <>, >, > =, <, <=, IS NULL, <=>, BETWEEN or IN; (ii) **Type 2 Requests** represent the database access requests that uses one or more of the following operators: cross join, inner join, left outer join, right outer join or full outer join. Finally, (iii) **Type 3 Requests** represent the database access requests that use aggregation, joins, union, grouping and/or nesting operators. They can be UNION, INTERSECTION, EXCEPT, ANY, IN, UNIQUE, EXISTS, NOT EXISTS, GROUP BY, HAVING, ORDER BY or FETCH WITH.

B. Metadata and Performance

It is worth noting that before the effective execution of a request, it is replicated to a metadata server. The metadata has main objective extract, process and store information about the request that will be useful to its execution. Furthermore, the metadata monitors the real-time performance of each slave node with the aim to make estimates query execution. The following are presented main information of metadata:

Request Costs: To estimate the cost of a request, in this work was used the EXPLAIN command that shows query plan chosen by the DBMS optimizer. The query plan or query execution plan is the sequence of operations DBMS performs to run a request. The values obtained does not represent the correct estimated cost

if the query is too complex, but it serves as a basis for estimating the request performance. The EXPLAIN command returns the variables: cost, rows and width. The cost estimates are measured in units of disk I/O. An operator that reads a single block of 8.192 bytes (8K) from the disk has a cost of one unit. CPU time is also measured in disk I/O units, but usually as a fraction. For example, the amount of CPU time required to process a single tuple is assumed to be 1/100th (0,01) of a single disk I/O. Finally, the rows variable corresponds the number of tuples to be returned of a request and the width variable corresponds the quantity of bytes of each returned tuple. Therefore, the total cost is the sum of the quantity of disk pages to access the data plus the quantity of returned rows times 0,01, i.e. cost=disk_pages +rows * 0,01.

Request Types: As defined previously, the requests that will executed are classified between three types, according to complexity level. The result of classification are used to trace a request-profile that will used by other requests in search of similar characteristics. Therefore, the EXPLAIN command of DBMS can too be used to obtain these information.

Probability of SRT Violation: Based on the requests of the similar characteristics that executed on the provider, it is calculated the probability of SRT violation. Let PV_{Ri} be the percentage of times that the response time of similar requests was bigger than SRT. If PV_{Ri} exceeds 50%, the query plan will take on a pessimistic approach, which consists to use more computational cost to decrease the probability of SRT violation. If PV_{Ri} does not exceeds 50%, the query plan of R_i request will take on an optimistic approach, which consists to use enough computational cost to execute R_i.

Performance Monitoring: To get the current performance of a slave node in this work were used the variables util and iowait of mpstat and iostat tools. They are very important to identify problems of CPU and device saturation and percentage of CPU time during which I/O requests were issued to the device (bandwidth utilization for the device). In metadata these values of each slave node are updated and stored in the metadata at regular intervals. Finally, whether this percentage is above 80%, the slave node is unavailable for executing requests, because there is too much risk of not meet the expectations of query response, else, on multiprocessor systems is used mpstat tool and through the iowait is checked the each CPU core availability. Case all CPU cores is above 70%, the slave node is unavailable, else, case at least one core is below 70%, the slave node is available to execute requests. In uniprocessor systems, it is checked only global iowait, not being necessary the use of mpstat tool.

C. *Query Processing: FlowChart*

In summary, Figure 1 shows the flowchart of possible execution plans to execute a request. For **Type 1 Requests**, the requests are partitioned in the initial provisioning and its subqueries are distributed in order according to the current performance of each slave node in order to have an execution plan that ensures the SRT. For this, it will be used the metadata variables (statistical data). Therefore, during the execution of each partition, the monitoring checks the elapsed time and estimates the probability of SRT violation.

For **Type 2 Requests**, they are initially executed the partitioning of request according to its simple nested loops and if exists, its predicates. Then, each subquery is executed according to the **Type 1 Requests**. After all process, the result is unified in accordance with its joins.

For **Type 3 Requests**, they can be executed using a pessimistic or optimistic approach. The pessimistic approach is used when the PV_{Ri} of similar requests is greater than 50% and the optimistic approach when the PV_{Ri} is less than or equal to 50%.

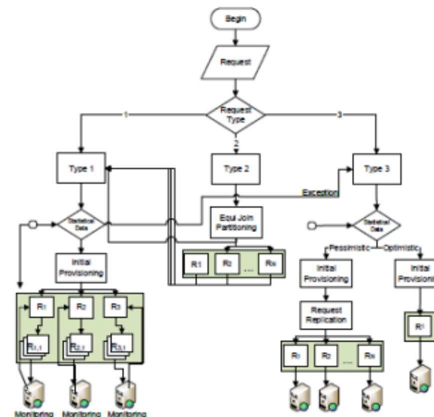


Figure. 1. Flowchart of query processing for each type of request.

The adaptive query processing algorithm (AQP Algorithm) is shown below. For each type of request is used a strategy of partitioning and execution. After its execution, a result of request is presented to customer and for the provider is presented request information, such as SRT violation, elapsed time of request etc. The information of SRT violation is important for the provider to understand the reasons of the violation and to make decisions to reduce the problem.

AQP ALGORITHM (**R**, **TR**, **ET**): RETURN **RESULT**

```
– ET; //Elapsed Time = RSRT - ET.
– R; //Request
– TR; //Type of Request
– METADATA; //Metadata Class
– SLAVE_NODE[0..i]; //Available Slave Nodes
1.  BEGIN
2.    SWITCH(TR)
3.    CASE 1: //Type 1 Request
4.          IF (R.hasPredicate("WHERE T.pk = <<value>>;"))
    //Exception
5.            AQP(R,3,ET);
6.          ELSE
7.            Partition[0..i] = METADATA.getSelectedSlaveNode(R,
    SLAVE_NODE[0..i]);
8.            FOR EACH Partition DO
9.                RESULT += DQM(Partition, ET, 1,
    SLAVE_NODE[j]);
10.           ENDFOR
11.           RETURN RESULT;
12.         ENDIF
13.         BREAK;
14.    CASE 2: //Type 2 Request
15.         Partition[0..i] = PartitionEquiJoin(R);
16.         FOR EACH Partition DO
```

```
17.             SubResult [0..i] = AQP (Partition,1,ET);
18.         ENDFOR
19.         RETURN JOIN(SubResult);
20.         BREAK;
21.     CASE 3: //Type 3 Request
22.
    SelectedSlaveNodes[0..i]=METADATA.getSelectedSlaveNodes(R,
    SLAVE_NODE);//all nodes > ET
23.         IF (METADATA.getProbability(R) == OPTIMISTIC)
24.             RESULT = DQM(R,ET,3,SelectedSlaveNodes[i]);
25.         ELSE //All slave nodes satisfies ET
26.             RESULT = DQM(R,ET,3,SelectedSlaveNodes[0..i/2]);
27.         ENDIF
28.         RETURN RESULT;
29.         BREAK;
30.     ENDSWITCH
31.     IF(ET > RSRT)
32.         METADATA.setViolation(TRUE);
33.     ENDIF
34. END
```

The monitoring algorithm verifies, periodically, the possibility of a query to be executed before a SRT. Therefore, the DQM (Dynamic Query Monitoring) algorithm reevaluates each subquery at runtime and checks the possibility of SRT violation, whether it is low, and the query continues its execution; otherwise, the query will be re-optimized in AQP algorithm.

The monitoring will check the request execution progress. Whether the performance of slave node decreases, the system can try recovering and meeting the recommended SRT or if the performance of slave node increases, the system can use that to its advantage. Therefore, monitoring is adaptive with non-regular intervals, because the framework uses a strategy is based on following variables: CPU, memory and processing and reading percentage in DBMS of each slave node used by request. Thus, this work considers that slave nodes can have different performance.

The challenge of monitoring algorithm is to monitor in the best time. It should not be so frequent, since original queries would be partitioned into many subqueries. Thus, the overload added can prejudice more than help. Moreover, it should not be infrequent, because if that happens, it may be difficult to make corrections in a timely manner and avoid possible penalties.

DQM uses historical data of similar requests to establish the most efficient number of partitions for monitoring. Thus, the algorithm checks the request selectivity and the current performance of the first slave node in the initial provisioning. When there are no statistical data, by default, if the request selectivity is less than 10.000 tuples, the component will fragment the request within 2 partitions. If it is between 10.000 and 100.000 tuples, the component will fragment the request up to 4 partitions. If the selectivity is greater than 100.000 tuples, the framework will fragment the request up to 8 partitions.

When there are statistical data, the number of partitions and the SRT used in the execution of similar requests are checked in metadata. Thus, the number of partitions for monitoring is chosen based on the similarity of request (selectivity) and SRT. It is important to note that the operations will be realized in the metadata and will be available now that is required by the request.

The summary of DQM component algorithm is shown below. As presented, Type 1 and 3 requests use different strategies. For **Type 1 Requests**, the DQM uses monitoring and adaptive query processing and for **Type 3 Requests**, it does not use adaptive query processing, it uses greedy algorithm in optimistic approach and the fastest execution in set of slave nodes in pessimistic approach. For the better understanding, the next section we present samples/examples of the scheduling, partitioning and monitoring algorithms.

```
DQM ALGORITHM (R, ET, TR, SLAVENODES): RETURN RESULT
– R; //Request.
– ET; //Elapsed Time: RSRT - ET
– TR; //Type of Request.
– SLAVENODES; //Slave Node to execute R.
1.  BEGIN
2.      SWITCH(TR)
3.      CASE 1: //Type 1 Request
4.          Partition[0..i] = Metadata.Partitioning(R);
5.          FOR EACH Partition DO
6.
    IF((RESULT+=EXECUTE(Partition,SLAVENODES[0])).getElaps
    edTime()>T2R)
7.              AQP (MERGE(Partition[j..i]),1, ET);
8.          ENDIF
9.          ENDFOR
10.         BREAK;
11.     CASE 3: //Type 3 Request
12.         //optimistic approach: SLAVENODES.getLength() returns 1.
13.         RESULT=EXECUTE(R,SLAVENODES[0..i]);
14.         BREAK;
15.     ENDSWITCH
16. RETURN RESULT;
17. END
```

D. *Query/Subquery Scheduler*

To scheduler of the query is responsible for distributing the partitions of a request to each slave node available based on its performance. To do this, Let $T2R_{SN}$ the *Tuple Read Rate*, the estimated time in seconds for a slave node to process a quantity of tuples:

$$T2R_{SN} = \frac{rows * 1000}{cost * Svctm} \quad (1)$$

where $rows$ corresponds the number of tuples to be returned of a request, $cost$ is estimated in units of disk I/O and $Svctm$ the average service time (in milliseconds) for I/O requests that were issued to the device of a slave node. This last parameter can be obtain through iostat tool. To better understanding, consider the R request with SRT received by a cloud provider:

```
Select * // ← R
From Table T;
```

Consider that SRT is 100 seconds and through the Explain command we have the cost = 368 and rows = 12.000. Moreover, consider that SN1 is an available slave node and it has Svctm = 13 milliseconds. Thus, $T2R_{SN1}$ of SN1 presents read rate of 250 tuples/seconds. Thus, SN1 ensures the SRT because it was estimated that SN1 in 100 seconds could process 25.000 tuples.

It is worth noting that equation do not consider CPU overhead as well as the use of DBMS cache. However, it presents an estimate used only in the initial provisioning. Thus,

at query processing, the $T2R_{SN}$ is calculated by dividing the number of rows retrieved (RT) by the time to retrieve them (TRT):

$$T2R_{SN} = \frac{RT}{TRT} \quad (2)$$

For complex queries, the strategy is similar to select-range queries. However, the rows variable is obtained by sum the quantity of tuples accessed by each query execution plan operator. Even if more than one operator uses these tuples and/or if these tuples are not part of the result. As well as select-range queries, this work considers that all access to a tuple block (on disk or temporary data pagination) is a cost I/O.

It is worth noting that this estimate does not consider the CPU overhead. However, the overhead of temporary data pagination is considered, since it does not distinguish the repetition of tuples during each step of the query execution plan.

Therefore, if we have the current speed of tuples read rate per second of a slave node, it is possible to partition a request in accordance with the estimated time to execute the request on each node. For not violate the SRT, the sum of the times for each partition to execute a subquery, according to the times of each slave node (SN), it has to be less than the SRT:

$$SRT_{Ri} \geq T2R_{SN1} + T2R_{SN2} + \cdots + T2R_{SNk} \quad (3)$$

In this work, the partitioning strategies depends on the type of request and we consider that the all tables are clustered by primary key.

For example, assume that a cloud provider receives the following select-range request R with SRT:

```
SELECT * // ← R
FROM table T
WHERE T.pk >= 1000 and T.pk < 5000;
such that pk is the primary key of table T.
```

Considering that primary key values of T are sequential, without gaps between values, then we can extract rows = 4.000 tuples. Besides, consider that SRT is 100 seconds and that initial provisioning is a single slave node (SN1) such that the current moment $T2R_{SN1}$ = 20 tuples/sec.

Consequently, the initial provisioning using only SN1 will bring a penalty to be paid by the provider because it was estimated that SN1 in 100 seconds will process in 2.000 tuples. In this case, it is necessary to allocate a new slave node (SN2) to help. Assume that $T2R_{SN2}$ = 10 tuples/sec then only 1.000 tuples can be processed in 100 seconds. Then, a new slave node (SN3) is required to process the request. Then, consider $T2R_{SN3}$ = 10 tuples/sec.

At this point, it is possible that three slave nodes are sufficient to process R and ensure the SRT. R is rewritten in three subqueries: R_1, R_2 and R_3, the first one is executed in SN1, the second one in SN2 and the third one in SN3, respectively. Note that in this case a virtual partitioning is used (i.e. we partition using the predicate of the primary key) to divide R in R_1, R_2 and R_3.

```
SELECT * // ← R1
FROM table T
WHERE T.pk >= 1000 and T.pk < 3000;
```

```
SELECT * // ← R2
FROM table T
WHERE T.pk >= 3000 and T.pk < 4000;
```

```
SELECT * // ← R3
FROM table T
WHERE T.pk >= 4000 and T.pk < 5000;
```

Using only three slave nodes do not guarantee that the quality defined in SRT will be met, because the cloud environment is unstable and the performance of nodes can change during the queries execution. Therefore, a proactive approach based on statistical data in metadata indispensable use. For this, the query are partitioned in such a way that the performance of the nodes can be monitored at a frequency that allows other nodes to be added when necessary in order to ensure the SRT.

An important issue is the monitoring frequency. If too frequent, the original queries would have to be partitioned into many subqueries. Thus, the overload added could prejudice more than help. If monitoring is infrequent, it may be difficult to make corrections in a timely manner and avoid possible penalties.

The partitioning process uses historical data about the request containing information about how long it was necessary to process similar requests (same type of request), including the number of partitions used. From this information, it is possible to monitor efficiently the request execution.

Consider for example, in similar requests, 2 partitions were used for each partition of the initial provisioning. Then, R_1 is partitioned in two requests:

```
SELECT * // ← R1,1
FROM table T
WHERE T.pk >= 1000 and T.pk < 2000;
```

```
SELECT * // ← R1,2
FROM table T
WHERE T.pk >= 2000 and T.pk < 3000;
```

When $R_{1,1}$ is done, it have the first opportunity to monitor the query execution performance in a non-intrusive way. Consider that 70 seconds were spent to execute $R_{1,1}$. This means that the performance $T2R_{SN1}$ was below of predicted, which leads to a completion time with the expected processing of the next subquery of 140 seconds. However, this value is above the SRT. Thus, it starts a revision of the initial provisioning for that SRT can be satisfied. Before reviewing, the remaining partitions will be merged in a single query.

In this case, a solution relocates remain subquery to another slave node. Suppose a new slave node (SN4) is such that $T2R_{SN}$ = 30. Thus, all the 1.000 remaining tuples can be read by SN4 in 30 seconds in the best-case scenario, and that does not lead to a violation of SRT. To monitor the request execution, it is again partitioned into two, each of the following way:

```
SELECT * // ← R1,2,1
FROM table T
WHERE T.pk >= 2000 and T.pk < 2500;
```

```
SELECT * // ← R1,2,2
FROM table T
WHERE T.pk >= 2500 and T.pk < 3000;
```

Consider that performance is stable and it is able to finish their workload on schedule. Thus, the same strategy can be applied in the processing of R_2 in SN2 and R_3 in SN3. This partitioning method using the primary key as the partitioning attribute is the same for similar select-range requests and similar requests with aggregation.

For requests with joins (**Type 2 Requests**), it rewrites the query separating all tables of FROM clause. Consider that a cloud provider receives the following trivial select-join request R with SRT:

```
SELECT * // ← R
FROM table T1, table T2
WHERE T1.fk = T2.pk;
such that T1.fk the foreign key referenced by the primary key T2.pk.
```

The request R is rewritten in two subqueries, R_1 and R_2:

```
SELECT * // ← R1
FROM table T1;

SELECT * // ← R2
FROM table T2;
```

In this case, R_1 and R_2 will be executed utilizing strategies of **Type 1 Requests**. Thus, it is used the partitioning methodology described for **Type 1 Requests**. As well as the monitoring and provisioning of slave nodes to execute the rewritten query is made the same way. Finally, after the execution of all partitions the slave node that executed R_1 makes the join to present the result.

For complex requests and others not shown here (**Type 3 Requests**), it adopts the strategy of seeking the set of available slave node with $T2R$ enough to process the request that ensures the SRT. This strategy are adopted before due to the highly complexity of estimates costs and making partitioning. Therefore, this type of request does not use monitoring nor adaptive partitioning during query execution. Consider the following a complex request with SRT is 100 seconds and rows = 200.000 tuples.

In the optimistic approach, the greedy strategy is adopted, in which only one slave node executes the request and it is expected that it ensure the SRT. Now consider three slave nodes, SN1, SN2 and SN3, with $T2R_{SN1}$ = 4.000 tuples/sec, $T2R_{SN2}$ = 2.000 tuples/sec, $T2R_{SN}$ = 1.000 tuples/sec, respectively. In this case, the algorithm using greedy strategy chooses SN1 because it was the first and enough in such a way to execute R within the SRT. In pessimistic approach, the algorithm strategy is to choose half the number of slave nodes available with the highest $T2R$ to execute request R. Consider four available slave nodes, SN1, SN2, SN3, SN4, with $T2R_{SN}$ = 4.000 tuples/sec, $T2R_{SN2}$ = 2.000 tuples/sec, $T2R_{SN3}$ = 1.000 tuples/sec and $T2R_{SN4}$ = 1.500 tuples/sec, respectively. Then, the algorithm replicates the request R for SN1 and SN2, in such a way that least one can ensure the SRT.

In the worst-case scenario, if there is no slave nodes that meets the SRT, the closest node to meet the SRT in terms of $T2R$ is selected. Monitoring the slave node to process this type of request is made after its processing, when it is checked violation or not of SRT and metadata updates its information.

IV. Experimental Evaluation

A. Experimental Environment

The strategies presented was implemented in using the Java language and concurrent programming with threads and API based on OpenMP (Open Multi-Processing) [22]. It was deployed in the Amazon EC2 cloud infrastructure using small instances. Due to the limitations of Amazon, it was used 20 VMs, each one with an Intel Xeon Processor with turbo up to 3.3GHz, 1.7 GB of main memory and 160 GB of disk storage.

It was created an AMI (Amazon Machine Image) of a VM with the database. This image allows startup new VMs quickly. The Amazon EBS (Elastic Block Store) was used to storage the AMI. Each VM runs the Ubuntu 12.04 operating system and PostgreSQL 9.3 DBMS. This work focuses on OLAP applications with very large and complex database. Thus, the TPC-DS like benchmark was used to generate a database of approximately 13 GB, fully replicated in all VMs. Therefore, the database generated represents the customer data.

B. Methodology

The experiments aim at showing the efficiency of queries processing strategies proposed in this work. This way, it will check the ability to avoid penalties associated with SRT violation and the elasticity of the algorithm in according to the number of VMs allocated when processing queries. For **Type 1** and **Type 2 Requests**, the experiments consisted to stress the system using 10 workloads and each workload having 10 queries of the same type. For **Type 3 Requests**, as the strategy is predictive and queries are complex, 5 workloads were used, each workload having 5 queries of the same type. Finally, the experiments were performed using 10 workloads and each workload having 10 queries of several types of requests.

The minimum amount of required machines is a complex task. Therefore, previous tests were performed using a fixed number of VMs. Thus, the minimum number of machines was found for the workload of the experiments. However, if new workloads arrive to the system, it will be necessary to perform extensive experiments again to obtain a new configuration of service provider. The arrival time of the queries workloads was disposed uniformly varied distribution (non-uniform distribution): each workload arriving at a random time intervals between 10 and 60 seconds. This distribution is closer to real environments, since the unpredictability of workloads arriving to the system and performance variation are characteristics of cloud environments. Moreover, it was used different values of SRT, from the most restricted to the most relaxed.

Seeking more accurate results for each type of request, experiments were repeated 10 times. Finally, to eliminate any possible interference between successive experiments, in particular, effects of other queries already executed, the OS cache was deleted and the DBMS has been restarted before executing the queries workloads again.

For each experiment, the number of virtual machines used are observed in accordance with time. To calculate the computational cost it was enough to observe the number of virtual machines used by each query. Finally, the query runtime is measured according to the strategies described in previous Section.

C. Results

For **Type 1 Requests**, the Figure 2 shows experiments of select-range queries with the arrival of workloads following the non-uniform distribution. The graphs present the number of VMs used by time in seconds and the SRTs used were 80, 100 and 120 seconds. The queries predicate may be on a non-key attribute or on a key attribute.

It is important to emphasize at this point that when the attribute is not a primary key, our strategy scans all tuples of the table, and i.e. all tuples are checked to verify whether they satisfy the predicate. Thus, this type of queries requires more processing time than the select-range that have a predicate on key attribute. Consequently, this causes the increase of VM computational cost, since the response time is higher. According to the results, it can see the increase and decrease of the workloads on the system and the elasticity on the number of virtual machines allocated to execute the queries. When the SRT is more restricted, the computational cost is higher or equal to the computational cost of the most relaxed SRT, this happens to avoid penalties. Moreover, the computational cost is higher when the workloads arrive at random times (non-uniform distribution) if compared to uniform distribution. We believe that the system may not recover quickly when there is an unexpected overload resource, and seeking quick reaction to execute the queries, the algorithm allocates more VMs to execute the workload in SRT time. Consequently, the computational cost increases.

For **Type 2 Requests**, experiments were realized in queries with or without SQL predicate. Then, the primary difference between these types (**Type 1 and Type 2 Requests**) is the query partitioning and merge of their results. Figure 3 shows the experiments with the arrival of workloads following the non-uniform distribution. As in previous experiments, the SRT was varied, the most restricted SRT was 130 seconds and the most relaxed SRT was 180 seconds. The partitioning time of queries was not considered because the low complexity of queries used in the experiments. However, it was observed that the merging of the results cause a higher time to execute queries, approximately 10% more than the select-range requests. Finally, it is important to observe that in the most restricted SRT, the ninth workload reached the limit of the infrastructure service provider. For **Type 3 Requests**, the experiments were realized with complex queries obtained from the TPC-DS. As shown in previous sessions, the following graphs show the number of VMs allocated by time in seconds and the SRT was varied, the most restricted SRT was 800 seconds and the most relaxed SRT was 1200 seconds. However, due to the complexity of these queries and the limit of VMs available in service provider, it was used only 5 workloads and each workload having 5 complex queries. According to proposed strategy of this work, the experiments stressed the system searching a VM that could execute successfully a query in SRT time (optimistic approach) or executing a query over a set of VMs that one VM could execute successfully the query in SRT time (pessimistic approach). Therefore, it is not used monitoring nor adaptive partitioning during query execution. Figure 4 shows the results of experiments following the non-uniform distribution of workloads. We can observe that due to the strategy used in this work a large number of VMs are used since the first query workload. In addition, in accordance to the strategy presented, the algorithm chooses through the metadata the optimistic or pessimistic strategy for executing a query and after its execution the metadata are updated. However, we believe that the decrease in the use of virtual machines after the third workload happened due to the algorithm starts to use more often the optimistic approach. Consequently, the queries were being executed successfully. Moreover, it can observe that due to the complexity and selectivity of the queries, there is a greater overhead for the ending its results.

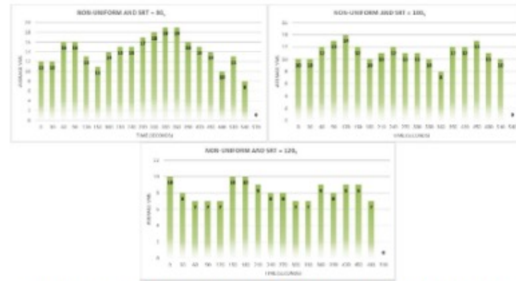


Figure. 2. Type 1 Requests (Select-Range): average virtual machines used for workloads randomly arriving between 10 and 60 seconds for the SRTs: 80, 100 and 120 seconds.

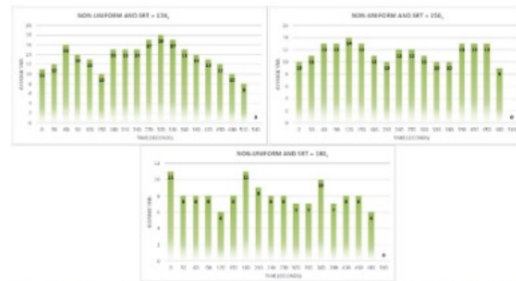


Figure. 3. Type 2 Requests: average virtual machines used with workloads randomly arriving between 10 and 60 seconds for the SRTs: 130, 150 and 180 seconds.

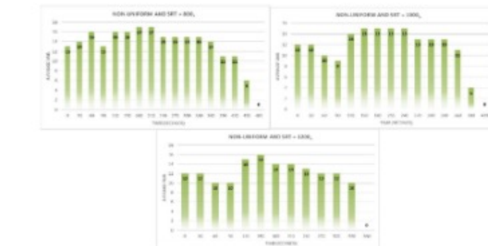


Figure. 4. Type 3 Requests: average virtual machines used with workloads randomly arriving between 10 and 60 seconds for the SRTs: 800, 100 and 1200 seconds.

Finally, experiments were realized using **all requests types** over the same queries workload. According to the strategies proposed in this paper, it was obtained similar results to previous ones. The main overhead is of the algorithm having to classify each query to be executed. After classifying the query, the query is executed according to the already mentioned strategies. Figure 5 shows the experiments with the arrival of workloads non-uniform distribution. The graph show the number of VMs used by the time in seconds. However, unlike previous experiments, each query after classification has a different SRT, according to their type of request. For several moments, it can see the limit of the provider's infrastructure is reached; however, it has not been exceeded. Thus, it can see the increase and decrease in workloads due to elasticity in the number of allocated virtual machines to execute all queries. It is important to observe that no penalty occurred with all queries. Finally, the results of all experiments shown that the proposed solution reacts to the resources variation of the environment and to different sizes of workloads. The strategies ensured that the SRT was satisfied in a non-intrusive and automatic way.

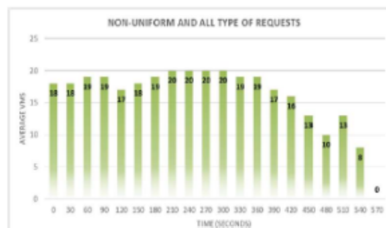


Figure. 5. All Type Requests: average virtual machines used with workloads randomly arriving between 10 and 60 seconds.

V. CONCLUSIONS

In this work, it was presented partitioning, monitoring and provisioning strategies for adaptive processing of different types of queries (database access requests) in cloud environment. The strategies were implemented in a framework and the experiments were evaluated in Amazon EC2 cloud infrastructure. This work focuses on OLAP applications because this kind of environment the adaptive processing produces positive effects at query runtime.

Given the increase and decrease of the workloads, it can see the elasticity in the number of virtual machines allocated by the methods proposed to execute queries. Furthermore, results show that the solution reacts to the resources variation of the environment and to different sizes of workloads. A solution ensures that the SRT is satisfied in a non-intrusive and automatic way. Finally, our proposal was effective to avoid the penalties in the execution of queries and the SRT was satisfied in all experiments without incurring penalties. As future work, we intend to develop adaptive strategies for more types of queries. Moreover, we intend to improve the cost model involving others SLA parameters, such as resiliency, throughput and efficiency, since they are important measures to evaluate the performance of services in cloud infrastructures.

ACKNOWLEDGMENT

This work was supported by Federal Institute of Rio Grande do Norte, University of Minho and Institute for Systems and Computer Engineering, Technology and Science – INESC TEC.

REFERENCES

[1] CSMIC, "Service Measurement Index Introducing the Service Measurement Index (SMI)," no. July, pp. 1–8, 2014.
[2] S. K. Garg, S. Versteeg, and R. Buyya, "A framework for ranking of cloud computing services," *Futur. Gener. Comput. Syst.*, vol. 29, no. 4, pp. 1012–1023, 2013.
[3] V. C. Emeakaroha, M. a S. Netto, R. N. Calheiros, I. Brandic, R. Buyya, and C. a F. De Rose, "Towards autonomic detection of SLA violations in Cloud infrastructures," *Futur. Gener. Comput. Syst.*, vol. 28, no. 7, pp. 1017–1029, 2012.
[4] W. Iqbal, M. Dailey, and D. Carrera, "SLA-Driven Adaptive Resource Management for Web Applications on a Heterogeneous Compute Cloud," in *1st International Conference on Cloud Computing (CloudCom '09)*, 2009, pp. 243–253.
[5] "AWS EC2 Service Level Agreement," 2015. [Online]. Available: http://aws.amazon.com/ec2-sla. [Accessed: 15-Jun-2015].
[6] D. Sanderson, *Programming Google App Engine:*, 2nd ed. O'Reilly Media | Google Press, 2012.
[7] Transaction Processing Performance Council, "Tpc Benchmark™ Ds," 2012.
[8] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguadé, "Dynamic CPU Provisioning for Self-managed Secure Web Applications in SMP Hosting Platforms," *Comput. Netw.*, vol. 52, no. 7, pp. 1390–1409, 2008.
[9] Amazon Web Services, "Auto Scaling: Developer Guide," 2015.
[10] J. Rogers, O. Papaemmanouil, and U. Cetintemel, "A generic auto-provisioning framework for cloud databases," in *IEEE 26th International Conference on Data Engineering (ICDE'10)*, 2010, pp. 63–68.
[11] H. Kllapi, E. Sitaridi, M. M. Tsangaris, and Y. Ioannidis, "Schedule Optimization for Data Processing Flows on the Cloud," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011, pp. 289–300.
[12] J. Zhao, X. Hu, and X. Meng, "ESQP : An Efficient SQL Query Processing for Cloud Data Management," in *2nd International Workshop on Cloud Data Management (CloudDB'10)*, 2010, pp. 1–8.
[13] Y. Chi, H. J. Moon, H. Hacıgümü, J. Tatemura, H. Hacigümüş, and J. Tatemura, "SLA-Tree: A Framework for Efficiently Supporting SLA-based Decisions in Cloud Computing," in *14th International Conference on Extending Database Technology (EDBT/ICDT '11)*, 2011, p. 129.
[14] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan, "Workload-aware Database Monitoring and Consolidation," in *ACM SIGMOD International Conference on Management of Data*, 2011, pp. 313–324.
[15] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A Cost-Aware Elasticity Provisioning System for the Cloud," in *31st International Conference on Distributed Computing Systems (ICDCS'11)*, 2011, pp. 559–570.
[16] J. Cerviño, E. Kalyvianaki, J. Salvachúa, and P. Pietzuch, "Adaptive provisioning of stream processing systems in the cloud," in *IEEE 28th International Conference on Data Engineering Workshops (ICDEW)*, 2012, pp. 295–301.
[17] R. Mian, P. Martin, and J. L. Vazquez-Poletti, "Provisioning data analytic workloads in a cloud," *Futur. Gener. Comput. Syst.*, vol. 29, no. 6, pp. 1452–1458, 2013.
[18] Y. Vigfusson, A. Silberstein, R. Fonseca, B. F. Cooper, and R. Fonseca, "Adaptively Parallelizing Distributed Range Queries," in *VLDB Endowment*, 2009, vol. 2, no. 1, pp. 682–693.
[19] D. Alves, P. Bizarro, and P. Marques, "Deadline Queries: Leveraging the Cloud to Produce On-Time Results," in *IEEE 4th International Conference on Cloud Computing*, 2011, pp. 171–178.
[20] T. L. Coelho Da Silva, M. a. Nascimento, J. A. F. De Macêdo, F. R. C. Sousa, and J. C. Machado, "Non-intrusive elastic query processing in the cloud," *Comput. Sci. Technol.*, vol. 28, no. 6, pp. 932–947, 2013.
[21] Í. Goiri, F. Julià, J. O. Fitó, M. Macías, and J. Guitart, "Supporting CPU-based guarantees in cloud SLAs via resource-level QoS metrics," *Futur. Gener. Comput. Syst.*, vol. 28, no. 8, pp. 1295–1302, 2012.
[22] OpenMP.org, "About the OpenMP ARB," 2013. [Online]. Available: http://openmp.org/wp/about-openmp/.