PhD Thesis submitted to the University of Coimbra
(Tese de Doutoramento submetida à Universidade de Coimbra)

Quality of Experience in Database Systems
(Qualidade de Experiência em Sistemas de Bases de Dados)

Rogério Luís de Carvalho Costa

Advisor: Professor Pedro Nuno San-Bento Furtado

Table of Contents

Table of Figures

Abstract

This work aims at providing mechanisms to increase users' satisfaction when using database systems. We express users' satisfaction in terms of Quality of Experience (QoE). Therefore, our proposals aim to increase the degree of QoE a database system provides.

Traditional database systems execute operations immediately upon submission and, since they do not allow users to express execution-related constraints, they do not evaluate whereas those constraints are covered, and they do not take corrective action when necessary.

Our proposal for QoE makes the database system take into consideration users' expectations on deciding how or when to execute operations. This is based on a set of Data Access Requirements (DAR) that users can associate to database operations and the QoE-prepared system considers those when processing the operations.

Since the objective of the QoE-oriented database system is to provide user satisfaction, the analysis of its performance must consider success rate measures on achieving the user specified constraints. We have defined Key Performance Indicators based on such measures and used those as part of our comparison of approaches.

Our proposed Data Access Requirements (DARs) include execution deadlines, execution start and end times, data availability, data freshness, execution priority, disconnected execution and job repetition. Some of those, such as execution deadlines, are useful in any data processing architecture - centralized, parallel or distributed – whereas DARs such as availability are especially designed for parallel and distributed contexts.

For the proposed approach to work on any of those architectures we needed to develop a set of features that includes runtime estimations, requirements-based task scheduling and future jobs monitor and scheduling.

Besides that, we also developed some other features required for parallel and distributed QoE-oriented database systems. Those include an election-based global scheduler, capacity to evaluate data availability degree and capacity to decide on data replication. Another important aspect in parallel and distributed settings was the development of reputation strategies. These allow the system to constantly have updated quantitative information on the degree of QoE expectations fulfillment by nodes or sites and, based on these, to adapt in order to maintain high QoE capabilities. Requirements fulfillment rates and runtime estimations are the bases of proposed reputation algorithms, which support better scheduling decisions.

We show experimentally, using benchmark scenarios, that the proposed QoE-oriented database system is able to satisfy the user-defined execution-related constraints in both centralized, parallel and distributed cases. In order to do this we have created a prototype and tested the most important concepts proposed in this thesis. The approaches were compared with best effort (no QoE) counterparts and, when relevant, with scheduling approaches such as round-robin or on-demand.

Resumo da Tese em Português

Dado que a tese foi escrita em Inglês (língua franca desta área de conhecimento) e foi desenvolvida na Universidade de Coimbra em Portugal, faz-se nesta secção um breve resumo em Português do conteúdo da tese. A secção começa com o sumário da tese e depois apresenta uma breve descrição do conteúdo de cada capítulo.

Sumário

Este trabalho visa fornecer mecanismos para aumentar a satisfação dos utilizadores quando utilizam sistemas de bases de dados. Consideramos *satisfação dos usuários* em termos de Qualidade de Experiência (QoE). Desta forma, as nossas propostas visam aumentar o nível de QoE fornecido por sistemas de bases de dados.

Sistemas de bases de dados tradicionais executam as operações imediatamente após a submissão e, como eles não permitem que os utilizadores expressem restrições relacionadas com a execução de comandos, não podem avaliar se essas restrições são atendidas e não podem executar acções corretivas quando necessário.

A nossa proposta de QoE faz com que os sistemas de bases de dados tenham em consideração as expectativas dos utilizadores ao decidir como ou quando executar os comandos que lhes são submetidos. Esta estratégia é baseada em Requisitos de Acesso a Dados (*Data Access Requirements* - DAR) que os utilizadores podem associar a operações de bases de dados e que os sistemas preparados para fornecer QoE tomam em consideração quando processam tais operações.

Como o objetivo de sistemas de bases de dados orientados a QoE é prover satisfação aos utilizadores, a análise do seu desempenho deve considerar indicadores específicos sobre a satisfação de restrições definidas por estes. Nós definimos Indicadores Chave de Desempenho baseados nessas medidas e utilizamo-los como parte de comparação de abordagens.

Os tipos de Requisitos de Acesso a Dados (DARs) propostos incluem prazos de execução, hora de início e término de execução de comandos, actualidade dos dados utilizados, prioridade de execução de comandos, execução desconectada e repetição de trabalhos. Alguns destes, como os prazos de execução, são úteis em quaisquer arquiteturas de bases de dados – centralizada, paralela ou distribuída – enquanto outros DARs, como o de disponibilidade dos dados, são especialmente preparados para os contextos paralelo e distribuído.

Para que as abordagens funcionem em quaisquer destas arquiteturas, precisámos desenvolver um conjunto de funcionalidades que incluem estimativas de tempo de execução, escalonamento de tarefas baseado em requerimentos e monitoramento e escalonamento de trabalhos futuros.

Além disso, também desenvolvemos algumas funcionalidades que são requeridas para sistemas paralelos e distribuídos de bases de dados orientados a QoE. Estas incluem um escalonador global baseado em eleições, capacidade de avaliar o nível de disponibilidade de dados e capacidade de decidir sobre replicação de dados. Outro aspecto importante nas configurações paralela e distribuída é o desenvolvimento de estratégias baseadas em reputação. Estas permitem que o sistema tenha informações quantitativas constantes sobre o nível de atendimento às expectativas de QoE por parte

de nós ou sítios e, baseado nisso, execute acções para se adaptar no intuito de manter altos níveis de QoE. A taxa de atendimento de requisitos e as estimativas de tempo de execução são a base dos algoritmos de reputação, os quais suportam melhores decisões de escalonamento.

Nós mostramos experimentalmente, utilizando cenários e "benchmarks", que o sistema de bases de dados orientado à QoE é capaz de satisfazer as restrições definidas pelos utilizadores sobre a execução de comandos, tanto no caso centralizado, como nos casos paralelo e distribuído. Para isso, criámos um protótipo e testámos os conceitos mais importantes propostos nessa tese. As abordagens propostas foram comparadas com as correspondentes dos ambientes de "best effort" *melhor esforço* (sem QoE) e, quando relevante, com abordagens de escalonamento como a circular ou a sob demanda.

Introdução

No Capítulo 1 introduzimos os conceitos da tese, incluindo a estratégia utilizada para QoE e a utilização de mecanismos de reputação.

O principal objectivo de sistemas orientados a QoE é satisfazer aos usuários. De facto, considerar as expectativas dos usuários é um dos principais aspectos da QoE [Zapater & Bressan, 2007]. Desta forma, a nossa estratégia para oferecer QoE quando executamos operações de bases de dados é permitir aos utilizadores que exprimam as suas necessidades e fazer com que o sistema as considere quando executa os comandos.

Nesse contexto, propomos que os utilizadores possam especificar Requisitos de Acesso a Dados (*Data Access Requirements - DARs*) em conjunto com os comandos de bases de dados. Tais requisitos colocam restrições sobre a execução de comandos ou objetivos a serem atingidos quando se gere dados armazenados.

Caberá ao sistema avaliar os requisitos e, caso não seja possível atendê-los, avisar o utilizador quanto antes.

No Capítulo 1 também são introduzidos os conceitos de escalonador comunitário e de tarefas e a necessidade de utilização de mecanismos para incrementar os níveis de QoE fornecidos em ambientes de bases de dados paralelas e distribuídas.

Trabalhos Relacionados

Para implementação do mecanismo proposto para aumentar o nível de QoE fornecido pelos sistemas de gestao de bases de dados, foi necessário incorporar diversas funcionalidades. Neste contexto, o Capítulo 2 apresenta trabalhos de diferentes áreas que são relacionados com alguns aspecto estudados nesta tese.

São apresentados trabalhos das seguintes áreas:

- Sistemas orientados a QoE;
- Escalonamento de comandos e colocação de dados em bases de dados paralelas e distribuídas;
- Controlo de admissão e execução de transações sujeitas a restrições;
- Estimativas de tempo de execução de comandos em sistemas de bases de dados.

Definindo Requisitos

No Capítulo 3 apresentamos propostas relativas à especificação de requisitos de acesso a dados e de extensão da linguagem SQL para suportar tais requisitos.

Os requisitos são especificados numa área específica, denominada área de especificação de requisitos, e identificada pela palavra-chave `REQUIREMENTS`.

Os requisitos podem estar associados por uma associação do tipo `E` (separados por vírgulas) ou ainda ter uma associação do tipo `OU`, como na Figura 1. Podem estar associados a comandos de manipulação de dados (Figura 1), a comandos de definição de dados (Figura 2) e a blocos de comandos.

```
SELECT *
FROM SALES
WHERE STATE_ID = 1
REQUIREMENTS
    (DEADLINE 900)
    OR
    (START AFTER '2010/12/15 19:00',
    FINISH BEFORE '2010/12/16 08:00',
    EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS TMP_SALES,
    AVAILABLE DURING 100 PERCENT
        IN PERIOD FROM '2010/12/16' TO '2010/12/17'
```

Figura 1 - Exemplo de Área de Especificação de Requisitos– Comando SELECT

```
CREATE TABLE CUSTOMERS (
 CUSTOMER_ID INTEGER PRIMARY KEY,
 CUSTOMER_NAME VARCHAR(100)
)
REQUIREMENTS MyRequirements
    AVAILABLE DURING 100 PERCENT
        IN PERIOD FROM '2010/12/16' TO '2010/12/17'
```

Figura 2 - Exemplo de Área de Especificação de Requisitos - Comando CREATE TABLE

Blocos de comandos são conjuntos de comandos delimitados por BEGIN BLOCK e END BLOCK, e para os quais é especificado um conjunto de DARs. Os blocos de comandos podem conter clausulas PARALLEL ou SEQUENTIAL, utilizados para identificar se os comandos devem ser executados em sequência ou se podem ser executados em paralelo.

O exemplo da Figura 3 apresenta um bloco de comandos com o seu conjunto de requisitos. Cada comando pode ser executado em paralelo e possui os seus próprios requisitos.

```
BEGIN PARALLEL BLOCK

 SELECT C.ID, C.NAME, C.PHONE
 FROM CUSTOMERS C
 WHERE EXISTS (SELECT 1
               FROM SALES S
               WHERE S.CUSTOMER_ID = C.CUSTOMER_ID
                 AND S.REVENUE > 1000
                 AND S.DATE > '2010/01/01')
 REQUIREMENTS REQ1
   EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS TOP_CUSTOMERS
   AVAILABLE DURING 10 MINUTES AFTER EXECUTION;

 SELECT S.STATE_ID, ST.STATE_NAME, SUM(S.REVENUE) AS SUM_REVENUE
 FROM SALES S
     INNER JOIN STATES ST
     ON S.STATE_ID = ST.STATE_ID
 WHERE S.DATE > '2010/01/01'
 GROUP BY S.STATE_ID, ST.STATE_NAME
 REQUIREMENTS
   EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS REVENUE_PER_STATE
   AVAILABLE DURING 10 MINUTES AFTER EXECUTION;

END BLOCK
REQUIREMENTS REQ_BLOCK
 FRESHNESS OF STATES HIGHER THAN '2010/07/01',
 FINISH BEFORE '14:00',
 REPEAT EVERY FRIDAY IN PERIOD FROM '2010/07/01' TO '2010/08/01'
```

Figura 3 - Exemplo de Blocos de Comandos com Requerimentos

O Capítulo 3 apresenta ainda a definição formal e exemplos de utilização dos vários tipos de requisitos propostos.

Tarefas e Requisitos ao Nível de Tarefas

No capítulo 4, são apresentados os conceitos de trabalho e tarefa. Um trabalho será uma unidade lógica para os quais são definidos, pelo utilizador, um ou mais requisitos. Para a execução de um trabalho será necessário executar uma ou mais tarefas. Cada tarefa poderá ter um ou mais requisitos, que são derivados dos requisitos definidos para os trabalhos.

Nesse contexto, utilizamos dois níveis de escalonadores: o de trabalhos (comunitário) e o de tarefas (relacionado com um serviço de dados específico).

O Capítulo 4 apresenta como são definidas as tarefas e seus requisitos a partir dos trabalhos e dos DARs. Apresenta ainda diversos exemplos de tais definições nos contextos de bases de dados centralizada, paralela e distribuída.

Reputação e Escalonamento Baseado em Eleições

No Capítulo 5 propomos estratégias para escalonamento de trabalhos em ambientes de bases de dados paralelas e distribuídas, com a respectiva alocação de tarefas aos serviços.

As estratégias propostas são baseadas em eleições. Inicialmente, alguns serviços são selecionados como pré-candidatos para executar uma tarefa. Então, os serviços avaliam se podem ou não executar a tarefa em questão, satisfazendo seus requisitos (e ainda satisfazer os requisitos das outras tarefas que já lhes tinham sido atribuídas). Caso possam, tornam-se candidatos e apresentam promessas de tempo para a sua execução. O escalonador comunitário de trabalhos irá, então, alocar tarefas aos candidatos considerando critérios de reputação e de tempo de execução prometido.

As estratégias propostas são utilizadas ainda para identificar réplicas de dados que poderiam ser criadas para incrementar os níveis de QoE fornecidos pelo sistema.

Avaliação e Gestão de Tarefas nos Serviços de Dados

Durante a realização das eleições para atribuição das tarefas aos serviços de dados, cada pré-candidato deverá avaliar se conseguirá atender aos requisitos das tarefas ou não. No Capítulo 6 são apresentados os mecanismos de avaliação utilizados pelos serviços de dados.

São apresentadas, também, as estratégias de escalonamento utilizadas pelos escalonadores de tarefas e os mecanismos de estimação de tempo de execução de consultas.

Medindo a QoE proporcionada por Sistemas de Bases de Dados

Os indicadores de desempenho tradicionais não são capazes de avaliar os níveis de QoE fornecidos por sistemas de bases de dados, uma vez que não tomam em consideração nas suas medidas se os requisitos de utilizador foram cumpridos ou não. No Capítulo 7, apresentamos propostas de quatro indicadores de desempenho (*Acceptance Rate, Commitment Maintenance Rate, Success Rate, QoE-Level*) para sistemas de bases de dados orientados a QoE.

Os indicadores propostos são baseados nas seguintes estatísticas:

- Número de trabalhos com DARs submetidos ao sistema;
- Número de trabalhos com DARs que o sistema aceitou executar;
- Número de trabalhos cujos DARs o sistema satisfez.

Além de serem utilizados para avaliar o QoE, tais indicadores podem ainda ser utilizados para alertar os administradores do sistema quando o nível de QoE fornecido for insatisfatório.

Avaliação Experimental

O Capítulo 8 apresenta resultados obtidos experimentalmente que comprovam a utilidade das propostas da tese. São apresentados resultados de avaliações em três cenários:

- Ambientes de data warehouse globais;
- Ambientes de data warehouse paralelos;
- Ambientes *on-line transaction processing* (OLTP) sobre bases de dados centralizadas.

Além dos estudos relacionados com os três cenários citados acima, são também avaliados aspectos específicos relativos a algumas das propostas dessa tese. São eles:

- Utilização de reputação no escalonamento;
- Mecanismos de filas de tarefas e estimativas de tempo de execução.

Conclusões e Trabalhos Futuros

O Capítulo 9 apresenta as conclusões do trabalho e os possíveis trabalhos futuros.

Acknowledgements

At first, I would like to thank to Professor Pedro Furtado for his advice. During the last years, he continuously encouraged me and was always ready to discuss directions and objectives. His critical comments and observations helped me to keep moving forward with this work.

I thank administrative and technical staff members of DEI, for always being available to help me in administrative issues and on using department's infrastructure.

I would like to thank my parents, Alvaro and Marlene, for their continuous support in my journey and never-ending love. I would like to express my special thanks to my brother and sister, and their families.

Finally, I would like to dedicate this thesis for my wife Renata, who always kept motivating and supporting me, and for my daughter Mariana. I thank them for their love and patient, which were indispensable.

1 Introduction

In this work, we propose how to provide high Quality of Experience (QoE) to database users. QoE mechanisms allow the database system to take into consideration execution-related constraints that are important to users in processing decisions. This way a QoE-oriented database system is able to adjust based on user expectations. This is unavailable in traditional database systems.

The level of QoE a system provides is closely related to the *satisfaction* such system provides to users [Kilkki, 2008; Nokia, 2004]. In order to provide high satisfaction to database users, we intend to make the system behave *as users expect it to do*.

The popularity of the *Quality of Experience* term increased in recent years [Kilkki, 2008]. Nokia (2004) argues that mobile service providers, which do not provide high levels of QoE, are in competitive disadvantage and may lose revenue. Marez & Moor (2007) argue that providing and measuring QoE is central in today's information and communication thechnology. In this work we propose how to incorporate QoE-related mechanisms in database systems.

The key difference between a traditional database system and a QoE-oriented database system is that the former aims at finishing every request as soon as possible, while the later aims at satisfying users' expectations.

For instance, let us consider that a user needs a certain report in 3 minutes, but the database would take 5 minutes to conclude the report's query. In a *best effort* oriented system, the user would wait for 3 minutes for the report and, then, discover that the system would not finish report execution by the time the user needed it to be finished. In a QoE-oriented system, the user would specify that he needs the report to be finished in 3 minutes and the system would immediately inform the user that the report would not be finished by the specified deadline. Then, the user may consider changing the deadline, may decide not to start report execution (reducing the waste of user and processing time) or may schedule report execution for another period.

Users may also want to specify that a certain report must be ready tomorrow at a certain hour or every 28th day of each month and leave the system to take decisions and actions.

Now consider a distributed database composed of several sites and distributed query workloads. The unavailability of even a single site may compromise the whole query workload. A QoE system can take into consideration user specifications that the data should be available either always or in a certain time interval to take autonomous decisions that manage data replication.

In the above-described examples, users specify the way they expect the system to behave (e.g., when a certain query should finish or a certain data would be

accessible). A QoE-oriented system must *know* what users expect and with that information, it must try to maximize user satisfaction.

In our QoE-oriented database system, users' expectations are expressed in terms of *Data Access Requirements* (DARs). Some examples of DARs are the deadline to execute a query, the acceptable freshness of a certain data replica and a period on which a certain dataset must be available to users. Programmers may include requirements in application programs or end users may specify them through applications' interfaces.

Figure 1-I illustrates the difference between a QoE and a non-QoE system and the use of user-defined requirements. For each statement or block of statements, a user may specify a set of requirements that the system should try to fulfill.

The level of user satisfaction in using the database systems is related to the amount of requirements that the system is able to fulfill. If the system executes user's commands and fulfills specified requirements, then the user will become happy with the system (Figure 1-II). But what if the system is unable to meet the requirements set by the user?

If the system tries to execute every submitted command, even though it is not capable to satisfy specified requirements, it will lead to user frustration, as the user/application may wait for a long time to realize that stated requirements would not be met (Figure 1-III). Besides that, resources may be wasted executing commands that may not be useful to users (as their requirements are not satisfied).

A QoE-oriented database management system must analyze the requirements specified by users, execute the operations only if possible and inform users as soon as possible when it is unable to meet the requirements. This eliminates (or at least reduces) the time lost by users waiting for the execution of commands that would not be performed in a satisfactory way and also reduces the processing time wasted doing tasks that would have no practical use for end users. Besides that, if the system (almost) immediately informs the user that a requirement would not be satisfied, he/she may change the requirement or take any other actions he/she wants to. This strategy (Figure 1-IV) would generate less dissatisfaction than the alternative in Figure 1-III.

A QoE database system is a superset of a traditional database system. Since the specification of QoE requirements is optional, it is still possible to apply the default (*best effort*) processing mode to statements or blocks of statements.

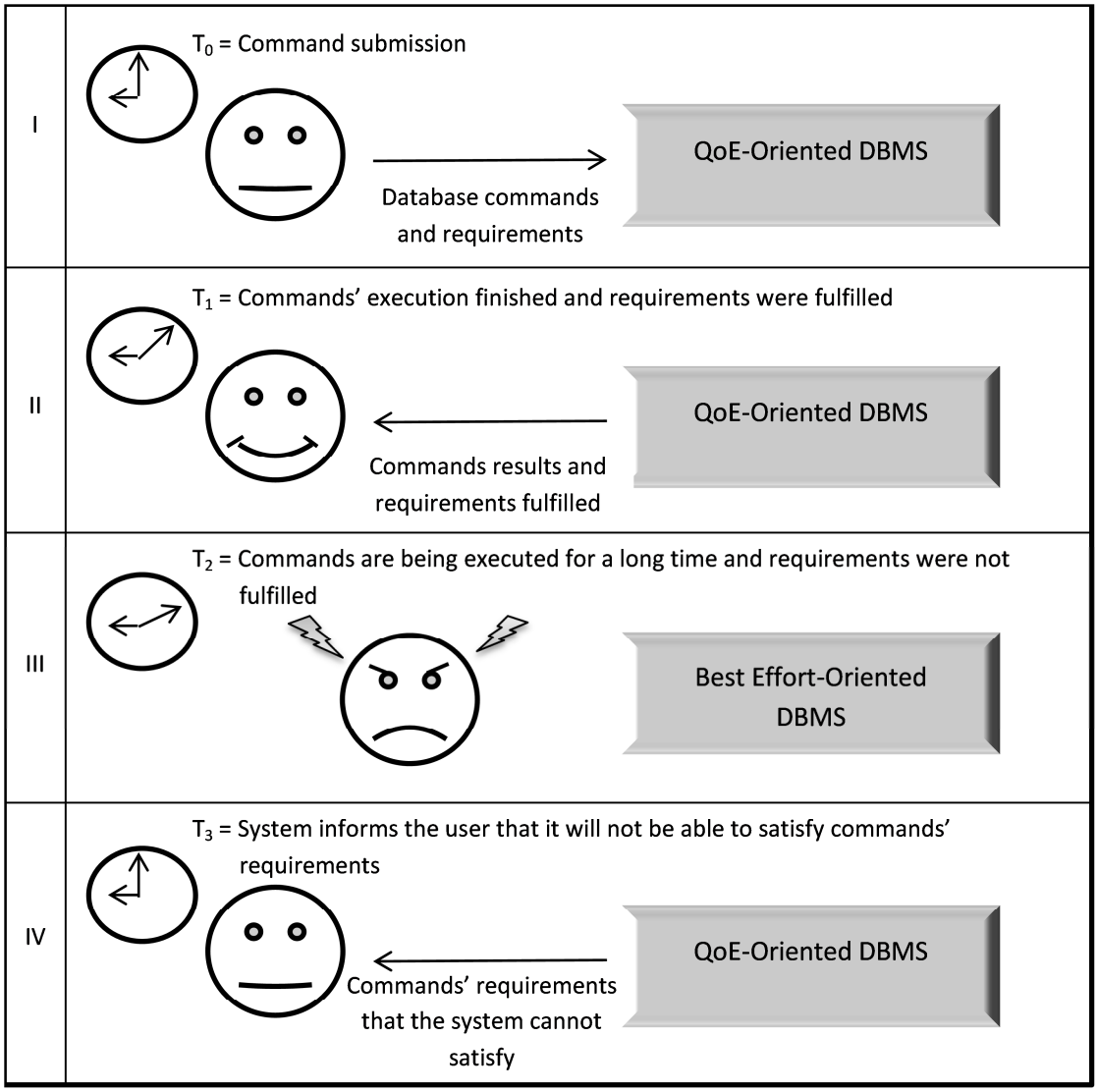

Figure 1 - User satisfaction in database systems – Alternatives

The rest of the introduction will overview the approach (Section 1.1), discuss how we test it (Section 1.2) and present the main contributions of the thesis (Section 1.3). Finally, Section 1.4 concludes the chapter by describing the structure of the remainder of the thesis.

1.1 The Quality of Experience Proposal

In order for a database system to support Quality of Experience, several mechanisms must be added to it, including runtime estimations, requirement-based task scheduling and capacity to decide on data-replication.

In this section, we provide an introductory overview of the main mechanisms proposed in this thesis.

In Section 1.1.1 we discuss the use of data access requirements to improve the QoE level provided by the database system. We also present the need of specialized

performance indicators for QoE-oriented databases. Then, Section 1.1.2 introduces some key aspects of the proposed architecture, including considered schedulers and the existence of task level requirements. In Section 1.1.3 we introduce the use of reputation to increase the level of QoE the system provides.

1.1.1 Data Access Requirements and Specific Performance Indicators

Our QoE-oriented system proposal is capable of guaranteeing a set of user-defined data access requirements (DARs). Such requirements are useful in several environments. For instance, consider distributed computing models in which users do not have full control over available computing resources (e.g. Grid Computing). Resources' owners may set limitations on the use of resources by remote users. Thus, shared resources (including data) may become unavailable to users for considerable periods, not only due to failures or scheduled maintenance, but also due to restrictions on resources use imposed by resources' owners. The execution of tasks in an environment with periodic outages and performance variation can be quite frustrating for users, especially if using traditional databases strategies, which initiate the execution of each task as soon as possible. In fact, it is common in highly distributed systems, that tasks have different requirements to be executed.

Proposed types of DARs are:

- Data freshness – specifies the minimal required freshness (timestamp) of a certain data replica in order for the system to use such replica in command execution;
- Execution deadline – specifies a deadline interval for command execution;
- Disconnected execution mode – specifies that the system should execute the command even though the user is not connected to the system;
- Data availability – specifies a period in which a certain dataset must be available to users;
- Execution periodicity – specifies a periodicity for command execution;
- Execution finish time – specifies an upper bound for command execution finish time;
- Execution start time – specifies a lower bound for command execution start time;
- Execution priority – specifies the command priority.

Although some of such DARs are especially useful in parallel and distributed database systems (e.g. data freshness requirements), most of them are also useful in centralized databases.

Proposed DARs can be associated to a single user statement or to a block of statements (that may be executed sequentially or in parallel, as defined by the user). Besides that, users can specify alternative DARs that may be used by the system to choose the ones that are feasible (or that lead to the highest levels of QoE). For instance, users may specify that a certain set of statements must be executed in no more than 5 minutes or, in case the system cannot do that, then it must be executed at night.

We also propose SQL extensions that enable DARs specification in SQL language.

QoE-oriented systems and best effort systems have different objectives. Hence, we should not use the same performance indicators to measure the performance of both kinds of systems. For instance, executing a high number of transactions per minute does not imply satisfying users' expectations. In fact, QoE-oriented systems aim to provide high QoE levels to users. Therefore, the *provided QoE level* is one among the possible performance indicators used to measure the performance in QoE-oriented systems. In Chapter 7, we describe such performance indicator and propose a set of key performance indicators for QoE-oriented database systems. Besides being used for performance evaluation, such indicators can also be used to alert system's administrators when the level of QoE the system provides is undesirable.

1.1.2 QoE-Oriented Scheduling and Placement

User defined requirements may be associated with one or more database commands (as detailed in chapter 3). We call *job* to a set of database commands that have shared DARs. Each job is transformed into one or more *tasks* (e.g. database operations) and associated task level requirements. A job's tasks should only be executed if all the DARs associated with the job can be satisfied (i.e. all task level requirements can be fulfilled). In our architecture, we have a community scheduler that is responsible to transform jobs into tasks and to assign task execution to data services, and a tasks scheduler which evaluates if task level requirements can be satisfied or not and schedules tasks execution at data services level. Chapter 4 presents a formal definition of jobs and tasks, the generation of tasks and task level requirements and the use of proposed schedulers in centralized, parallel and distributed databases.

Figure 2 presents an overview of the proposed architecture in an environment with several data services (i.e. parallel or distributed database system). The community scheduler is the system component responsible for global operations, including assigning tasks execution to data services and maintaining reputation information about participating services. At each data service, there is a tasks scheduler, which is the local scheduler responsible to do task level requirements evaluation and local command scheduling.

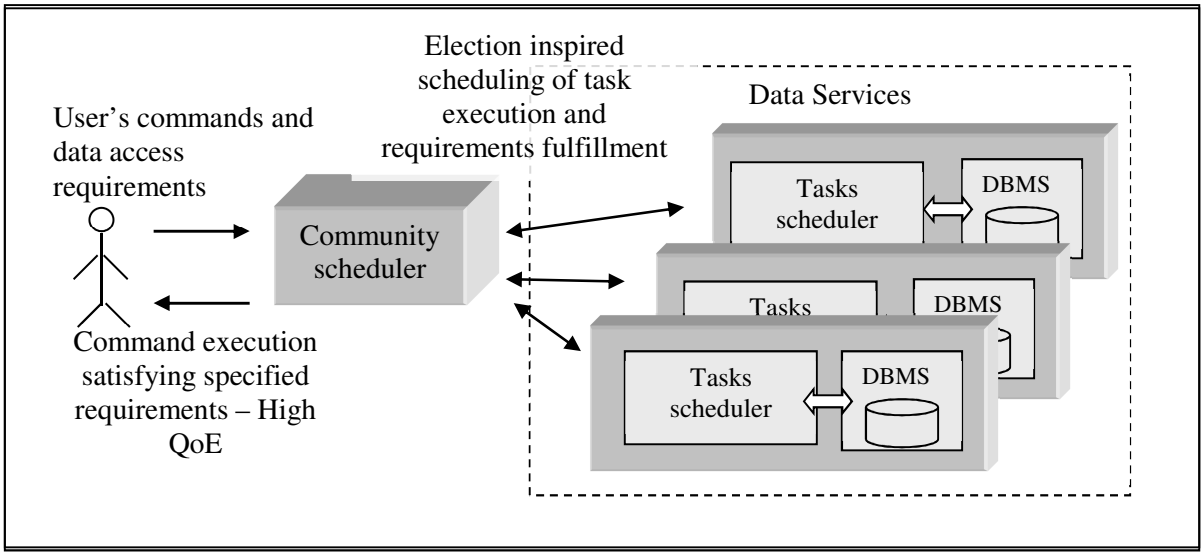

Figure 2 - QoE-oriented system in a multi-data services environment - Overview

After having transformed users' jobs and DARs into tasks and task level requirements, the community scheduler initiates task execution negotiation with data services. If there is any task level requirement that cannot be satisfied (leading to the impossibility of satisfying a user's DAR), then the QoE-oriented database system immediately informs the user that it cannot execute the command and satisfy specified requirements. Therefore, the user may change a DAR or give up on command execution.

Task execution assignment to data services is based on an election-inspired scheduling model. In such model, data services should commit themselves on fulfilling task level requirements and make promises on the required interval to execute the task. The community scheduler uses reputation information when deciding which service should execute a task. Besides that, election inspired mechanisms are used to identify data replicas that can improve the level of QoE the system provides. Chapter 5 discusses election-inspired scheduling and placement.

Services do not execute concurrently all the tasks assigned to them. Each service executes just a few tasks concurrently and maintains a task queue with remaining tasks. During task execution negotiation, each service must evaluate if it can execute the task that is being negotiated, satisfying its requirements, while executing all the tasks the service has already committed itself to execute, and also satisfying corresponding requirements. The service's local scheduler automatically adjusts the number of queries that are executed concurrently by the underlying DBMS (multiprogramming level) in order to increase the number of tasks the site can execute while fulfilling negotiated requirements. The methods used by local schedulers to evaluate requirements fulfillment, schedule query execution and automatically adjust the number of concurrently executed queries are discussed in chapter 6.

1.1.3 Reputation for QoE

Our proposals are capable to handle several autonomous data services that may agree (or not) to provide a certain service (i.e. database operation) according to consumer's terms (i.e. task level requirements).

When a data service agrees to specified terms and the community scheduler selects such service to execute a certain task, then the service should fulfill the specified requirements.

Sometimes, selected data services may fail (intentionally or not) to achieve negotiated task level requirements. For instance, requirements fulfillment failure may happen when data services misestimate the size of a certain task. Besides that, when there are no penalties, data services may intentionally agree to execute a task whose requirements they cannot satisfy, just in order to obtain some benefit. Anyway, when a service fails to satisfy a task's requirement, the system will probably fail to satisfy some of the user's DARs, thus reducing user's QoE.

Therefore, the community scheduler should always assign a task to the most dependable data service among the ones that can execute such task. In order to do that, our community scheduler maintains reputation information about service's capacity to maintain its commitments of satisfying specified requirements and about the precision of services' runtime estimations. Reputation information is used to support task execution assignment to data services, as proposed in Chapter 5.

The use of the proposed reputation system contributes significantly to increase the level of QoE the system provides, as demonstrated in Chapter 8.

1.2 Evaluation Methodology

In order to evaluate the techniques proposed in this thesis, we will make a set of experiments that provide quantitative results and show the importance of QoE oriented strategies. The experiments will be based on a prototype that implements most of the proposals, and lab experiments run over benchmark data.

In order to show the usefulness of QoE related mechanisms, we will build several experiments that run over three scenarios: global warehouse, parallel warehouse and centralized on-line transaction processing (OLTP) application. In such scenarios, we present the use of DARs and, when relevant, compare proposed techniques with best effort strategies, like round-robin and on-demand scheduling.

Besides such three scenarios, we also use made a set of experiments to evaluate some specific aspects of proposed techniques, like the use of reputation, the management of tasks queues and the query execution time estimation strategy.

We evaluate the proposed techniques using a set of metrics, which include the specialized metrics proposed in Chapter 7, which are designed to measure the performance of QoE-oriented systems.

Chapter 8 describes the experiments and presents the experimental results that prove the importance of proposed strategies. The experimental testbed environment is detailed in Appendix A.

1.3 Main Contributions

In this thesis, we advance several concepts and mechanisms that are relevant for implementing a QoE-oriented database system. The main research contributions of the research can be summarized as:

- → **Uses of Quality-of-Experience in databases systems** - Most existing strategies for data management are best effort oriented. In this work, we detail the use of Quality-of-Experience in data management, presenting the main benefits of such approach and the mechanisms that enable the system to provide high QoE to users.
- → **Key Performance Indicators for QoE-oriented Systems** – Traditional performance indicators for database systems are not adequate to measure the performance of QoE-oriented systems. We define a set of specialized KPIs that can be used to estimate the QoE levels a system provides, to compare QoE-oriented systems and to alert systems' administrators when the system is providing low levels of QoE.
- → **Definition of Data Access Requirements and proposal of SQL Extensions** - We define a set of Data Access Requirements (DARs) that users may specify for database operations and also propose some SQL extensions to enable the specification of Data Access Requirements in SQL.
- → **Election inspired query scheduling model –** An election inspired query scheduling model which combines task-based requirements and runtime estimations is proposed in order to provide a highly dependable environment while maintaining site autonomy.
- → **Mapping of user-defined requirements into task level requirements and use of requirements for scheduling command execution and placing data** – We present how to map user-defined DARs into task level requirements that should be fulfilled by data providers. Data providers should satisfy such requirements in order to improve users' satisfaction and also to reduce the amount of unnecessary (or useless) work executed by a system. Proposed replica placement strategy also considers task level requirements, providing a more reasonable use of data replication, reducing the number of useless data replicas and increasing space for placing replicas that actually increase the QoE levels the system provides.
- → **Reputation Models for QoE-oriented Database Systems** – We use reputation systems to rank service providers according to their commitment to maintaining their promises on satisfying certain requirements and to adjust services' runtime estimations. Reputation systems increase system's dependability and QoE levels.
- → **Dynamic Replica Placement Strategy for QoE** – We propose a strategy that detects which data may be replicated (to which site) in order to increase the level of QoE provided by the system.
- → **Data service query scheduling policies** – We propose how to implement external schedulers that consider query execution-related requirements. The

proposed policies aim at determining whether task requirements of the submitted task and of executing tasks would be fulfilled if the task was accepted for execution, and to maximize the number of database queries to be executed while fulfilling specified requirements.

→ **Query execution time estimation method** – Foreseeing query execution time is an important tool in several situations. We propose a query execution time estimation method that provides some reasonable estimate.

→ **Automatic adjustment of the multi-programming degree for fulfilling task level requirements** – Query execution time can be highly affected by concurrent execution of others queries. We propose a method to automatically adjust the number of simultaneously executed queries considering specified task level requirements.

→ **Actual implementation of a prototype with the proposed features and experimental evaluation** – Results experimentally obtained with the actual evaluation of proposed strategies prove the validity of such strategies. Experimental evaluation is done considering centralized, parallel and distributed database systems.

Parts of this work are described in the following publications:

- Conference Papers:
  - Costa, R.L.C., Furtado, P. (2009) *Runtime Estimations, Reputation and Elections for Top Performing Distributed Query Scheduling*. 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2009). pp. 28-35.
  - Costa, R.L.C., Furtado, P. (2008) *QoS-Oriented Reputation-Aware Query Scheduling in Data Grids*. In 14th International Euro-Par Conference on Parallel Processing (Euro-Par 2008). pp. 489-498.
  - Costa, R.L.C., Furtado, P. (2008) *A QoS-Oriented External Scheduler*. In 23rd Annual ACM Symposium on Applied Computing (ACM SAC 2008). pp. 1029-1033.
  - Costa, R.L.C., Furtado, P. (2008) *Scheduling in Grid Databases*. In Proc. of the 22nd International Conference on Advanced Information Networking and Applications - Workshops (AINAW 2008). pp. 696-701.
  - Costa, R.L.C., Furtado, P. (2007) *An SLA-Enabled Grid DataWarehouse*, Eleventh International Database Engineering and Applications Symposium (IDEAS 2007). pp. 285-289.
  - Costa, R.L.C., Furtado, P. (2006) *Data Warehouses in Grids with High QoS*, 8th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2006). pp. 207-217.
- Book Chapters:
  - Costa, R.L.C., Antunes, R., Furtado, P. (2009) *Optimizer and Scheduling for the Community Data Warehouse.* Architecture, Methods and Supporting

Technologies for Data Analysis Series: Studies in Computational Intelligence Vol. 225, pp. 21-55. Springer.

- Costa, R.L.C., Furtado, P. (2009) *QoS-Oriented Grid-Enabled Data Warehouses.* Data Warehousing Design and Advanced Engineering Applications: Methods for Complex Construction, pp. 150-170. IGI Global.
- Costa, R.L.C., Furtado, P. (2009) *Deploying Data Warehouses in Grids with Efficiency and Availability*. Complex Data Warehousing and Knowledge Discovery for Advanced Retrieval Development: Innovative Methods and Applications, pp. 208-229. IGI Global.
- Costa, R.L.C., Furtado, P. (2009) *Placement and Scheduling over Grid Warehouses*. Grid Technology for Maximizing Collaborative Decision Management and Support: Advancing Effective Virtual Organizations, pp. 83-104. IGI Global.

1.4 Dissertation Organization

In the following chapter, we describe related work. Then, chapter 3 proposes a set of Data Access Requirements types and SQL extensions that can be used to specify such requirements.

Chapter 4 defines *jobs* and *tasks*, presents the community scheduler and the tasks schedulers, and how users' commands and DARs are transformed into tasks and task level requirements.

In Chapter 5, we detail election-inspired scheduling. We present how to select possible candidates to execute a task and elect a winner to execute a task among the candidate data services. We propose how election-inspired strategies can provide information about replica placement. We also detail reputation mechanisms that are used to increase the system's dependability and the QoE level it provides.

Chapter 6 describes task level scheduling. We present how tasks scheduler evaluates if task level requirements can be fulfilled or not. We also describe the proposed query execution time estimation method.

Then, Chapter 7 presents a set of Performance Indicators for QoE-oriented database systems. We formally define proposed indicators and present some examples on their usage. Besides that, we discuss the use of such indicators to alert system's administrator when the system is providing undesirable levels of QoE.

Chapter 8 proves the usefulness of the proposed strategies by presenting experimental results obtained in centralized, parallel and distributed databases scenarios. Besides that, it also evaluates some key aspects of the proposed features, which include the use of reputation for scheduling, queues management in data services local schedulers and time estimation.

In Chapter 9, we present some final considerations and open aspects left for future work.

Appendix A details the experimental environments used to evaluate the proposed techniques.

2 Related Work

In this work, we propose strategies to provide QoE to database users. Our strategy considers user-specified requirements such as the freshness of data replicas that can be used to provide QoE while answering queries, or the execution deadline of a certain block of statements (such requirements are formally defined in the following chapter). Besides that, in order to render possible QoE-related strategies, we propose specialized query scheduling and dynamic placement strategies. Our scheduler also uses a Reputation System as part of its strategy to schedule command execution, and estimation of the necessary time to execute a database command is also important to provide QoE to users.

In this Chapter, we discuss some work related to the main topics that we deal with in this thesis. Such related work is organized in the following areas:

- QoE-oriented systems (Section 2.1);
- Scheduling and placement in parallel and distributed databases (Section 2.2);
- Admission control and real-time databases (Section 2.3);
- Runtime estimations in database systems (Section 2.4).

Section 2.5 concludes the chapter.

2.1 QoE-oriented Systems

Quality of Experience is a measure of user satisfaction when using a certain service or system [Kilkki, 2008; ITU, 2007]. The QoE level provided by a system includes the effects of all the system's components and all the underlying components (e.g. network) [ITU, 2007].

Quality of Experience is distinct from the traditional Quality of Service (QoS) metrics. While QoS is mainly focused on technology and performance in the technical perspective, QoE is a user-centric approach that considers users' goals [Marez & Moor, 2007; Zapater & Bressan, 2007].

Measuring QoE is not easy, due to its subjectivity. Traditional QoE measuring methods are based on opinion tests and questionnaires [Moller, Engelbrecht & Kuhnel, 2009; Sanchez-Macian et al., 2006], like the Mean Opinion Score (MOS). MOS [ITU, 1996] is a graduation on results observed by users during a listening-opinion test. Table 1 presents MOS's score values.

Table 1 - MOS Scores

Obtained Quality	Score
Excellent	5
Good	4
Fair	3
Poor	2
Bad	1

Questionnaire-based methods have several problems, like the difficulty in selecting the users that will answer the questionnaire, the amount of time it takes to retrieve users' opinions and the need for continuous user cooperation [Sanchez-Macian et al., 2006].

The use of QoS to estimate provided QoE is used in some works. Sanchez-Macian et al. (2006) use ontologies and rules in a framework to calculate QoE by the values of QoS parameters. Kim et al. (2008) estimate QoE using information about network-level QoS metrics. Krauter, Buyya & Maheswaran (2002) present the factors that influence QoS and QoE in multimodal dialogue systems, like kiosks and smart home environments. Among the factors that influence the QoE level in multimodal systems are the system's utility and effectiveness. Martinez-Yelmo, Seoane & Guerrero (2010) organize network-related aspects into layers (similar to the TCP/IP model) and discuss how each layer can affect QoE.

QoE measurement can also be done with the use of Key Performance Indicators (KPIs). The process of using KPIs to measure QoE involves identifying the most appropriate KPIs and weighting them [Nokia, 2004].

To the best of our knowledge, our work is the first one to incorporate the QoE concept in database systems. Our strategy is based on making the system fulfill execution-related constraints specified by users. Performance indicators are used to alert system's administrators when the level of QoE provided by the system is unacceptable.

2.2 Scheduling and Placement in Parallel and Distributed Databases

Query scheduling in parallel and distributed databases has been studied for several years and there is a wide range of works on such issue.

Parallel data allocation and processing approaches typically rely on partitioning the data. Stöhr et al (2000) propose a *multi-dimensional hierarchical fragmentation* strategy called MDHF for use in OLAP systems based on shared-disk parallel machines. In MDHF, fact tables are partitioned based on a set of dimension attributes. Authors show that queries' performance benefits from the use of dimension-based partitioned fact tables, even when the submitted queries are over hierarchical levels different from the ones that were used to generate the facts table's fragments.

Röhm et al (2000) discuss the execution of parallel OLAP queries in the PowerDB system, which is composed by a set of independent processing nodes (off-the-shelf computers) and a *coordinator* node. Each processing node has its own DBMS. The

coordinator node is responsible for scheduling query execution between processing nodes and to collect execution results, sending them back to users. Authors compare the use of two data placement strategies: (i) fully replication of the database into all nodes, and (ii) an hybrid approach where the largest table is partitioned across all nodes and the other tables are fully replicated at all nodes. The presented experimental results stand that the hybrid approach leads to higher throughput than the other one.

Virtual Partitioning [Akal et al, 2002] is another database allocation scheme experimentally tested in PowerDB. Its goal is to achieve intra-query parallelism in cluster-based data warehouses. In such strategy, all data is fully replicated into all processing nodes. Clustered primary key indexes are created on the facts table, and one (or more) of the primary key attributes is chosen to be a *partitioning attribute*. Assuming that the possible values on the partitioning attribute are known, virtual partitions are created, considering each one a range of values in the partitioning attribute. Submitted queries are transformed into several subqueries each one addressing a virtual partition. This strategy can lead to performance gains if the underlying DBMS considers the range predicates very selective and uses the clustered index to access only a fragment of the fact table (reducing I/O) instead of doing a full table scan in such table. Hence, one great challenge on Virtual Partitioning is to properly choose partitions' bounds (determining its sizes). Akal et al (2002) claim that they should be chosen so that the duration of parallel subqueries execution is approximately equal.

The Node-Partitioned Data Warehouse (NPDW) [Furtado, 2004; Furtado, 2004b] is another strategy for the implementation of parallel data warehouses in shared-nothing systems. In NPDW, facts and large dimension tables are hash-partitioned and the resulting partitions are placed on different nodes. In order to minimize repartitioning costs, the most frequently used equi-join attribute should be used to generate facts tables' partitions. Small dimension tables are replicated at all nodes, while large dimension tables are hash-partitioned as well.

The Skalla System [Akinde et al, 2002] enables the use of OLAP tools in order to analyze distributed network trace data. IP flow data[1] is obtained in several *data collection points*. Next to each data collection point is placed a *Skalla site*, which is a local data warehouse storing the information captured in the collection point. Hence, it is reasonable to consider that the *conceptual* fact table (about IP flow) is partitioned across the local data warehouses. OLAP queries are submitted to a Skalla coordinator, which is responsible to construct a distributed query execution plan, to submit it to Skalla sites, to collect the results and to send them back to the user. Submitted OLAP queries are translated into *GMDJ* expressions (which are composed by specialized *GMDJ* operators (Akinde & Böhlen, 2001). Distributed execution plans are constructed to execute each GMDJ expression.

Most of the existing works on *generic* distributed databases aim at providing high performance systems. Stonebraker et al. (1996) proposed Mariposa, which is one of the first works on Distributed Database Management that considered user-specified constraints. Mariposa query scheduling model is economic-inspired: the user specifies how much he/she can pay for a certain query to be executed by a certain time limit. Each participating node specifies the execution (monetary) cost and the necessary time to execute the query at the node. The system considers the available budget and the

[1] Information about packets transferred from a given source to a given destination. This includes, for example, origin and destination IP, port and mask, besides the number of transferred packets and the total transferred bytes.

necessary cost to execute the query when choosing which node should execute the query. Mariposa also considers an economic model to distribute data replicas among nodes [Sidell et al., 1996]. Mariposa aims at providing high performance while satisfying a certain budget, which is not the objective of our QoE-Oriented Distributed Database (our QoE-oriented database considers several types of user-defined data access requirements).

Garlic [Haas et al., 1997] is another scheduler for widely distributed and heterogeneous data sources. In fact, it is a middleware that enables data access through the use of wrappers. Garlic uses a set of rules to construct global execution plans for distributed queries. Wrappers participating in the execution of distributed queries, transforming operations of the global query execution plan into operations that can be performed by the database corresponding to the wrapper. Distinct global execution plans are evaluated and the system chooses the plan with the lowest foreseen execution cost.

In [Li et al, 2005a], IBM's DB2 Information Integrator is used together with some complementary modules to do efficient query scheduling in federated databases. Estimated and measured query execution costs are used in order to foresee which database would provide the lowest query execution time.

Grid based systems became of special interest in the last decade. Grid based applications are usually run over some kind of Grid Resource Management (GRM) system, like Globus Toolkit [Foster & Kesselman, 1997] or Legion [Grimshaw et al., 1997]. In fact, GRM systems provide some basic functionality that can be used by a wide range of application systems, like remote job monitoring or efficient data transfer between sites.

The Legion GRM system creates a virtual machine abstraction of the grid. Each available resource is modeled as an object. User applications are instantiated as objects of the *Application* class. Users can specify several parameters while instantiating objects of such class. Legion has some built-in scheduling mechanisms that implement random and round-robin job assignment policies [Natrajan, Humphrey, & Grimshaw, 2004], but it also supports the use of third part job schedulers. In fact, the scheduler and the scheduling strategy that should be used by an application are examples of the parameters that can be specified while instantiating objects of the *Application* class.

The Globus Toolkit can also use application level schedulers. Condor-G [Frey et al., 2002] and Nimrod-G [Buyya, Abramson, & Giddy, 2000] are examples of such schedulers.

In Condor-G, execution requirements can be associated to jobs. Some examples of possible requirements are: the network domain, the file system and the operation system on which the job can be executed. Condor-G uses classified advertisements (*ClassAds*) to advertise jobs' requirements and nodes' characteristics. A specialized process is responsible to scan the advertisements and to do the matchmaking between jobs and nodes that are compatible. Nimrod-G is economy inspired (like Mariposa). Each job may have a budget and a deadline. Auctions are conducted by the system's central module. Each job is executed by the node with the lowest execution cost (considering only the nodes that can finish job's execution by a specified deadline). Both Condor-G and Nimrod-G are *general purpose* grid schedulers.

There are also some works on the use of reputation to schedule grid jobs. Silaghi, Arenas & Silva (2007) present *generic functions* that determine reputation

values of service providers and of issues of interest. Reputation-based job scheduling in donation grids is discussed by Sonnek et al. (2006). Such work aims at obtaining high performance when executing queries in unreliable environments, where (malicious) nodes can answer a query with uncertain data. There are other works that use reputation systems to detect malicious nodes [Kamvar, Schlosser & Garcia-Molina, 2003; Singh & Liu, 2003]. In our system, reputation is used to rank (autonomous and heterogeneous) data services on their capacity to maintain promises and commitments.

Some of the available works on scheduling in grids aim at providing good load balancing among available resources. Cao et al (2003; 2005) use software agents to represent available processing resources (i.e. workstation clusters and multiprocessor machines). Some of the agents' functionalities are: (i) schedule job execution at the resource it represents; (ii) publish the resources capabilities, and (iii) cooperate with other agents to find a resource to execute jobs that cannot be executed locally. A job execution time prediction system (PACE [Nudd et al, 2000]) is used by agents to predict resources' performance. An agent verifies if the resource it represents can execute the job by the required deadline. If so, then the job is scheduled to be executed by such agent's resources. If the required deadline cannot be achieved by the resource the agent represents, then the agent searches between its *neighbors* for one that can execute the job by the specified deadline. Agents do not do exhaustive searches between available resources for the one that would finish job execution earlier.

Koenig & Kale (2007) discuss load balancing in grid applications with high volumes of inter-processor communication. Grid's resources are organized hierarchically into clusters according to communication latency. The proposed strategy comprises two phases: (i) jobs are allocated into nodes in order to minimize inter-cluster communication; and (ii) intra-cluster job assignment considers the processing capacity of each node - more work is assigned to nodes that have the fastest processors.

The abovementioned *general purpose* schedulers do not consider database specific factors, like data replica and placement and skews that may happen during database query execution. Such factors are commonly considered in the context of *data grids*.

In *data grids*, the grid infrastructure is used to store huge volumes of widely distributed data or to manage the execution of jobs that generate or consume great volumes of data [Krauter, Buyya & Maheswaran, 2002; Venugopal, Buyya & Ramamohanarao, 2006].

Ranganathan & Foster (2004) evaluate several job scheduling strategies for data grids, including *Random*, *Least Loaded* and *Data Present.* In the random strategy, jobs are randomly assigned to participating nodes. In *least loaded* strategy, each job is assigned to the node that has the lowest number of jobs waiting for execution. In both random and least loaded strategies, if the node selected to execute a job does not store the data that the job needs, then such data is fetched from a remote site during job execution. In *data present* strategy, a job can only be assigned for execution to a site that already stores the data that is necessary to execute the job. Authors claim that accessing remote data in grids may be time consuming and that, in some situations, jobs should only be assigned to sites that already store required data locally.

Park & Kim (2003) also discuss the use of locally stored data and remote data while executing data grid jobs. Authors present a cost model to predict job execution time in several configurations, like: (i) executing a job at the site that stores required

data; (ii) executing a job at the site on which the job was submitted but accessing remote data; and (iii) executing a job at a remote site but using the data from the site on which the job was submitted. The proposed cost model considers several parameters, including network bandwidth and input and output data size. A central scheduler assigns each job to the site that has the lowest foreseen execution time.

Although most of the initial works on data grids considered the use of flat files, the use of grid-enabled database management systems is very promising [Nieto-Santisteban et al., 2005; Watson, 2001].

Watson (2001) proposes the use of ODBC/JDBC to build federated databases composed by heterogeneous systems. Alpdemir et al. (2004) use a set of OGSA (Open Grid Services Architecture [Foster et al., 2002]) compliant web services to provide access to the Polar* [Smith et al., 2002] distributed query processor. The Polar* processor builds distributed execution plans by dividing each query into a set of operators that are executed by distinct nodes.

Scheduling in grid-based data warehouses is discussed in [Lawrence & Rau-Chaplin, 2006; Dehne & Lawrence, 2007; Wehrle, Miquel & Tchounikine, 2007]. Dehne & Lawrence (2007) and Lawrence & Rau-Chaplin (2006) use a two-tiered data warehouse with local cached data at the first tier and database server at the second tier. The system aims at executing queries using only locally stored data. If it cannot be done, then incoming query is transformed into a set of queries that access local data and some complementary ones that are executed by the database server (at the second tier). Cached data is also maintained in an R-tree at database server level. This strategy aims at reducing data movement over the grid.

Wehrle, Miquel, & Tchounikine (2007) use the Globus toolkit and a set of services to build a distributed data warehouse. Dimension data is replicated across participating nodes. Fact's data is partitioned and partitions are distributed at participating nodes. Each node has a *local data index* that provides information about locally stored data. The system tries to use locally stored data to answer queries (through the use of the local data index). When it is not possible, a *communication service* is used to search for required data at remote sites. The *communication service* uses the *local data index service* of remote nodes to access remote data.

None of the above presented works on grid query scheduling deals with several types of user-defined requirements. Such works are oriented to provide high performance, not high QoE.

Database replication can be used in distributed databases to improve query execution performance and data availability, but determining the necessary number of replicas and optimally placing such replicas into the nodes of a distributed system is an NP-hard problem [Loukopoulos & Ahmad, 2000].

Wolfson & Jajodia (1992) present algorithms to dynamically replicate data and to minimize communication costs and times.

Ranganathan & Foster (2001) evaluate several strategies of dynamic file replication in grids, including *Best Client Replication* and *Cascading Replication*. In such strategies, the system creates a new file replica whenever the number of accesses to a certain file reaches a specified threshold value. The *best client node* of a certain file is the node with the highest number of access requests to the file. In *Best Client Replication*, the new replica of a file is created at the file's *best client node*. In

Cascading Replication, the system places the new replica file at the first node in the path between the file's node and the file's *best client node*.

In [Li et al, 2005b], IBM's DB2 Information Integrator is used together with complementary modules to deal with federated databases. Dynamic replica placement is used to maintain the average query execution time bellow a threshold value.

Sathya, Kuppuswami & Ragupathi (2006) discuss *Best Replica Site Replication*, *Cost Effective Replication* and *Topology Based Replication*. The *Best Replica Site* strategy is inspired in the *Best Client Replication* strategy: the main difference between such policies is that in *Best Replica Site* the site where the new replica would be created is chosen considering the number of accesses to the file, the replica's expected utility for each site and the distance between sites (*Best Client* considers only the number of access requests to the file). *Cost Effective Replication* uses a cost function to evaluate the cost of accessing a replica at each site: the new replica is created in the site that minimizes the foreseen costs. In *Topology Based Replication*, file replicas are created in the node with the highest number of direct connections to other nodes.

Lin, Liu & Wu (2006) also consider system topology in order to choose where to place data replicas. In [Lin, Liu & Wu, 2006], the database is placed at the root node of a hierarchical (tree-like) grid. The system tries to answer each job using data placed at the node where the job was submitted. If the node does not store the required data, then the system looks for the data at the node's parent node. If data cannot be found there, then the system looks for data at the node's grandparent, and so on. When the number of hops is greater than a certain threshold value, the system places a new data replica at the node that maximizes the number of queries that can be answered without creating new replicas.

Dang, Hwang & Lim (2007) considers the amount of data accessed from a data set in order to decide if the data set should be replicated or not. When the amount of accessed data is greater than the system's average access rate to data files, the system creates a new replica of accessed data. If the amount of accessed data of a certain data set is smaller than the system's average access rate by a certain amount, then the data replica is dropped. The authors use a distance function in order to choose the location for data replicas: new replicas are created at the node that minimizes the used function.

Haddad & Slimani (2007) aims at maximizing the economic value of data stored at each node. Authors propose that each data fragment have a certain price. System's nodes try to foresee the future price of data fragments and to store fragments that are forecasted as the most valuable ones.

Besides the abovementioned works on replica selection and placement, there are also works that deal with replica catalogs and replica synchronization and consistency ([Chen et al, 2005; Chervenak et al, 2004; Chervenak et al, 2005; Deris et al, 2004; Düllmann & Segal, 2001]). We do not deal with these topics in this work, considering that capabilities may be offered by the underlying infra-structure system (e.g. the grid management system).

Existing works on scheduling and placement are not capable deal adequately with user-defined execution-related constraints. On the other hand, our strategies are oriented to improve the level of QoE the system provides and, then, are capable to deal with user-defined requirements.

2.3 Admission Control and Real Time Databases

In our approach for QoE-oriented database systems, users can define requirements for the execution of database commands. The execution of queries and transactions with constraints is also studied in the context of *real time* databases. But in real time databases the objective (and variety) of constraints is distinct from the objective in QoE-oriented databases.

In fact, in real time databases, data may become outdated and transaction constraints (e.g. deadlines) are used to guarantee that transactions are executed while data is still valid (i.e. data has temporal validity and constraints are used to maintain temporal data consistency) [Ozsoyoglu & Snodgrass, 1995; Ramamritham, 1993]. Hence, real time transactions may only be *correct* it they meet executing time constraints and use time consistent data [Stankovic, Son & Hansson, 1999].

Kang (2003) deals with QoS management in main-memory real time databases. The author presents three QoS metrics: deadline miss ratio, database freshness (which refers to the fresh of the entire data in the database) and perceived freshness (which refers to the data accessed by timely transactions). The proposed strategy aims to guarantee a certain perceived freshness value, while transactions are grouped according to desired miss ratios. In order to increase the number of query operations that the system can afford to execute, the authors propose a mechanism that does not update immediately the entire database with recent data acquired from sensors. *Hot* data (i.e. data that is frequently accessed) is updated immediately while *cold* data (i.e. data that is not regularly accessed) is updated on demand (reducing CPU utilization). An admission control mechanism estimates the CPU utilization of every incoming transaction. A new transaction is only accepted if the required CPU is available. The system monitors deadline miss ratios and dynamic adapts the data update policy in order to reduce the measured miss ratio.

But real time databases are generally used in controlling systems, where a deadline miss may result in tragic situations [Ramamritham, 1993]. In QoE-oriented databases, constraints (requirements) are used to provide high levels of Quality-of-Experience and a failure to fulfill a requirement will result in user's frustration. In some situations, a QoE-oriented database system mail choose to not satisfy a user requirement in order to fulfill some others and, then, increase the total QoE level provided by the system. A QoE-oriented database system should be able to deal with a wider variety of types of requirements than a real time database system.

Early works on admission control mechanisms (including the ones used in real-time databases) often study how to specify an upper bound to the multi-programming degree (i.e. the number of commands being executed concurrently by the DBMS) without leading to system thrashing. Schroeder et al (2006a) studies the lowest multi-programming degree that can be used without hurting system performance (if the multi-programming degree is too low, one will have a lower throughput, since not all DBMS resources will be utilized). In Schroeder et al (2006b), the authors stand for the use of an external scheduler in order to schedule transaction execution considering expected response times. Database transactions are organized into classes according to expected response time and placed in a transactions queue. The mean execution time of transactions of each class is measured by a performance monitor. A transaction is accepted to be executed if the time that it would spend in the queue plus the mean execution time for the transactions of the class it belongs to is lower than its expected

response time. Query execution performance depends, among others, on the number and type of database commands that are being executed concurrently by the DBMS. This work aims to use low multi-programming degrees in order to reduce the impact of multiple concurrently query executions in each query's execution performance and, then, increase the accuracy of using the mean execution time of transaction execution of a certain class to predict future executions.

Elnikety et al. (2004) present an admission control system to be used in dynamic content Web sites. The main objective of the work is to maintain system's throughput even in high load situations (i.e. prevent thrashing). In such work, an admission control system (called Gatekeeper) is implemented in a proxy between an application server and a database server. It monitors the execution time of application servlets (i.e. web site components that interacts with the database system). The average response time of a servlet execution is the load that such servlet produces. Authors also define the system capacity as the load level that leads to the highest throughput. Every time a servlet execution starts, the admission control mechanism estimates if the load produced by such servlet would make the system's load exceed its capacity. If it is true, then servlet execution does not start immediately: the servlet is placed in an admission (First-In-First-Out) queue. When a servlet execution ends, the admission control system evaluates if it can start the execution of a servlet of the admission queue. The execution of accepted servlets is scheduled using the shortest-job-first (SJF) strategy. Aging is used in order to prevent long jobs from starving.

Therefore, existing admission control systems are not capable to adequately deal with distinct types of user-defined execution-related constraints. Our proposals, on the other hand, consider several types of user-defined requirements in order to schedule command execution, verifying what requirements can be fulfilled and choosing when to execute submitted commands while satisfying specified requirements (e.g. to start the execution of a certain long-running transaction immediately or to start its execution at night, if executing immediately is not a requirement).

2.4 Runtime Estimations in Database Systems

An important issue in providing QoE for users is to verify which requirements may be fulfilled and which may not. In order to do that for requirements involving deadlines or target times the system needs to estimate a query execution time.

In [Spiliopoulou, Hatzopoulos & Costas, 1996] the authors propose some models to estimate communication and I/O costs in query execution over parallel shared nothing machines. Such models are used to compute the execution time of query plan operators. Authors use the costs of individual query operators to estimate the execution time of a query in the considered environment. The work does not consider the interference in query execution performance that may exist when more than one query is executed simultaneously.

Gupta, Mehta & Dayal (2008) uses a machine learning approach to estimate a range for the execution time of a query. The authors use similar queries (do not consider sudden changes in workload). Besides that, it should be noticed that to define the time ranges to consider is a key aspect in the proposed strategy. Using a large number of small time ranges leads to high error in time estimation, while using a too low number

of large time ranges may turn out to be meaningless. Therefore, if the time ranges are not adequately chosen, then the system may fail to predict query execution time, in a way that leads to low user satisfaction.

Some recent research papers on estimating query execution time deal with progress indicators for long-running queries. These works include [Chaudhuri, Kaushik & Ramamurthy, 2005; Chaudhuri, Narasayya & Ramamurthy, 2004; Luo et al, 2005; Luo, Naughton & Yu, 2006]. Most of then are able of estimating the progress of a single query, without considering the existence of multi-query influence, which leads to high errors when estimating the progress of query execution in systems where several queries are being executed concurrently ([Luo, Naughton & Yu, 2006]).

Luo, Naughton & Yu (2006) discuss query execution progress indicators on multi-query environment. The authors consider that the progress indicator mechanism has perfect knowledge about the cost of a query that is still remaining to execute at a certain time and of the speed (units of cost per unit of time) of the execution of each query. Query execution is divided in stages that are somewhat similar to our workload executing phases (discussed in Chapter 6) and the amount of cost executed of each query at a certain phase is computed considering query's execution speed. In contrast to that work, we consider that the scheduling mechanism does not have a perfect knowledge of a query's execution speed and neither of the exact cost of a query that is still to be executed at a certain time. Instead, our strategy estimates the cost units that were processed and that are still remaining for execution at each phase change (entering or exiting of queries). An adaptive conversion function is used to model the relation between execution time and processed cost.

2.5 Conclusion

In this chapter, we discussed some works concerning the use of mechanisms in database systems that are somehow related to our QoE proposals.

First, we reviewed some works on QoE-oriented systems. Then, we presented some key works on query scheduling and data placement on parallel and distributed database systems. We also discussed some related work on database admission control systems, external schedulers and real time databases. Finally, we presented some work on query execution time estimation.

In the following chapter, we present our approach to provide high QoE to database users. We present how users can specify data access requirements and propose a set of requirements and SQL extensions that can be useful in many situations.

3 User Defined Requirements for QoE-oriented Database Systems

According to [ITU, 2007], Quality of Experience (QoE) is the measure of user's satisfaction when using a certain system or service. Therefore, a QoE-oriented system can be seen as a system whose main objective is to provide high satisfaction to users.

Our proposal of a QoE-oriented database system considers that the database system should take into account users' expectations (which is a key aspect of QoE-oriented systems [Zapater & Bressan, 2007]) when executing database operations.

In order to do that, we propose that users should have the possibility to specify their objectives through *Data Access Requirements* (DARs), which are execution requirements related to database statements or blocks of statements. In Section 3.1 we propose the way users can specify (and remove the specification) of requirements for statements, blocks of statements and database objects.

In Section 3.2, we formally define a set of types of DARs and some SQL extensions that enable users to use SQL in order to specify requirements from each of the proposed types. The proposed set of DARs covers some of the most common database operations' requirements.

Finally, Section 3.3 presents a summary of the chapter and some final comments.

Hence, the main contributions of this chapter are: (i) the specification of a set of types of Data Access Requirements and (ii) the description of SQL extensions that are used to specify user requirements in SQL.

3.1 Requirements: How to Specify and Remove Them

The main objective of a QoE-oriented system is to satisfy users. In fact, taking into account users' expectations is a key aspect of QoE [Zapater & Bressan, 2007]. Therefore, our approach to provide QoE in database systems' operations is to allow users to express their needs and to make the system take such needs into account when executing users' commands.

In such a context, we propose that users specify optional *Data Access Requirements* (DARs), together with database commands. Such requirements impose restrictions on statement execution (e.g. the use of a certain data replica that is not up to date) or objectives that should be met by the system during command execution (e.g. a deadline for query execution) or while managing stored data (e.g. a guarantee that a certain table would be available for users in a certain time period).

In the following section, we present the *Requirements Specification Area*, which is our proposal to include requirements specification in SQL commands. We discuss the use of the *Requirements Specification Area* in data manipulation and data definition commands.

Some of the most common operations may be applied also for a block of commands. For instance, a user may need to build a report in a certain deadline, and such report is built based on three database queries. Therefore, in order to build the report within the required time, all the three database queries must be executed before the deadline. Hence, the deadline is a requirement that applies for the block of three queries. In Section 3.1.2, we propose the use of the *Requirements Specification Area* together with a `BEGIN BLOCK/END BLOCK` that enables users to specify requirements that are valid for a block of statements.

In the real world, requirements may change over time. Therefore, there should be a way to enable users to cancel some of the specified requirements. In 3.1.3, we discuss how users can inform the system that DARs specified in the past should not be satisfied anymore.

3.1.1 Requirements Specification Area

In the QoE-oriented database system, each user's command may have a set of associated DARs. We propose the use of the keyword `REQUIREMENTS` to identify in SQL the beginning of an area of DARs specification. Such keyword may be followed by the *requirements area name*, as defined in Figure 3.

```
REQUIREMENTS [Requirements_Area_Name]
```

Figure 3 - Requirements Specification Area format

Therefore, data access requirements are defined in the *requirements specification area*. Each requirements specification area may contain several DARs' definitions, which should be separated by colons (the syntax of each of the proposed requirements is defined later in this Chapter). Requirements may also be grouped using parenthesis. The keyword `OR` can also be used to indicate that some group of requirements may be satisfied instead of another group.

Example 3.1 Figure 4 presents an example on the use of multiple requirements in a single user query. In such query, *Requirement_Definition_1*, *Requirement_Definition_2* and *Requirement_Definition_3* represent user defined requirements, such as a deadline for query execution or the possibility to use data replicas that have a certain freshness (the specification of user-defined DARs should follow the syntax that is defined in the following Sections). The system should fulfill requirements *Requirement_Definition_1* and *Requirement_Definition_2*, or *Requirement_Definition_3*.

```
SELECT *
FROM SALES
WHERE STATE_ID = 1
REQUIREMENTS MyRequirements
    (Requirement_Definition_1,
    Requirement_Definition_2)
    OR
    (Requirement_Definition_3)
```

Figure 4 - Requirements Definition Area - Example

Besides the use on data manipulation commands, user defined requirements can also be associated to database objects (e.g. tables and views). We propose that the requirements can be defined during or after object creation. In both situations, we also use the `REQUIREMENTS` keyword.

Example 3.2 In Figure 5, we present an example of requirements definition during table creation. Figure 6 presents requirements definition via an `ALTER TABLE` command.

```
CREATE TABLE CUSTOMERS (
 CUSTOMER_ID INTEGER PRIMARY KEY,
 CUSTOMER_NAME VARCHAR(100)
)
REQUIREMENTS MyRequirements
    Requirement_Definition_1,
    Requirement_Definition_2,
    Requirement_Definition_3
```

Figure 5 - Requirements Definition Area in a CREATE TABLE Command - Example

```
ALTER TABLE CUSTOMERS
ADD REQUIREMENTS Requirement_Definition_n
```

Figure 6 – Adding a Requirements Definition Area - ALTER TABLE Command – Example

3.1.2 Requirements for Blocks of Statements

In some situations, users may want to specify requirements that are valid for a *block of statements* (or commands) instead of for a single command.

Consider, for instance, a certain report that should be built in five minutes. But such report is based in two database queries. Therefore, both queries should finish within the specified deadline or none of them should be executed. In such situation, the user may specify the deadline for the *block of statements* (or *block of commands*).

Definitions

Let $Q = \{q_1, q_2, ..., q_n\}$ denote a set of user statements (or commands), with q ranging on Q. Let $R = \{r_1, r_2, ..., r_n\}$ denote a set of data access requirements with r ranging on R. A Block of Statements (or Commands) with Requirements is a set of user statements Q that share the same set of data access requirements R. Statements defined in a Parallel Block of Statements may be executed in parallel. Statements defined in a Sequential Block of Statements must be executed in the same order as they are defined.

SQL Extensions

In order to enable the specification of a *Block of Statements with Requirements*, we propose the use of the key expressions `BEGIN BLOCK` and `END BLOCK`. The clauses `PARALLEL` and `SEQUENTIAL` are used to specify if the commands defined in the block should be executed in parallel or sequentially. The block's requirements specification area is placed after the `END BLOCK` clause, like it is represented in Figure 7.

```
BEGIN [PARALLEL|SEQUENTIAL] BLOCK
     Command_1
     Command_2
     …
     Command_N
END BLOCK
REQUIREMENTS Requirements_Area_Name
     Requirement_Definition_1,
     Requirement_Definition_2,
     ...
```

Figure 7 - SQL Extensions – Blocks of Statements with Requirements

When executing the statements that are in the block, the system should satisfy all the requirements in the block's requirements specification area. But each statement may have its own requirements, which are specified in the statements' requirements specification area.

Example 3.3 Figure 8 presents an example on the use of a parallel block of statements. In such figure, several requirements are specified (types of requirements are defined in Section 3.2). The block presented in Figure 8 has two queries; each of them has a *Data Availability* and a *Disconnected Execution Mode* requirement. Both queries should be executed even though the user that submitted them is disconnected from the database system. The result set of the execution of the first query should be named `TOP_CUSTOMERS`. The result set of the execution of the second query should be named `REVENUE_PER_STATE`. The result set of the execution of each query should be available for users during 10 minutes after the end of the corresponding query's execution. Besides that, the entire block has a *Data Freshness Requirement*, an *Execution Finish Time Requirement* and an *Execution Periodicity Requirement*. Both

queries inside the block may use a replica of the STATES relation which is 72 hours old, should have its execution finished at 14 o'clock and should be executed every Friday during July, 2010.

```
BEGIN PARALLEL BLOCK

 SELECT C.ID, C.NAME, C.PHONE
 FROM CUSTOMERS C
 WHERE EXISTS (SELECT 1
               FROM SALES S
               WHERE S.CUSTOMER_ID = C.CUSTOMER_ID
                 AND S.REVENUE > 1000
                 AND S.DATE > '2010/01/01')
 REQUIREMENTS REQ1
   EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS TOP_CUSTOMERS
   AVAILABLE DURING 10 MINUTES AFTER EXECUTION;

 SELECT S.STATE_ID, ST.STATE_NAME, SUM(S.REVENUE) AS SUM_REVENUE
 FROM SALES S
     INNER JOIN STATES ST
     ON S.STATE_ID = ST.STATE_ID
 WHERE S.DATE > '2010/01/01'
 GROUP BY S.STATE_ID, ST.STATE_NAME
 REQUIREMENTS
   EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS REVENUE_PER_STATE
   AVAILABLE DURING 10 MINUTES AFTER EXECUTION;

END BLOCK
REQUIREMENTS REQ_BLOCK
 FRESHNESS OF STATES HIGHER THAN '2010/07/01',
 FINISH BEFORE '14:00',
 REPEAT EVERY FRIDAY IN PERIOD FROM '2010/07/01' TO '2010/08/01'
```

Figure 8 - SQL Extensions – Blocks of Statements with Requirements – Example

3.1.3 Dropping Requirements

Requirements change over time. Therefore, there should be a way for users to inform the database system that a certain requirement should not be fulfilled anymore.

In order to cancel the execution of commands that have associated DARs, we propose the use of the DROP REQUIREMENTS command. Figure 9 represents the syntax of such command, which accepts as parameter the *Requirements Area Name* of the command whose execution the user wants to cancel.

```
DROP REQUIREMENTS Requirements_Area_Name
```

Figure 9 - SQL Extensions – Dropping an Execution Periodicity Requirement

Example 3.4 In the example of Figure 8, the requirements area of the block of commands is named as `REQ_BLOCK`. If the user wants to cancel the execution of such block of commands before August 1st, 2010, then he/she should execute the `DROP REQUIREMENTS` command specified in Figure 10.

```
DROP REQUIREMENTS REQ_BLOCK
```

Figure 10 - SQL Extensions – Dropping a Requirement - Example

Database objects can also be altered in order to remove specified requirements. We propose the use of the `ALTER` command together with the `DROP REQUIREMENTS` clause.

Example 3.5 In Figure 11, specified requirements are removed from table `CUSTOMERS`.

```
ALTER TABLE CUSTOMERS
DROP REQUIREMENTS
```

Figure 11 – Dropping a Requirements - ALTER TABLE Command – Example

3.2 Data Access Requirements: Definitions and SQL Extensions

In this section, we define a set of possible types of DARs and propose how they can be specified in SQL. Some of proposed DARs are especially useful in distributed databases (e.g. Replica Age) but most can also be used in centralized and parallel environments. The proposed types of DARs are:

- Data Freshness Requirement – defines the acceptable *timestamp* that a certain replica must correspond to in order to be used to answer a query;
- Execution Deadline Requirement – defines an upper bound for the duration of a command's execution;
- Disconnected Execution Mode Requirement – specifies that a certain command should be executed even though the user who specified the requirement is no longer connected to the database. This allows users to schedule for later use;
- Data Availability Requirement – defines a time period over which a certain dataset should be available to users;
- Execution Periodicity Requirement – defines that a certain command must be executed in some time windows in a determined time period;

- Execution Finish Time Requirement – defines a timestamp on which the command execution should already be finished;
- Execution Start Time Requirement – defined a lower bound on the time that a certain command may have its execution started;
- Execution Priority Requirement – defines the execution order priority of distinct statements (considering that the system would also satisfy the DARs it had already committed itself to satisfy).

In the following, we formally describe the abovementioned set of types of DARs for database operations and propose some extensions to SQL language that would enable users to specify such DARs using SQL.

3.2.1 Data Freshness Requirement

In databases, data replication can improve query execution performance and data availability. However, in most situations it is not possible to use synchronous replication, which means that some data replicas are not up to date. Users may choose to do not use up to date data in order to improve query execution performance or even to execute a query (when up to date data is not available).

The *Data Freshness Requirement* specifies a filter to the system about which data replicas may be used to answer a certain query.

Definitions

Let $R = \{r_1, r_2, \ldots, r_n\}$, with r ranging on R, denote a set of replicas of a certain data set D. Consider that the data of r_i is matched with the that existed in D at a certain timestamp T_i. The Data Freshness Requirement (ω) specifies a timestamp that is used as a lower bound for T_i. Therefore, a certain data replica r_i can only be used to answer a query when:

$$\omega \leq T_i$$

SQL Extensions

We propose an SQL extension that enables the specification of the Data Freshness Requirement for each relation used in a query command. Such extension is the `FRESHNESS` clause used in Figure 12 (where *relation* is the relation name and *freshness_parameter* is the requirement's value).

```
FRESHNESS OF relation HIGHER THAN freshness_parameter
```

Figure 12 - SQL Extensions – Data Freshness Requirement

Example 3.6 In the example of Figure 13, a user specifies that the system can execute the query using any replica of the `SALES` relation whose data corresponds at least to the data stored in the master `SALES` table in December 1st, 2010.

```
SELECT P.PRODUCT_ID, PRODUCT_NAME, SUM(REVENUE)
    FROM SALES S
          INNER JOIN PRODUCTS P
    ON S.PRODUCT_ID = P.PRODUCT_ID
    WHERE DATE >= '2010/06/01'
    GROUP BY P.PRODUCT_ID, PRODUCT_NAME
REQUIREMENTS
    FRESHNESS OF SALES HIGHER THAN '2010/12/01'
```

Figure 13 - SQL Extensions – Data Freshness Requirement - Example

3.2.2 Execution Deadline Requirement

In many situations, users may need to finish a command's execution in a certain time period. For instance, a user may need a certain report for a briefing that would happen in a few minutes, which would impose a deadline for the execution of report's queries.

An *Execution Deadline Requirement* specifies a timestamp interval by which the execution of a statement should be finished.

Definitions

Let q denote a user statement which takes a certain time (t) to be executed. Let s represent the timestamp on which q's execution starts. Let t_0 represent the timestamp on which q is submitted to the system. The Execution Deadline Requirement (δ) of q represents a timestamp interval on which a user needs the execution of q to finish. The Execution Deadline Requirement is satisfied when:

$$t_0 + \delta \geq (s_q + t_q)$$

SQL Extensions

We propose a `DEADLINE` clause, which accepts a *deadline_parameter* that represents the maximum acceptable duration of command's execution (in seconds). In Figure 14 we present the syntax of the `DEADLINE` clause.

```
DEADLINE deadline_parameter
```

Figure 14 - SQL Extensions – Execution Deadline Requirement

Example 3.7 The sample query shown in Figure 15 must be completed in no more than two minutes after its submission to the system.

```
SELECT PRODUCT, SUM(REVENUE)
 FROM SALES
 WHERE DATE >= '2010/06/01'
 GROUP BY PRODUCT
REQUIREMENTS
 DEADLINE 120
```

Figure 15 - SQL Extensions – Execution Deadline Requirement – Example

3.2.3 Disconnected Execution Mode Requirement

When users or applications submit commands to databases, they usually stay connected to the system waiting for the results of command's execution. This is the default behavior of the system. Nevertheless, sometimes users or applications may want to submit a command and disconnect from the database. In such situation, the user would only connect again to obtain command's results after sometime (when he/she believes that command execution is already finished).

For instance, consider a manager who needs a report, which is based on a certain query or set of queries. He is about to leave home and wants the report to be finished when he arrives at his office (10 minutes latter). Then, just before leaving home, he submits the report for execution using his cell phone and establishes an execution deadline of 10 minutes. The manager disconnects from the database after submitting the report and connects again only after he arrives at this office. Then, he would query for his report.

The *Disconnected Execution Mode Requirement* specifies that the database command should be executed even though the user who submitted it is disconnected from the system.

Definitions

Let q denote a user statement and u denote the user who submitted the statement for the system. Let r denote the result of the execution of q. The Disconnected Execution Mode Requirement is satisfied when the system executes q and makes r available for database users even though u is disconnected from the system.

SQL Extensions

We propose the use of the `EXECUTE DISCONNECTED` clause in order to specify the Disconnected Execution Mode Requirement. Figure 16 presents the full syntax of such clause: the *DatasetName* represents the name of the object on which the system would store command's results.

```
EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS DatasetName
```

Figure 16 - SQL Extensions – Disconnected Execution Mode Requirement

Example 3.8 Consider a manager who uses a cell phone application to submit a report for execution when leaving home and who wants to receive the report when arriving at the office 10 minutes later. Figure 17 presents an example of the command submitted by the cell phone application. This is a query with two requirements: *execution deadline* and *disconnected execution mode*. Such query should be executed in 10 minutes even though the user is disconnected from the system and its results should be identified as `REVENUE_PER_STATE`.

```
SELECT STATE, SUM(REVENUE)
 FROM SALES
 WHERE DATE >= '2010/01/01'
 AND STATUS IN (0,1)
 AND NOT EXISTS (
     SELECT 1
     FROM REFUNDS
     WHERE STATUS = 0
     AND REDUNDS.SALE_ID = SALES.SALE_ID)
 )
 GROUP BY STATE
REQUIREMENTS
 DEADLINE 600,
 EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS REVENUE_PER_STATE
```

Figure 17 - SQL Extensions – Disconnected Execution Mode Requirement – Example

In Figure 16, the *DatasetName* represents the name of the object that would store the results of user's command. When the user or application wants to retrieve the command's results, he/it would query the database for the *DatasetName*, as presented in Figure 18.

```
SELECT * FROM DatasetName
```

Figure 18 - SQL Extensions – Retrieving the Results of a Disconnected Executed Command

Example 3.9 In order to retrieve the results of the query of Figure 17, the application may submit the command of Figure 19 (i.e. query `REVENUE_PER_STATE`).

```
SELECT * FROM REVENUE_PER_STATE
```

Figure 19 - SQL Extensions – Retrieving the Results of a Disconnected Executed Command - Example

It is expected that reports that would be executed in disconnected mode would produce result sets that are significantly smaller than the data sets the computations access. Therefore, retrieving the results of a disconnected executed command is not usually very time consuming. Besides that, as such results set has just being generated, the system may choose to maintain it in memory for a certain time. But, users would only have a guarantee that the results set would be available for a certain time if they specify a *Data Availability Requirement* (which we describe in the following).

3.2.4 Data Availability Requirement

In several situations, users may need to have a guarantee that certain datasets are available. Users may want to make the result set of a certain time-consuming query available for other users during a certain time; In distributed databases, some sites may become offline during certain time periods, making the data they store inaccessible for remote users.

The *Data Availability Requirement* specifies a period over which a dataset should be available to users.

Definitions

Let U denote a set of system's users with u ranging on U. Let D denote a dataset and $T=\{t_1,t_2,\ldots,t_n\}$ denote a set of time windows (e.g. every Friday of a certain month) with t ranging on T. The Data Availability Degree ($A_{D,T}$) is the interval of T on which D is actually available to U. The Data Availability Requirement ($\delta_{D,T}$) is the interval of T on which D should be available to U. Thus, δ is a lower bond for A ($\delta_{D,T} \leq A_{D,T}$).

When a user specifies a Data Availability Requirement for a certain piece of data during a certain set of time windows, he/she expects that the data availability "degree" of such piece of data during the specified set of time windows be higher or equal to the specified requirement.

In order to satisfy a Data Availability Requirement, a QoE-oriented system dynamically creates and places data or replicas. Such data replicas may be materialized in disk or not. For instance, if a user wants to guarantee that a certain query result is available during a small time period, he/she can specify an availability period and the system may choose to provide the desired availability using the main memory or not.

SQL Extensions

We propose an `AVAILABILITY` clause in order to identify a data availability requirement. The definition of a data availability requirement must include the desired min-value for the data availability degree.

Figure 20 presents the syntax of the `AVAILABILITY` clause. Such clause enables users to specify the min-value for the data availability degree in minutes (using the *Minutes* parameter of Figure 20) or as a percentage of a certain time window (using the *Percentage* parameter).

In Figure 20, *Repeating* and *InPeriod* are used to specify the set of time windows T over which the defined availability degree should be provided. *InPeriod* specifies date and time boundaries for T. In case T is composed by several periods, which happens with some periodicity, *Repeating* should be used (repeating is defined later on).

In order to guarantee the desired availability degree, the system may choose to dynamically create data replicas. Users may indicate that the data that should be available is the one that exists when the command is submitted (option `SNAPSHOT`), that master data and replicas updates may occur asynchronously (option `ASYNCHROUNOUS`) or that data master data and data replicas must always be synchronized (option `SYNCHROUNOUS`).

```
AVAILABLE DURING [Period MINUTES]|[Percentage PERCENT]
     [Repeating][InPeriod][CollectData]

Repeating = [EVERY MONTH FROM BegiDay TO EndDay]|
   [EVERY BeginDayOfWeek TO EndDayOfWeek]|[EVERY DayOfWeek]

InPeriod = IN PERIOD FROM BeginDateTime TO EndDateTime

CollectData = [SNAPSHOT|ASYNCHRONOUS|SYNCHRONOUS]
```

Figure 20 - SQL Extensions - Data Availability Requirement

Example 3.10 Figure 21 presents some commands that use the `AVAILABILITY` clause. In the first one, a query has a 5 minutes execution deadline and is executed in disconnected mode. Query's results are stored in `REVENUE_PER_PROD`, which should be available during 20 minutes after query execution. In command II of Figure 21, a snapshot of table `CUSTOMERS` should be available during 99% of the time between January 1, 2010 and March 31, 2010. In command III, the `TOP_SALES` table should be available every month from days 25 to 30. Data replicas may not be fully synchronized with master tables.

```
 I) SELECT PRODUCT, SUM(REVENUE)
      FROM SALES
      WHERE DATE >= '2010/01/01'
      GROUP BY PRODUCT
    REQUIREMENTS
      EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS REVENUE_PER_PROD,
      DEADLINE 300,
      AVAILABLE DURING 20 MINUTES

II) ALTER TABLE CUSTOMERS
    ADD REQUIREMENTES
      AVAILABLE DURING 99 PERCENT
      IN PERIOD FROM '2010/01/01'
            TO '2010/03/31'
      SNAPSHOT

III)ALTER TABLE TOP_SALES
    ADD REQUIREMENTS
      AVAILABILITY DURING 100 PERCENT
      EVERY MONTH FROM 25 TO 30
      ASYNCHRONOUS
```

Figure 21 - SQL Extensions - Data Availability Requirement – Examples

3.2.5 Execution Periodicity Requirement

Consider an organization in which every Friday there is a board meeting with the managers. In such meetings, some reports about sales are presented to the managers. Reports must be based on the most up to date data. Therefore, reports' queries are built using an *Execution Finish Time Requirement* (defined in Section 3.2.6) that specifies an execution finish time for the database commands. But such reports must be built every week. Hence, reports' queries may also have an *Execution Periodicity Requirement* which specifies the periodicity on which the database command should be executed.

Definitions

Let q denote a user statement and $T=\{t_1,t_2,\ldots,t_n\}$ denote a set of time windows (e.g. every Friday of a certain month) with t ranging on T. The Execution Periodicity Requirement (τ) is the set of time windows (τ C T) on which q should be executed.

SQL Extensions

We propose a `REPEAT` clause in order to identify an Execution Periodicity Requirement, as presented in Figure 22. Such clause accepts the time windows identification using the same *Repeating* and *InPeriod* clauses that we used in the Data Availability Requirement: *Repeating* specifies the frequency on which the command execution should be repeated and *InPeriod* specifies date and time boundaries for such executions.

```
REPEAT Repeating InPeriod

Repeating = [EVERY MONTH FROM BegiDay TO EndDay]|
   [EVERY BeginDayOfWeek TO EndDayOfWeek]|[EVERY DayOfWeek]

InPeriod = IN PERIOD FROM BeginDateTime TO EndDateTime
```

Figure 22 - SQL Extensions – Execution Periodicity Requirement

Example 3.11 Figure 23 presents an example on the use of the *Execution Periodicity Requirement*. Consider a managers' meeting that takes place every Friday and on which some reports about sales are presented to the managers. The query of Figure 23 may be used to construct one of the reports presented in such meetings: it is executed in every Friday of the first semester of 2010 with an execution finish time of 14 o'clock.

```
SELECT STATE, SUM(REVENUE)
 FROM SALES
 WHERE MONTH(SALES.DATE) = MONTH(SYSDATE)
 AND YEAR(SALES.DATE) = YEAR(SYSDATE)
 AND STATUS IN (0,1)
 AND NOT EXISTS (
     SELECT 1
     FROM REFUNDS
     WHERE STATUS = 0
     AND REDUNDS.SALE_ID = SALES.SALE_ID)
 )
 GROUP BY STATE
 REQUIREMENTS REQUIREMENTS_REPORT_1
     EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS MANAGER_REPORT,
     FINISH BEFORE '14:00',
     REPEAT EVERY FRIDAY IN PERIOD FROM '2010/01/01' TO '2010/07/01'
```

Figure 23 - SQL Extensions – Execution Periodicity Requirement – Example

3.2.6 Execution Finish Time Requirement

Users may need to have the execution of a certain statement or block of statements by a certain time. For instance, consider that a certain report may be executed at night, no matter when, since its execution is finished before 08AM from the next day. In this situation, users may specify an *Execution Finish Time Requirement*.

The *Execution Finish Time Requirement* indicates when the execution of a certain statement or block of statements should be finished.

Definitions

Let q denote a user statement, which takes a certain time (t) to be executed. Let s represent the timestamp on which q's execution starts. The Execution Finish

Time Requirement (φ) of q represents a timestamp on which the execution of q must have finished. The Execution Finish Time Requirement is satisfied when:

$$\varphi \geq (s + t)$$

SQL Extensions

We propose a `FINISH BEFORE` clause, which accepts a *finish_time_parameter* that represents the timestamp on which query's execution should already have finished. In Figure 14 we present the syntax of the `FINISH BEFORE` clause.

```
FINISH BEFORE finish_time_parameter
```

Figure 24 - SQL Extensions – Execution Finish Time Requirement

Example 3.12 The sample query shown in Figure 15 must be completed before 17:00 hours.

```
SELECT PRODUCT, SUM(REVENUE)
 FROM SALES
 WHERE DATE >= '2010/06/01'
 GROUP BY PRODUCT
REQUIREMENTS
 FINISH BEFORE 17:00
```

Figure 25 - SQL Extensions – Execution Finish Time Requirement – Example

3.2.7 Execution Start Time Requirement

Users may want to a certain report to be executed while they are disconnected from the system. However, in such situation, report execution may have a certain restriction on the timestamp on which the report execution is allowed to start. The *Execution Start Time Requirement* specifies the timestamp on which a certain command execution is allowed to start.

Definitions

Let q denote a user statement whose execution starts at timestamp s. The Execution Start Requirement of q (σ_q) represents the timestamp on which report execution is allowed to begin. The Execution Start Requirement of q is satisfied when:

$$\sigma_q \leq s$$

SQL Extensions

In order to express the Execution Start Time Requirement in SQL, we propose the `START AFTER` clause. Such clause accepts a *start_time_parameter*, which is the

lower bound (date and) time for command execution start. Figure 26 presents the syntax of the `START AFTER` clause.

```
START AFTER start_time_parameter
```

Figure 26 - SQL Extensions – Execution Start Time Requirement

Example 3.13 In Figure 27, we present an example on the use of the `START AFTER` clause. The query uses two requirements: *execution start time* and *disconnected execution mode*. Such query should be executed after 08PM and its results should be identified as `REVENUE_PER_STATE`.

```
SELECT STATE, SUM(REVENUE)
 FROM SALES
 WHERE DATE >= '2010/01/01'
 AND STATUS IN (0,1)
 AND NOT EXISTS (
     SELECT 1
     FROM REFUNDS
     WHERE STATUS = 0
     AND REDUNDS.SALE_ID = SALES.SALE_ID)
 GROUP BY STATE
REQUIREMENTS
 START AFTER 08PM
 EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS REVENUE_PER_STATE
```

Figure 27 - SQL Extensions – Execution Start Time Requirement – Example

3.2.8 Execution Priority Requirement

Statements may have distinct priorities, which should indicate that some of them should be executed before others. For instance, report queries submitted by the board members of an organization may have higher priority (and should be executed earlier) than the ones submitted by other members of the organization.

The *Execution Priority Requirement* indicates the execution priority of the considered command.

Definitions

Let q_i and q_j denote two user statements. Let ρ_i denote the Execution Priority Requirement of q_i and ρ_j denote the Execution Priority Requirement of q_j. In order to satisfy such requirements, q_i should be executed before q_j whenever $\rho_i < \rho_j$.

Each statement has a priority, which must be expressed in a scale. For simplicity, we use only two values for priority: normal priority and high priority.

When evaluating the execution of queries with DARs, the system must evaluate if such statements (or block of statements) are compatible with the ones that the system

already agreed to execute. This is also valid in the case of high priority statements or block of statements. For example, consider two queries: q_i and q_j. The query q_i has a deadline d_i, execution time e_i and low priority, and the query q_j is of high priority, has a deadline d_j ($d_j > 2 * d_i$) and execution time e_j ($e_j = d_j - d_i$). The system accepted query q_i before the submission of q_j. In order to satisfy the deadline requirement of both queries, the system should execute q_i before q_j. However, this schedule would not satisfy specified priority requirements, which indicate that q_j must be executed before q_i. Therefore, when the user submits q_j, the system verifies it cannot satisfy the specified priority requirement and informs the user.

SQL Extensions

We propose the use of a `HIGH PRIORITY` clause (Figure 28) to identify that a statement is of high priority. Statements that do not have such clause are of normal priority.

```
HIGH PRIORITY
```

Figure 28 - SQL Extensions – Execution Priority Requirement

Example 3.14 In Figure 29, query I has a high priority requirement, while query II is of normal priority.

```
I) SELECT PRODUCT, SUM(REVENUE)
     FROM SALES
     WHERE DATE >= '2010/06/01'
     GROUP BY PRODUCT
   REQUIREMENTS
     HIGH PRIORITY

II) SELECT PRODUCT, SUM(REVENUE)
     FROM SALES
     WHERE DATE >= '2010/06/01'
     GROUP BY PRODUCT
```

Figure 29 - SQL Extensions – Execution Priority Requirement – Examples

3.3 Conclusion

Quality of Experience is a measure on user's satisfaction when using a certain system or service. One way to increase user's satisfaction is to make the system behave the way the user needs (or expects) the system to do.

In this chapter, we presented the use of user-defined *Data Access Requirements* (DARs) in order to increase the levels of QoE provided by database systems.

We presented how to specify DARs to database commands, which may be single statements or blocks of statements. We also presented how to specify DARs for database objects (e.g. tables, result sets). We also proposed that users should be able to cancel requirements.

Then, we defined a set of types of DARs that cover the most common requirements in database operations. Proposed types of requirements are: *Data Freshness, Execution Deadline, Disconnected Execution Mode*, *Data Availability*, *Execution Periodicity, Execution Finish Time, Execution Start Time* and *Execution Priority*.

We also proposed some SQL extensions that enable users (e.g. programmers which include requirements in application programs) to specify each of the proposed types of DARs in SQL.

During this chapter, most database commands examples are data retrieval queries. One possible future line of work on this subject is to study which requirements should be defined concerning data insertion, deletion and updating commands. Some of the proposed types of requirements may also be used in those cases.

In the following chapter, we discuss tasks generation from users' commands and DARs.

4 Tasks and Task Level Requirements

In previous chapter, we proposed that database users should be capable to specify one or more Data Access Requirements (DARs) for statements or blocks of statements. Then, the system should evaluate if it can satisfy specified DARs and inform the user in case such DARs cannot be satisfied.

We call job to a logical unit that should only have any part of it executed if all its DARs can be satisfied. A job can be a block of statements with DARs or even a single command (when it is not part of a block of statements). The first step in QoE-oriented scheduling is to transform jobs into smaller units (tasks) whose execution may be better evaluated and scheduled by the system. In order to guarantee DARs' fulfillment, each task has one or more task level requirements. The number and types of tasks (and the corresponding requirements) generated to execute a job depends on several factors, including the types of DARs and the physical design of accessed data.

In this chapter, we discuss task generation and task level requirements specification. First of all, Section 4.1 make some basic definitions, presenting a formal definition for jobs and tasks, proposing schedulers architecture and discussing how such schedulers can be used in distinct database architectures. Section 4.2 discusses data placement alternatives. Then, in Section 4.3 we detail tasks generation. Section 4.4 presents task level requirements specification. Then, Section 4.5 contains several examples of task generation and task level requirements specification considering centralized, parallel and distributed database environments. Finally, in Section 4.6 we summarize the chapter and present some final comments.

The main contribution of this chapter is the mapping of user-defined requirements into task level requirements that are used scheduling command execution and placing data

4.1 Jobs, Tasks, Schedulers and Database System's Architecture

Users' commands and block of commands with DARs should be transformed into tasks that may have task level requirements.

Definitions

A job is a statement or a block of statements that should only have any part of it executed if the system satisfies all specified DARs associated to the job. Therefore, if the system cannot satisfy any of the DARs associated to any command participating in a job, then the system should not execute any part of

the job. In order to evaluate DARs fulfillment, the system may only consider a certain time window (e.g. the next thirty days).

A task is a logical unit that should be executed in order to complete a job. A job execution may comprise one or more tasks, which may be executed in parallel in some cases. When several tasks are generated for a single job, there may be tasks that depend *on others and tasks that* supply *others.*

Dependent *tasks are tasks whose execution that may not start until the end of the execution of the tasks on which they depend on.* Supplier *tasks are tasks whose execution must end to allow the execution start of the tasks that depend on them.*

A task's execution may have several task level requirements, which must be fulfilled in order to guarantee the fulfillment of user specified DARs.

Jobs and tasks are handled by two types of schedulers: *community* and *tasks scheduler*.

The community scheduler is responsible to manage job's execution, generating tasks and the corresponding requirements and determining where tasks will execute, while the tasks scheduler evaluates whether task level requirements can be satisfied or not, and schedules tasks' execution.

In a generic environment, there may be several data services (i.e. database management engines that may execute tasks). One approach is to use a single tasks scheduler to evaluate requirements fulfillment and schedule tasks execution in all available data services. Such scheduler would manage all available resources and be capable to estimate future conditions in all participating services. Besides presenting a limited scalability, such strategy also imposes that data services are *tightly coupled*, with a single scheduler that controls all available resources. In order to increase the system's scalability and to consider a more generic environment where heterogeneous data services may have a certain degree of autonomy, we consider that there is a task scheduler to manage the operations of each data service. Besides that, one or more community schedulers are used to interact with several data services. Such organization enables community and task schedulers to be used in centralized, parallel and distributed databases.

For instance, in a centralized database there is a single data service, which leads to a single task scheduler, and a single community scheduler is sufficient as well.

In a shared-nothing parallel architecture (e.g. cluster database), each node can be seen as a data service and have its own tasks scheduler, as represented in Figure 30. A single community scheduler may be used to interact with all the data services. Such scheduler may reside in any machine.

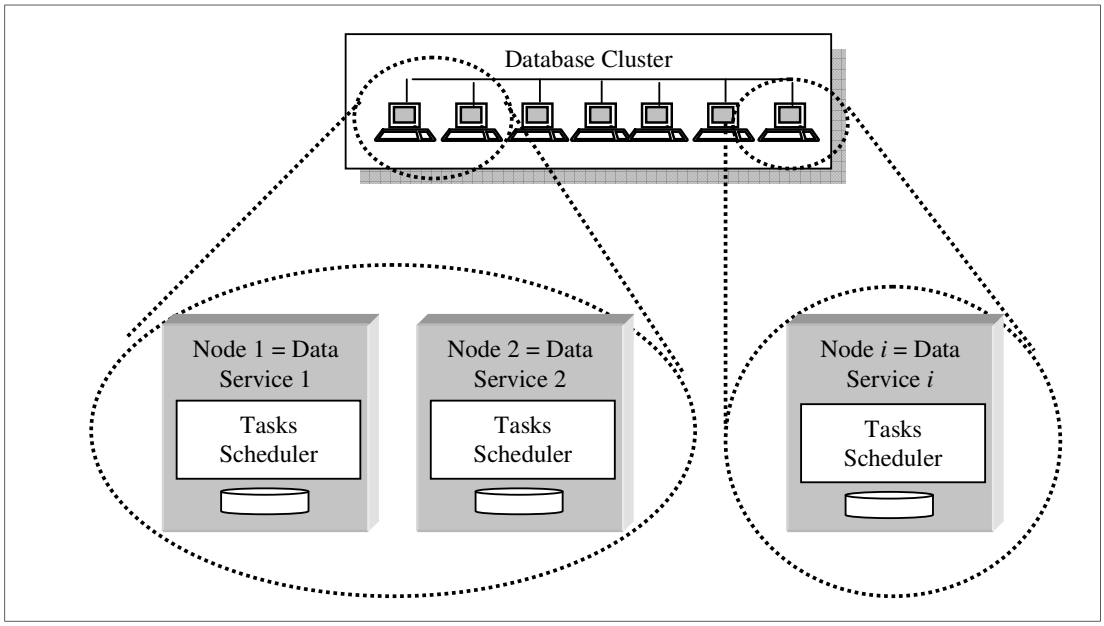

Figure 30 - Multiple Data Services in a Parallel Database System

Multiple data services can also be considered in distributed databases. In Figure 31, we present a globally distributed database, where each site is considered a single data service and has its own tasks scheduler. Task schedulers in a site may manage either a single or several nodes. A single community scheduler may be used in such situation, but the use of more than one can improve system scalability and availability (when using multiple community schedulers, they may share information about services' reputation in order to increase scheduling quality).

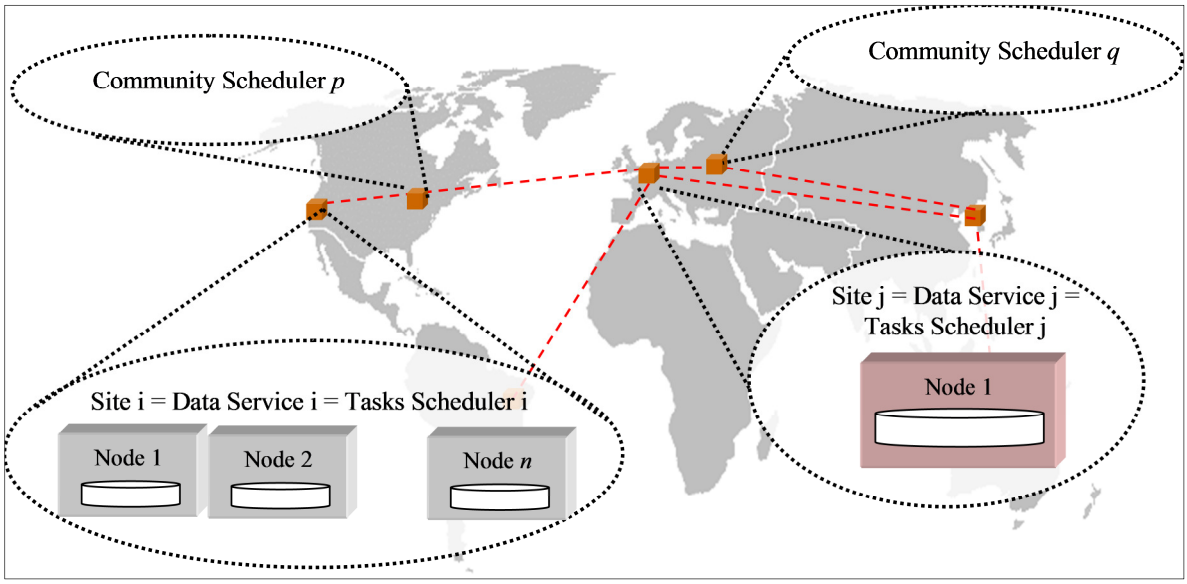

Figure 31 - Multiple Data Services in Global Databases

4.2 Data Placement Assumptions

Physical data placement is a key aspect of database systems and may influence in command execution performance and data availability. There are some alternatives strategies to physically place data, especially in the parallel and distributed database context.

One approach to physically place data in parallel/distributed environments is to distribute relations between available nodes. However, such approach commonly leads to low performance (as data movement between nodes is needed to do join operations). In fact, parallel/distributed database design commonly uses two primitive operations: replication and fragmentation. Such operations may also be combined in hybrid configurations.

The distributed data placement strategies we deal with are:

- Schema Replication – tables are fully replicated at distinct services. Queries can be entirely executed at a data service, with no need to do data shipping between services. On the other hand, it demands more space to store data (as all the data is placed in more than one place) and imposes some overhead to the system in order to maintain data replica synchronization;
- Schema Fragmentation (or Partitioning) – database tables are split into partitions considering pre-defined criteria. Table fragmentation may improve query execution performance and enable the use of intra-query parallelism (inter-query parallelism is provided by the database management system, which is capable to execute several commands concurrently).

  Table partitioning may be horizontal (where entire tuples of the relation are placed at distinct partitions) or vertical (where table's columns are placed at distinct partitions). Horizontal partitioning is the most commonly used strategy and is the one that may lead to the generation of several tasks for each job (as discussed in the following section). We treat vertical partitioning as if each partition is a distinct table and partition join operations must be explicitly defined in users' commands.

  → Subject-Based and Key-Based Multiple-table partitioning – in a schema with several tables, a common approach is to partition large or very large tables and replicate the other ones. In this case, all partitioned tables must follow the same partition criteria, which should be subject-based (e.g. tables that are geographically distributed, and where each table stores data about the events that occur in the region they are stored) or key-based partitioned (e.g. tables are partitioned considering the values of their *join attribute*);

  → Fragment Replication – besides creating fragments and replicating tables, it is also possible to replicate table fragments. This can simultaneously lead to some of the advantages of replication and fragmentation, like availability and intra-query parallelism [Costa & Furtado, 2009];

- Multiple Simultaneous Fragmentation Schemes – system administrators may simultaneously use more than one of the above described placement strategies. For instance, a certain table may be stored entirely stored in a single site, and simultaneously be key-based partitioned and have its partitions distributed placed across distinct services. This increases the available alternatives to the system when looking for providing high performance (and high QoE).

On the fly repartitioning may be necessary when tables that are partitioned considering distinct criteria are accessed by a single query. However, repartitioning is a very expensive operation. Therefore, we consider that multiple simultaneous fragmentation schemes are used in order to avoid the need of on the fly repartitioning.

4.3 Tasks Generation

Tasks generation from an SQL command concerns analyzing the submitted command, locating the necessary data to execute the command and writing new commands that should be executed in order to generate the result of the submitted command. Therefore, the following three situations should be considered when generating the tasks for a command:

I. The command accesses a schema that is not partitioned – a single task is generated for such command;

II. The command accesses a schema that is (subject or key)–based partitioned with *n* fragments – the command (query) must be transformed into a set of queries to operate over the individual fragments, as Furtado (2005) describes, and either a single or multiple commands to merge the partial fragment results [Furtado, 2005b]. The number of tasks generated for such command is therefore between n+1 and $2n - 1$ (n fragment operating queries and either 1 merge query or multiple merge query steps if hierarchical merge is used as described in [Furtado, 2005b]. Furtado (2005) discusses systematically the query clauses transformations that are necessary to operate over partitioned data, and the ones used to merge rewritten commands' results in order to obtain the original command's results set;

III. The command accesses a schema, which has multiple fragmentation schemes – several sets of tasks are generated, each one corresponding to a fragmentation scheme. The election-based task scheduling process (described in the next chapter) is used to choose which set of tasks should be executed.

Although tasks are generated considering existing data allocation schemes, a task can be executed by a data service that does not store all required data to execute task's query. In such case, missing data is copied during task execution.

Besides individual commands, users may also specify blocks of statements (as described in chapter 3) with DARs. The main difference between the use of blocks of statements and the use of individual commands is in the verification of DARs fulfillment: the system only accepts the block of commands for execution if all the tasks from all the commands can be executed while achieving all specified requirements.

Therefore, in case of blocks of statements, tasks are generated for each of the statements defined in the block in the same way they would be if the command was not included in a block. However, each task will have requirements generated from the command's DARs and from the block DARs.

Users may also specify alternative sets of requirements using the keyword `OR`. In this case, tasks are generated considering each set of requirements. The system would select among the alternative set of requirements the one that would be executed (described in Section 5.3).

Example 4.1 Consider a *sales* relation that is physically partitioned into several fragments that are placed across several distributed sites or across distinct nodes of a shared-nothing parallel machine. A user query that retrieves all rows from the *Sales* table must access all the physically distributed fragments.

Figure 32(a) presents an example of a user's SQL command that is defined over the distributed *sales* relation. Then, in order to execute such command, it is transformed into a set of tasks. Tasks are other SQL commands that can be executed at distinct sites: Figure 32(b) presents an example of a task's SQL that access the fragment *i* of the *sales* relation. The SQL command of Figure 32(b) should be executed at each of the data services that store a fragment of the *sales* relation. Figure 32(c) presents an example of the operation that should be executed in order to obtain the result of user's query (Figure 32(a)) from the results of tasks' queries (Figure 32(b)).

(a) User's command	(b) Task's SQL for site *i*	(c) Generating the resultset of user's command
Select product_group, sum(revenue) from sales where *conditions* group by product_group *DAR Specification*	*R*(*i*) = Select product_group, sum(revenue) from sales_i where *conditions* group by product_group	*Temp* = $\cup_i R(i)$; $1 \leq i \leq Number\ of\ sales\ relation's\ fragments$ *Resultset* = Select product_group, sum(revenue) from *Temp* where *conditions* group by product_group

Figure 32 - Task's generation and results merging – Example

4.4 Task Level Requirements Specification

User's commands and block of commands are transformed into one or more tasks and associated execution requirements. Table 2 lists transformations of DARs into task level requirements.

Table 2 - User Specified DARs and Task Level Requirements

Command's DARs	Task Level Requirements
When multiple tasks are generated for the same job	Task.ExecutionStartTime > *Necessary time to execute supplier tasks* (Tasks' executors are elected in the same order that the tasks should be executed – i.e. starting with tasks that does not depend on other tasks and ending on tasks that are not suppliers of any other tasks. After the executor of a task is elected, the community scheduler has a foreseen execution time of such task. Such foreseen time is used to define the *Execution Start Time* of the tasks that depend on the considered task)

Command's DARs	Task Level Requirements
`FRESHNESS OF ρ IS α`	Task.Relation(ρ).Freshness ≥ α
`START AFTER σ`	Task.ExecutionStartTime > σ
`DEADLINE δ`	*If the task has no dependent task (which includes tasks of single-task jobs) then*: Task.ExecutionDeadline = δ
`FINISH BEFORE φ`	Task.ExecutionFinishTime = φ
`EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS η`	Task.Results.StoreAtTemporaryRelation(η)
`AVAILABLE DURING τ MINUTES`	Task.TemporaryRelation(*relation_identifier*).AvailableTime = τ
`AVAILABLE DURING τ PERCENT`	Task.TemporaryRelation(*relation_identifier*).AvailabilityPercentage = τ
`IN PERIOD FROM τ1 TO τ2` (used in Data Availability DAR)	Task.TemporaryRelation(*relation_identifier*).AvailableFrom = τ1 Task.TemporaryRelation(*relation_identifier*).AvailableUntil = τ2
`SNAPSHOT`	Task.Results.StoreToFutureUse
`SYNCHRONOUS`	Task.RefreshMode = Synchronous
`ASYNCHRONOUS`	Task.RefreshMode = Asynchronous
`REPEAT EVERY [EVERY MONTH FROM` *`BegiDay`* `TO` *`EndDay`*`]| [EVERY` *`BeginDayOfWeek`* `TO` *`EndDayOfWeek`*`]| [EVERY` *`DayOfWeek`*`] IN PERIOD FROM` *`BeginDateTime`* `TO` *`EndDateTime`*	Task.ExecutionDate = *Date to execute the task* (A task is generated for each execution. Each task has its own execution date, which is assigned by the community scheduler when it generates the task)
`HIGH PRIORITY`	Task.ExecutionPriority = High
No DAR is specified	Task.ExecutionMode = ASAP

Most of transformations presented in Table 2 are straightforward (like the ones for Data Freshness and Execution Priority requirements).

In line 1 of Table 2, we present the *ExecutionStartTime* requirement specification for tasks that participate in multi-task jobs (i.e. jobs with several tasks). Such requirement is used in multi-task jobs. The value of such requirement is specified *on the fly*: the value for the requirement of a task *i* is specified after the election of the executors of all tasks that are suppliers of the i-*th* task, as described in Table 2. Task scheduling in such cases occur in *natural order,* that is, starting with supplier tasks and ending at dependent tasks.

Tasks generated for jobs with an Execution Deadline requirement (line 4 of Table 2) and that do not have dependent tasks should have an *ExecutionDeadline* requirement (therefore, such requirement is also used for tasks of single-task jobs that have an Execution Deadline DAR). However, the *general rule* of multi-task jobs is also valid and all the tasks should have *ExecutionStartTime* requirement, which is defined during the task execution assignment.

In the last line of Table 2, we present the requirement generated for database commands that have no DAR. Such tasks have a requirement that indicates that they should be executed *as soon as possible*.

In the following section, we present several examples of task generation and task level requirements specifications, considering centralized, parallel and distributed database environments.

4.5 Tasks and Task Level Requirements Specification: Examples

In the following, we present some examples of tasks generation and requirements specification.

First, in Section 4.5.1 we consider user commands and DARs submitted to a distributed database environment. We discuss the use of the Data Freshness, Execution Deadline, Disconnected Execution Mode and Data Availability Requirements.

Then, in Section 4.5.2 we discuss task and requirements generation for *best effort* commands in a parallel database system. We consider partitioned and replicated data.

Finally, in Section 4.5.3 we consider a centralized database and present examples of tasks and requirements specification for commands with Execution Start Time, Execution Finish Time, Execution Priority, Execution Deadline, Execution Periodicity, Disconnected Execution Mode and Data Availability requirements. We also present an example of tasks and requirements specification for a block of commands and for alternative sets of requirements.

4.5.1 Tasks and Requirements Generation in a Distributed Database System: Examples

Consider a global (possible *virtual*) organization which has a globally distributed database composed by several sites. Each site is considered as a data service and may have one or more nodes, as represented in Figure 33. Each data service has its own tasks scheduler, which manages site's resources.

Data service *i* stores a table named *Sales*, which stores the information about the company sales. The database at data service *i* is constantly being updated and stores huge volumes of data. In order to maintain high availability and to increase query execution performance, system administrators regularly replicate the *Sales* data to other sites. Nevertheless, as the volumes of data are too big, replica synchronization occurs just a few times per month and sales replicas are often outdated.

The following two examples consider such environment.

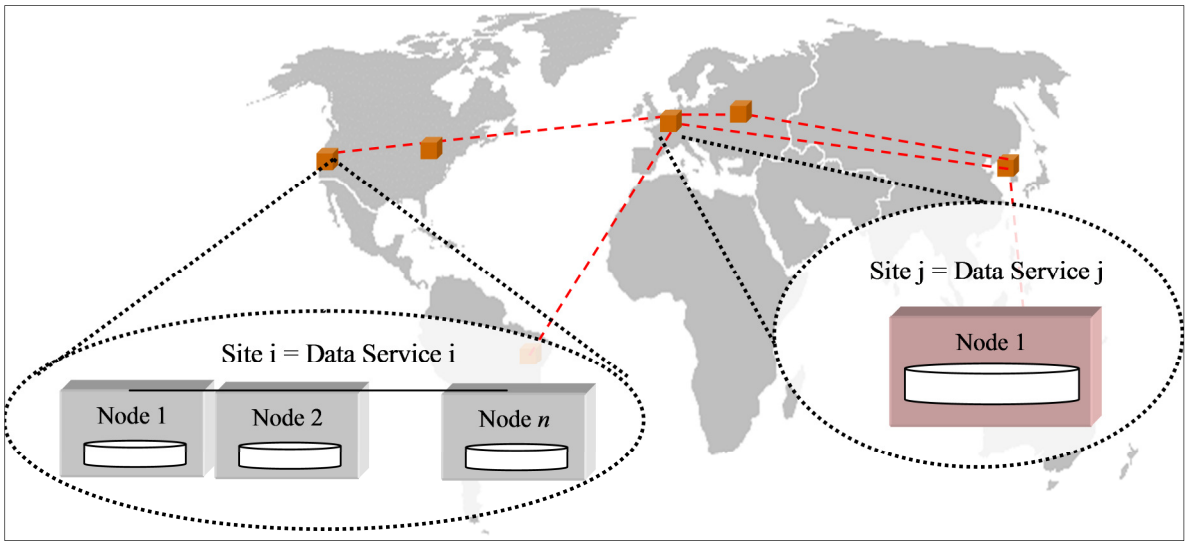

Figure 33 - Globally Distributed Data Services - Example

Example 4.2 At the middle of July 2010, a user must generate a certain report about revenues on the first semester of 2010. Such report must be presented in a briefing that would occur in less than 10 minutes. Although Sales replicas may be outdated, there is a great possibility that some of them already store the data about the entire first semester of 2010. Therefore, the user may submit a command using two data access requirements: Data Freshness Requirement and Execution Deadline Requirement. The user's command is represented in Figure 34, where an Execution Deadline of five minutes and a Data Freshness of July 1, 2010 are specified.

```
SELECT DATE, STATE, SUM(REVENUE)
 FROM SALES
 WHERE DATE BETWEEN '2010/01/01' AND '2010/06/30'
 GROUP BY DATE, STATE
REQUIREMENTS
 DEADLINE 300,
 FRESHNESS OF SALES HIGHER THAN '2010/07/01'
```

Figure 34 - User command with Execution Deadline and Data Freshness requirements – Example

The command represented in Figure 34 may be transformed into a single task with two requirements. Table 3 presents the values of each specified requirement (as a single task

is used, then the execution deadline of the task is similar to the one specified in the DAR and there is no need to specify a requirement on the task's start time).

Table 3 - Task Level Requirements for Execution Deadline and Data Freshness - Example

Command's DARs	Task Level Requirements
`DEADLINE 300`	Task.ExecutionDeadline = 300
`FRESHNESS OF SALES HIGHER THAN '2010/07/01'`	Task.Relation('Sales').Freshness ≥ '2010/07/01'

Example 4.3 Let us now exemplify the use of Data Availability and Disconnected Execution Mode requirements. Begging at January 2, 2011 and for the following two weeks, users from the entire organization need to build reports about the organization's revenue in 2010, even though site *i* becomes inaccessible for remote users.

In order to solve such situation, at January 1, 2011, a system administrator may replicate the data about all the sales in 2010 at all existing data services. However, this can be resource and time consuming. Alternatively, the system administrator submits the command of Figure 35 that has a Data Availability Requirement and a Disconnected Execution Mode requirement.

```
SELECT *
FROM SALES
WHERE YEAR(DATE) = 2010
REQUIREMENTS
     EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS SALES_OF_2010,
     AVAILABLE DURING 100 PERCENT
         IN PERIOD FROM '2011/01/02' TO '2011/01/15'
```

Figure 35 – Data Availability and Disconnected Execution Mode requirements – Example

In order to satisfy the requirements of such command, the query's results should be stored in a temporary relation named `SALES_OF_2010` (first requirement in Table 4). Second, third and fourth requirements in Table 4 impose restrictions on data availability, specifying the period in which the temporary relation should be available (which indicates, for instance, that the service cannot go down for scheduled maintenance during the specified period).

Table 4 - Task Level Requirements for Data Availability and Disconnected Execution Mode requirements - Example

Command's DARs	Task Level Requirements
`EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS SALES_OF_2010`	Task.Results.StoreAtTemporaryRelation('Sales_of_2010')

Command's DARs	Task Level Requirements
`AVAILABLE DURING 100 PERCENT IN PERIOD FROM '2011/01/02' TO '2011/01/15'`	Task.TemporaryRelation('Sales_of_2010').AvailableFrom = '2011/01/02' Task.TemporaryRelation('Sales_of_2010').AvailableUntil = '2011/01/15' Task.TemporaryRelation('Sales_of_2010').AvailabilityPercentage = 100

4.5.2 Tasks and Requirements Generation in a Parallel Database System: Examples

Consider a data warehouse stored at a shared-nothing parallel machine composed by off-the-shelf computers (i.e. database cluster). Each cluster's node is a data service. The warehouse follows a *star schema*, as the one represented in Figure 36, and has a big facts table (*Sales table*) and some small dimensions tables.

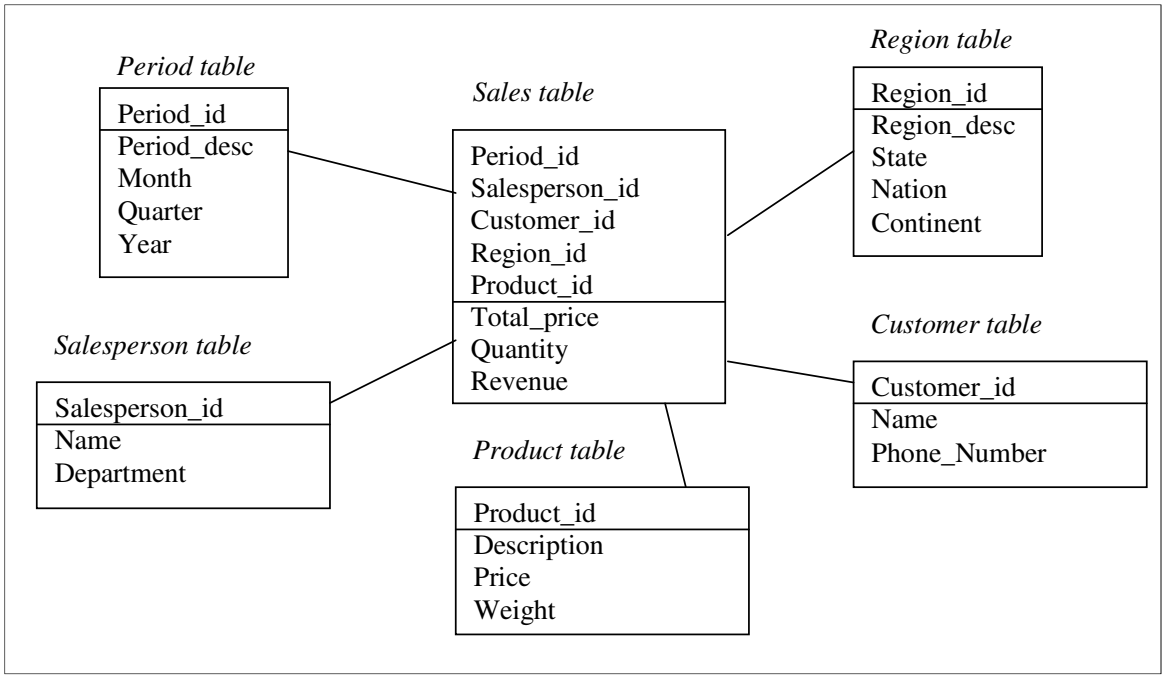

Figure 36 - Star Schema – Example

The Fact table is partitioned into 10 fragments, which are distributed across the cluster nodes. Each fragment may be replicated at several nodes. Dimensions tables are replicated at all nodes. Figure 37 represents such environment.

The examples of this section consider such environment.

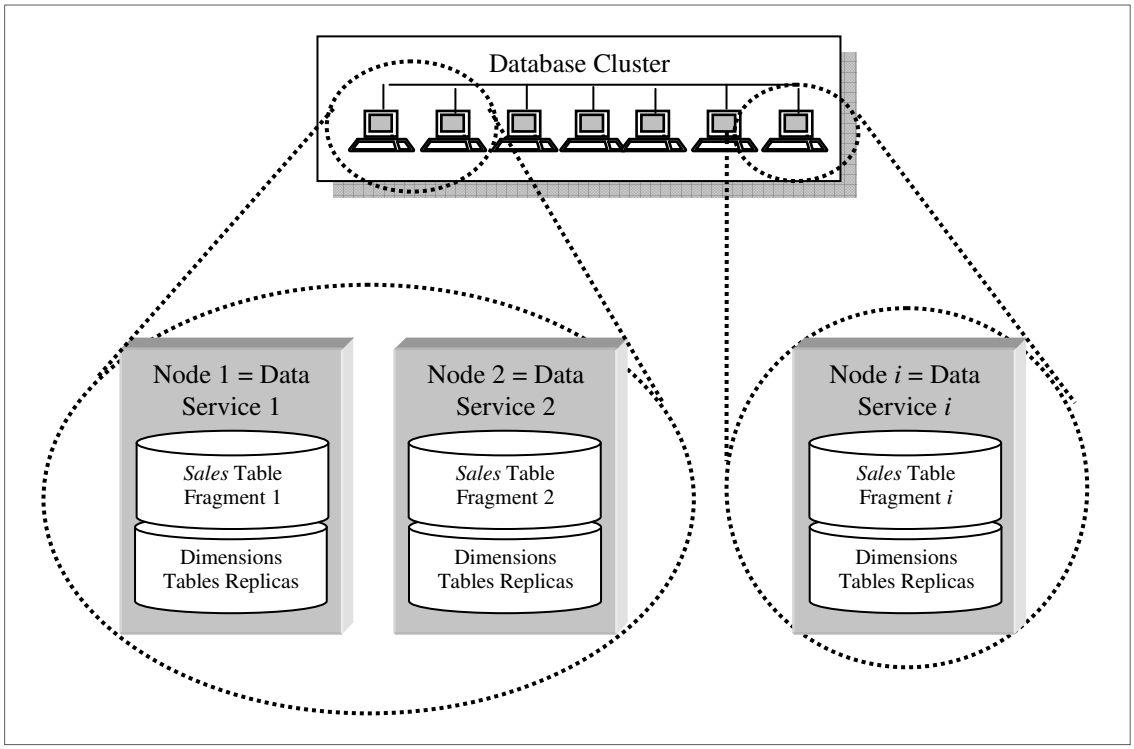

Figure 37 - Database cluster with fragmented and replicated tables

Example 4.4 The command of Figure 38 is a *best effort* command that accesses the Customer table. Each of the available data services stores a replica of such table. Therefore, any of the services is capable to execute the command, which may be transformed into a single task.

```
SELECT CUSTOMER_ID, NAME, PHONE_NUMBER
FROM CUSTOMER
WHERE NAME LIKE 'A%'
```

Figure 38 - User command that accesses a replicated table - Example

The command of Figure 38 is a best effort command (with no explicitly defined data access requirements). Tasks generated for such kind of commands have an *ExecutionMode* requirement, which indicates that the system must execute them as soon as possible (as represented in Table 5).

Table 5 - Task Level Requirements for Best Effort Oriented Query - Example

Command's DARs	Task Level Requirements
---	Task.ExecutionMode = 'ASAP'

Example 4.5 Consider that a user submitted the SQL command represented in Figure 39. Such command accesses the entire *Sales* table. But such table is fragmented and its

fragments are distributed across existing data services. Therefore, one possible way to execute such command is to generate eleven tasks. Each of the first ten tasks accesses a certain fragment of the Sales relation. Such tasks are executed in parallel. At the end of such tasks execution, another task is used to merge the results of the executed tasks in order to generate a single result set (as discussed in Section 4.3).

```
SELECT REGION_ID, MAX(REVENUE), MIN(REVENUE), AVG(QUANTITY)
FROM SALES
GROUP BY REGION_ID
```

Figure 39 - User command that accesses a fragmented table – Example

Table 6 presents the requirements generated for specified tasks. All tasks have the requirement that indicates that they must be executed *as soon as possible*. A requirement on the minimum value for the task's execution start time is specified for the final task (as the first ten may start in parallel and the last only starts when the first ten are finished). The value of such parameter is specified when the election for the executor of each of the first ten tasks is finished. Such value is the maximum estimated completion time between the ones of the first ten tasks.

Table 6 - Task Level Requirements for a Multi-Tasks Job - Example

Task Number	Command's DARs	Task Level Requirements
1 to 11	----	Task.ExecutionMode = 'ASAP'
11	----	Task.ExecutionStartTime > Foreseen time to execute tasks 1 to 10 – such value is estimated during the executor election of such tasks

4.5.3 Tasks and Requirements Generation in a Centralized Database System: Examples

Now consider a centralized database used by an OLTP (Online Analytical Processing) system. Consider that such database stores a *Sales* table (as in previous examples) with data about the sales that an organization does. Such table is continuously updated with new data (hundreds of rows per second during working hours). Such scenario is considered in the following three examples.

Example 4.6 Every working day, at 19h o'clock, there is a meeting of some of the organization managers where a report on the sales is presented. Such report is built considering the command of Figure 40. Such command accesses a single table stored at a centralized database and has some requirements.

```
SELECT PRODUCT_ID, CLIENT_ID, SUM(REVENUE), SUM(QUANTITY)
FROM SALES
WHERE DATE = SYSDATE
GROUP BY PRODUCT_ID, CLIENT_ID
REQUIREMENTS MANAGERS_REPORT
 START AFTER '18:30',
 FINISH BEFORE '18:55',
 EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS SALES_REPORT,
 AVAILABLE DURING 60 MINUTES,
 REPEAT EVERY MONDAY TO FRIDAY
```

Figure 40 - Command with several requirements - Example

The user who submitted the command of Figure 40 wants to receive the query's results at before 18:55hrs. He/she wants report execution to begin after 18:30hrs.

The user command may be transformed into a task with several requirements, which are represented in Table 7. Such task must be repeatedly executed every day from Monday to Friday. Therefore, the system may generate (and schedule) a set of tasks, each one for a distinct day (the *Execution Date* is different for every execution, as represented in Table 7). Such initially created set may comprise the period of the next 30 days. Every day, the system generates (and schedules) the task that would be executed in the thirtieth subsequent day.

Table 7 - Several Task Level Requirements - Example

Command's DARs	Task Level Requirements
`START AFTER '18:30'`	Task.ExecutionStartTime > 18:30
`FINISH BEFORE '18:55'`	Task. ExecutionFinishTime = 18:55
`EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS SALES_REPORT`	Task.Results.StoreAtTemporaryRelation('Sales_Report')
`AVAILABLE DURING 60 MINUTES`	Task.TemporaryRelation('Sales_Report').AvailableTime = 60
`REPEAT EVERY MONDAY TO FRIDAY`	Task.ExecutionDate = *Date to execute the task*

Example 4.7 Consider that, at the middle of the day, one of the organization's managers needs to urgently know the total revenue of that day. The SQL command of Figure 41 may be submitted to the QoE-oriented system. In such situation, the command can be treated as being of normal priority if its execution can finish in no more than 1 minute. Otherwise, the command must be treated as being of high priority.

```
SELECT SUM(REVENUE)
FROM SALES
WHERE DATE = SYSDATE
REQUIREMENTS
   (HIGH PRIORITY)
   OR
   (DEADLINE 60)
```

Figure 41 – Alternative set of requirements - High priority and execution deadline - Example

The command of Figure 41 accesses a single table stored at a centralized database. Therefore, it may be transformed into a single task. But alternative sets of requirements must generated (due to the use of the `OR` clause), as represented in Table 8.

Table 8 - Task Level Requirements for Execution Priority Requirement – Example

Set of requirements	Command's DARs	Task Level Requirements
I	`HIGH PRIORITY`	Task.ExecutionPriority = High
II	`DEADLINE 60`	Task.ExecutionDeadline = 60

Example 4.8 Consider the block of statement of Figure 42. Both queries of such block must be executed in a certain deadline, or none of them should be executed.

```
BEGIN PARALLEL BLOCK

 SELECT PRODUCT, SUM(REVENUE)
  FROM SALES
  WHERE DATE >= '2010/06/01'
  GROUP BY PRODUCT
 REQUIREMENTS
  EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS PRODUCT_REVENUE
  AVAILABLE DURING 10 MINUTES AFTER EXECUTION;

 SELECT STATE, AVG(REVENUE)
  FROM SALES
  WHERE DATE >= '2010/06/01'
  GROUP BY STATE
 REQUIREMENTS
  EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS STATE_REVENUE
  AVAILABLE DURING 10 MINUTES AFTER EXECUTION;

END BLOCK
REQUIREMENTS
 DEADLINE 60
```

Figure 42 - Block of statements with requirements – Example

The block represented in Figure 42 is transformed into two tasks: one corresponding to the first SQL query and another that corresponds to the second SQL query. Each task may be executed in parallel, as the block is a parallel block of statements. Table 9 presents the task level requirements of both tasks.

The community scheduler is responsible to guarantee that the system would only accept the block of statements (i.e. job) for execution if both tasks can be executed and have all their requirements satisfied.

Table 9 - Task Level Requirements for Tasks Generated for Blocks of Statements - Example

Task	Command's DARs	Task Level Requirements
I	EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS PRODUCT_REVENUE	Task.Results.StoreAtTemporaryRelation('Product_Revenue')
I	AVAILABLE DURING 10 MINUTES	Task.TemporaryRelation('Product_Revenue').AvailableTime = 10
I	DEADLINE 60	Task.ExecutionDeadline = 60
II	EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS STATE_REVENUE	Task.Results.StoreAtTemporaryRelation('State_Revenue')
II	AVAILABLE DURING 10 MINUTES	Task.TemporaryRelation('State_Revenue').AvailableTime = 10
II	DEADLINE 60	Task.ExecutionDeadline = 60

4.6 Conclusion

In this chapter, we defined jobs and tasks, and discussed the community and task schedulers, and how such schedulers can be used in centralized, parallel and distributed databases.

We detailed considered data allocation strategies and presented policies to do tasks generation and transformation of DARs into task level requirements. We also discussed issues related to task generation and requirements specification for blocks of statements.

Then, we presented several examples of task generation and task level requirements specification for DARs and considering both centralized, parallel and distributed databases.

Overall, this chapter explained how commands and DARs are transformed into tasks and their requirements for execution in data services. In the next chapter we discuss how data services are elected for execution and how reputation is used within the process of election.

5 Reputation and Election-Inspired Scheduling

In previous chapter, we discussed how users' commands and DARs can be transformed into tasks and task level requirements, which should be fulfilled in order to satisfy commands' DARs. In this chapter, we discuss task execution assignment to data services.

We propose an election inspired scheduling strategy. In this model, a set of services is defined as pre-candidates to execute a task. Then, each pre-candidate evaluates if it can satisfy task's requirements and makes promises on the necessary time to execute the task. The community scheduler elects a winner to execute the task, considering candidates' reputation and promises. If all tasks generated for a certain command have an elected executor, then the system accepts the command for execution. Otherwise, the system informs the user that the specified DARs cannot be satisfied and command execution is rejected.

The election inspired scheduling model maintains a certain degree of data service autonomy. Each data service may implement its own policy to decide on which elections it wants to participate. Local scheduling policies can be used to differentiate users and to do resource reservation. Heterogeneous environments (for instance, in terms of operational system, hardware architecture and database management system) can be used.

Another important issue in elections is reputation. When a task's requirement is not achieved, the system may fail to satisfy a DAR that it committed itself to satisfy. Therefore, task's requirements fulfillment is important to make the system dependable. But can a candidate break a promise?

In our proposed election-inspired scheduling strategy, we deal with candidates that break campaign promises after being elected (just as it happens in real-world elections). Candidates may make promises they cannot accomplish, intentionally or not. For instance, unintentional mistakes when estimating the execution time of a query or when foreseeing future conditions (e.g. in terms of availability of network resources) may lead to campaign promises that are unfeasible. Besides that, if data services receive some kind of incentive (e.g. monetary) to execute tasks, then some candidates may act maliciously and intentionally make a promise that they cannot accomplish just to win the election (and increase revenue). On the other hand, if there are any kind of penalties (e.g. fine) when a task's requirements are not fulfilled, then candidates may become too conservative when making promises (e.g. specifying a query execution time higher than the one the candidate had estimated as necessary). Such candidates would have some kind of *risk aversion*.

The election inspired scheduling strategy uses reputation information as part of the process of selecting an election winner. We consider the reputation as *an expectation about service's behavior based on its past behavior or on information about*

the service (which is similar to the definition of reputation used in [Abdul-Rahman & Hailes, 2000]).

Depending on the requirement's type, reputation is used to qualify the candidate (e.g. to help select the most trustful candidate) or to calibrate candidates promises (e.g. to help identify what is the difference between the candidate's promise and the candidate's act when elected). The reputation system helps to identify the best candidate to execute a task, both in terms of selecting the *best* promises and of choosing the most trustful candidate among existing ones. Therefore, the reputation system increases the system's dependability and leads to higher QoE levels, as it increases the possibility of assigning tasks execution to service providers that would satisfy all specified requirements, and avoiding those that miss many requirements.

The election-inspired scheduling differs from existing strategies because not only it deals with several types of user-defined data access requirements, but also because it also incorporates the reputation mechanism in order to improve the system's dependability when choosing the data service to execute each task. Therefore, our strategy can provide QoE-levels that other existing strategies cannot.

Section 5.1 presents the election-inspired scheduling mechanism, which considers task level requirements and services' reputation when assigning tasks' execution to data services. Then Section 5.2 discusses the use of election inspired scheduling for jobs with several tasks, presenting the use of *on the fly* elections to improve the level of QoE the system provides. In Section 5.3, we discuss the use of the election inspired scheduling strategy together with alternative sets of tasks generated for a single job. Then, in Section 5.4 we detail the evaluation of services' reputation on maintaining commitment to satisfy tasks. Section 5.5 details how the community scheduler estimates tasks' execution time. Section 5.6 presents the use of *what-if* elections to select which data can be replicated (and where it should be placed) in order to increase the level of QoE the system provides. Then, Section 5.7 discusses resource availability monitoring. Finally, Section 5.8 presents a chapter summary and some final comments.

The main contributions of this chapter are: (i) election inspired query scheduling model, (ii) reputation models for QoE-oriented database systems, and (iii) dynamic replica placement strategy for QoE.

5.1 Election-Inspired Scheduling for QoE-Oriented Databases

User's commands may have several Data Access Requirements (DARs). Jobs are transformed into tasks, which may have several associated requirements. We propose an election inspired strategy to assign task execution to data services. Figure 43 presents the main steps of the election inspired scheduling strategy. In Section 5.1.1 we detail the first step, pre-candidate definition. Then, Section 5.1.2 briefly discusses the campaign period, where services evaluate requirements and make promises. Section 5.1.3 details how the community scheduler elects the winner service.

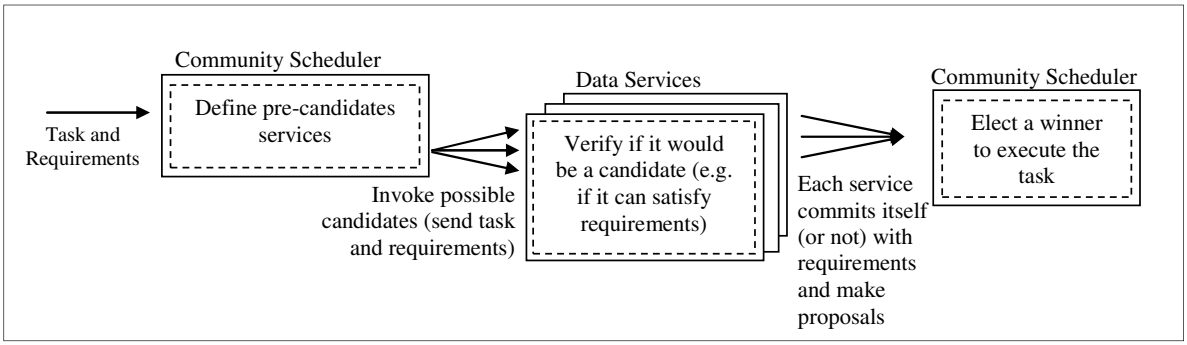

Figure 43 - Election inspired task scheduling: main steps

5.1.1 Defining Pre-Candidates

Election inspired scheduling begins (Figure 43) with the definition of the set of *pre-candidate* services that participate in the task's executor election.

A service can present itself *voluntarily* to be a pre-candidate or can be *selected* by the community scheduler as a pre-candidate. Pre-candidates (both the voluntarily presented and the selected by the community scheduler) are divided into levels: *first level pre-candidates*, *second level pre-candidates*, and so on.

In election inspired scheduling, the community scheduler starts the election considering the first level pre-candidates. If none of the first level pre-candidates agrees to be a candidate (i.e. execute the task while satisfying all its requirements), then the system invokes the pre-candidates of the second level. If none the second level pre-candidates agree to be a candidate then the system invokes the pre-candidates of the next level and so on, while there are services to be invoked. If no service presents itself as a candidate, then the job execution is canceled and the user is informed that the specified DARs cannot be satisfied.

Voluntarily Presented Pre-Candidates – A service presents itself voluntarily as a pre-candidate according to the policy defined by the services administrators. We propose the use of a load based policy that would make the service become a pre-candidate when its load is below a threshold value (we discuss such strategy in Chapter 6).

Voluntarily presented pre-candidates are classified into levels considering their reputation on maintaining its commitments (such reputation evaluation is detailed in Section 5.4). The system administrator should set the values that are used as threshold values to classify data services according to their reputation. Figure 44 presents the values for three levels of pre-candidates.

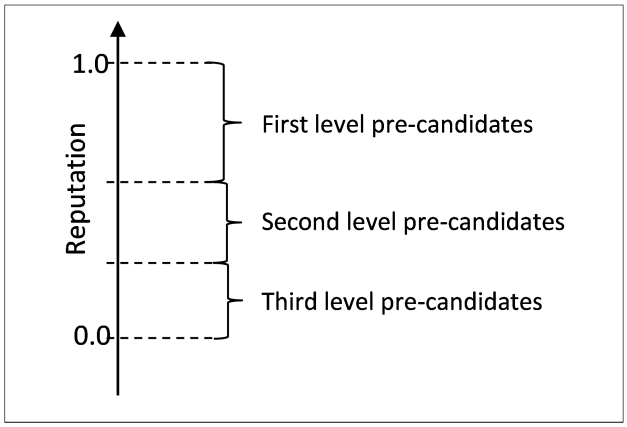

Figure 44 - Levels of Pre-Candidates – Voluntarily Presented Pre-Candidates

Pre-Candidates Selected by the Community Scheduler – A service is selected as a pre-candidate by the community scheduler if it has a high possibility to win the executor election (i.e. has a high reputation) and meets some requirements to execute the task (i.e. store most of required data to execute the task).

Therefore, in order to be selected as a pre-candidate by the community scheduler, a service must have high reputation on maintaining its commitments (detailed in Section 5.4) and store most of required data to execute a task, as represented in Figure 45. The lowest allowed reputation value and maximum allowed amount of data movement are parameters set by the system administrator.

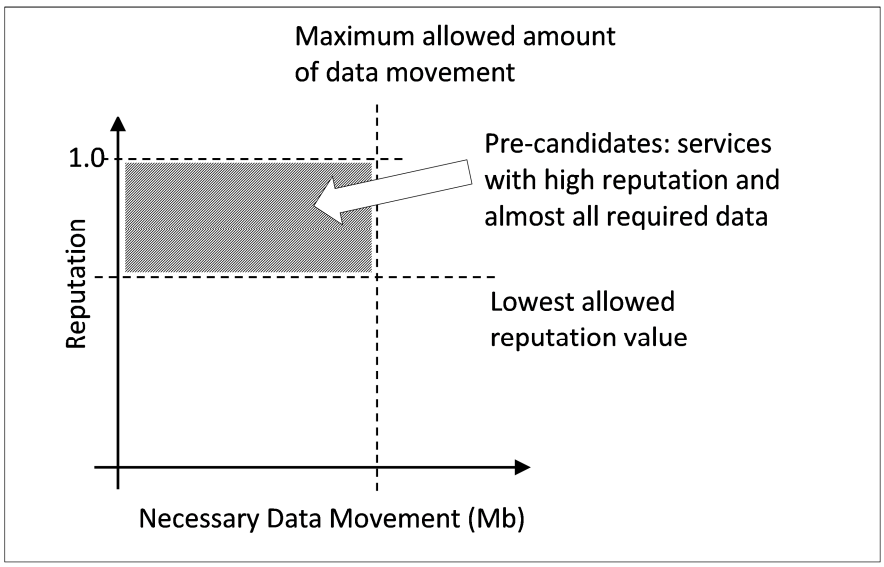

Figure 45 - Requirements to be a Selected as a Pre-Candidate

In fact, the system administrator should set several values to be used as threshold of reputation and amount of data movement, defining the levels of pre-candidates, as represented in Figure 46.

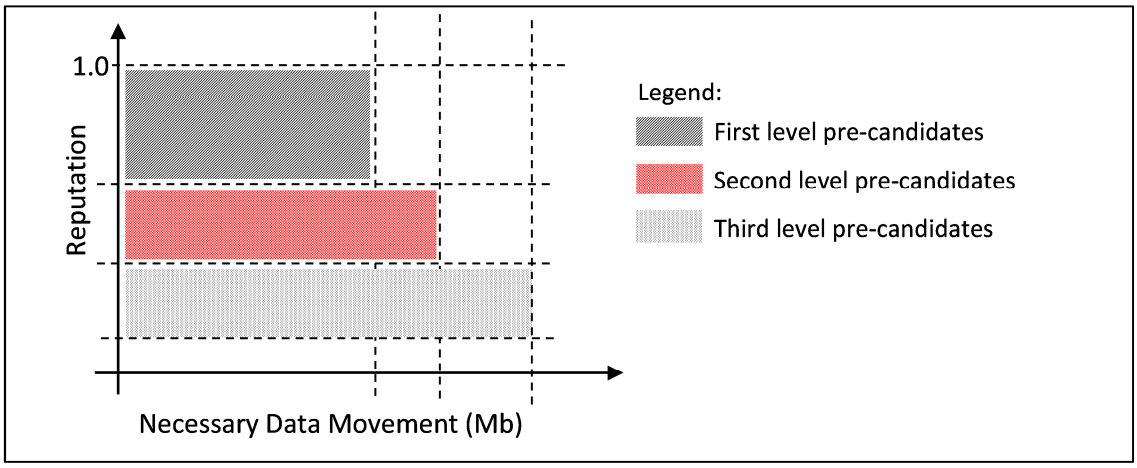

Figure 46 - Levels of Pre-Candidates – Pre-Candidates Selected by the Community Scheduler

Fixing a lower bound for candidates' reputation – in order to avoid that tasks execution are assigned to untruthful candidates, a lower bound for candidates' reputation must be specified by the system administrator.

If a service reputation falls below the minimal reputation allowed, then it is discarded as a candidate for some elections. After that, and in order to allow re-evaluation for potential increase in service's reputation, the service with low reputation is considered as pre-candidate for just a few elections. If service's reputation value remains smaller than the minimal allowed value, then the service is discarded as a candidate for some elections once again, and so on.

In Chapter 8, we present experimental results on the use of such mechanism. In our experimental evaluations, we used 0.9 as the lowest acceptable reputation value for pre-candidate services. Services whose reputation is below 0.9 participated only in 10% of elections.

Limiting the number of victories in sequence – service behavior may change over time. For instance, an external load (e.g. an antivirus system doing a full system scan) may transform a service that used to be trustful into one that fails the accomplishment of all tasks' requirements. But such change in behavior does not cause an immediate change in the service's reputation. Besides that, the higher the number of tasks being executed concurrently, more difficult it is to foresee execution times. In order to reduce the impact of such situations in the requirements fulfillment, it should be possible to limit the number of victories in sequence for a certain service.

When a service reaches the number of allowed victories in sequence, it is discarded from being a candidate for new elections during a certain time.

Using Availability Prediction – in order to improve the quality of the scheduling of tasks that are related to data availability, an availability prediction method is used: only services whose predicted availability for the considered period is equal or higher to the task's availability requirement can participate in the election as pre-candidate.

There are several works that address of predicting the availability of a resource during a certain period. Some of the works on resource availability prediction whose prediction methods can be used here are [Rahman, Hassan, & Buyya, 2010; Rood & Lewis, 2008].

5.1.2 The Campaign Period

Each pre-candidate evaluates if it would participate in the election or not. If the pre-candidate agrees to participate in the election, it becomes a *candidate* to execute the task and goes on campaign, committing itself to meet task's requirements and promising to execute the task within a certain time interval (intra-service task evaluation and management is detailed in Chapter 6).

The campaign period ends after a certain time or when all the candidates have already informed the community scheduler about their decisions on participating or not in the election. After the campaign period ends, the community scheduler elects a winner to execute each task.

5.1.3 Electing a Winner

When more than one service is candidate to an election, then the community scheduler must choose one service among the candidates to be the election's winner. The community scheduler assigns an *Election's Score* (*ES*) to each candidate. The candidate with the highest *ES* is selected as election's winner.

The ES of each candidate is computed considering two components: the *Normalized Reputation* (NRep) and the *Normalized Execution Completion Time* (NExecTime). Depending on the DARs of the original command, each of them may have more importance than the other. Therefore, we use two calibration factors: υ and ω ($\upsilon \in [0,1]$; $\omega \in [0,1]$; $\upsilon + \omega = 1$). The Election Score of the i-th candidate (ES_i) is computed by the following equation:

$$ES_i = \upsilon * NRep_{i,} + \omega * NExecTime_i$$

The value of each calibration factor (υ and ω) should be tuned by the system administrator in order to reflect the importance that the corresponding index has in the election. However, it is important to notice that reputation and completion time estimation have distinct importance depending on the DARs specified at user's job. For some types of DARs, the election winner should be the service that is most likely to satisfy task's requirements (i.e. the service with the highest reputation among the candidates). For other types of DARs, the scheduler should consider both the foreseen task's execution time and the service's reputation on satisfying specified requirements.

Table 10 summarizes possible criteria, presenting for which DARs only reputation is used and for which DARs the system considers both the candidates' reputation and runtime estimations.

Table 10 - Criteria to Select the Winner Candidate Considering DARs

Criteria Group	Criteria Group Description	Command's DAR
I	The winner is elected based on candidates' reputation (υ =1; ω = 0).	Execution Deadline (when not used together with Execution Priority or when the job generates a single task)
		Data Availability (when associated to database objects)
		Execution Start Time
		Execution Finish Time (when not used together with Execution Priority or when the job generates a single task)
II	The winner is elected based on candidates' reputation and on campaign promises (υ > 0; ω > 0), unless when DARs from this group are used together with DARs from Group I. If DARs from this group are used together with DARs from Group I, then the winner is selected based only on candidates' reputation (υ =1; ω = 0).	Execution Deadline (when used together with Execution Priority or when the job generates multiple tasks)
		Execution Finish Time (when used together with Execution Priority or when the job generates multiple tasks)
		Data Freshness
		Disconnected Execution Mode
		Data Availability (when associated to queries' results sets)
		Execution Periodicity
		Execution Priority
III	The winner is elected based on promises for the required time to execute the task (υ = 0; ω = 1). The candidate that promises to finish the task's execution earlier is the winner.	No DAR (*best effort* command)

For instance, when using a Data Availability Requirement with a database table (third line of third column in Table 10), the user expects the dataset to be available: in such situation, for each task generated to maintain data availability, the system should select the candidate with the highest reputation (foreseen execution completion time is of no importance). On the other hand, if a user submits a query without timing constraints, e.g using only a Data Freshness Requirement, he/she would like to receive the query's results as soon as possible (Data Freshness Requirement is placed in group II of Table 10). Therefore, campaign promises are used together with candidate's reputation in order to select a dependable candidate that provides a fast completion time.

In the case of a job with an Execution Deadline requirement that generates several tasks, supplier tasks should finish as soon as possible. Therefore, the fastest foreseen execution time is relevant when choosing the task's executor (group II of Table 10). The same assumption (i.e. fastest foreseen execution time is relevant during task execution assignment) is valid when a job has an Execution Deadline requirement and an Execution Priority requirement, because a job with such requirement should be executed *before low priority jobs* and *finish before the specified deadline.* On the other hand, when assigning for execution tasks generated from single-task jobs that have an Execution Deadline requirement and no priority-related requirements, then the system should assign the task execution for the job that is most likely to finish the execution before the specified deadline (no matter when, since it is before the deadline). Therefore, in such case, candidates' reputation is much more relevant than the foreseen execution time (group I of Table 10).

Normalized Reputation (NRep) – In order to compute the Normalized Reputation of a candidate, the community scheduler considers candidates' reputation on maintaining its commitment to satisfy task's requirements and the highest reputation value of election's candidates (candidates reputation assignment is discussed in Section 5.4).

Definition

Let HR be the highest reputation value on maintaining commitments to satisfy specified requirements of the candidates in a certain election. The Normalized Reputation of the i-th candidate ($NRep_i$) is computed as the relation between the reputation of the i-th candidate (R_i) and HR.

$$NRep_i = \frac{R_i}{HR}$$

Example 5.1 Consider three candidates *S1*, *S2* and *S3*, whose reputation on *maintaining commitments to satisfy specified requirements* are 0.7, 0.9 and 0.6, respectively. The values of *NRep* for *S1*, *S2* and *S3* are 0.78, 1.00 and 0.67, respectively.

Normalized Execution Completion Time (NExecTime) - Depending on specified DARs, the system should also consider the foreseen necessary time interval to execute the task. This is done by using the Normalized Execution Completion Time (execution time estimation is discussed in Section 5.5).

Definition

Let LET be the lowest (estimated) required time to execute a task at one of candidate services. The Normalized Execution Completion Time of the i-th candidate ($NExecTime_i$) is computed as the relation between LET and the estimated time to complete the task at the i-th (T_i) candidate.

$$NExecTime_i = \frac{LET}{T_i}$$

Example 5.2 Consider three candidates *S1*, *S2* and *S3*. The foreseen task's execution time at the candidates is 48 seconds, 32 seconds and 147 seconds, respectively (foreseen task's execution time is estimated as presented in Section 5.5). The value of NExecTime for *S1*, *S2* and *S3* is 0.67, 1.00 and 0.22, respectively.

5.2 *On the Fly* Elections and Jobs with Several Tasks

Election-inspired scheduling can be used in scenarios with several jobs and corresponding tasks. It is also used when several tasks are generated as part of the same job. In such situations, there are several elections, one for each task.

Consider that a data service was elected to execute several tasks from the same job but a task's execution took (or is taking) a lot longer than promised by the service, then the remaining tasks assigned for such service and that are queued to be executed may be at risk of failing their DARs. In fact, the job itself may be at risk. The system deals with this problem for future elections by means of reputation: the problem was due to an erroneous promise by the data service, therefore its reputation will be decreased. Nevertheless, it is also interesting to take into consideration this delay within the current jobs execution to try to improve the situation. The community scheduler does this by verifying which tasks are taking (or took) much longer than promised by the data service and making *on-the-fly* (re-)elections.

Upon discovering the delay, the community scheduler is able to make new elections with the remaining tasks of the data service in order to see whether there are better candidates to execute those tasks (the data service with the problem can still be a candidate itself and can win again the election for tasks execution, if there is no other data service that is able to do it better).

On-the-fly elections are also used when there is a dependency relation between tasks. In such situations, the candidate promise on the required time to execute a task is used as a lower bound on the start time of the tasks that depend on such task (i.e. used as the *ExecutionStartTime* requirement of dependent tasks). The community scheduler considers runtime estimations when electing a task's executor. An error when estimating the required time to execute a certain task *i* may lead to an incorrect specification of the value of the *ExecutionStartTime* requirement for tasks that depend on the i-*th* task. When the difference between the execution finish time of the *i*-th task and the time used as the *ExecutionStartTime* of the tasks that depend on the *i*-th task is greater than a threshold value, then (i) the *ExecutionStartTime* requirement of tasks that depend on the *i*-th task is updated to the real value and (ii) *on the fly* elections are started to choose the executor of the tasks that depend on the i-*th* task.

The *on the fly* elections have the following rules:

I) If the execution of the *i*-th task finishes earlier than estimated – Then the executor of each of the dependent tasks must maintain (in the new elections) their previous promises or make other promises that are *better* (i.e. lower response time) than previous one (which makes the system improve the provided QoE level).

If there is a new foreseen finish time for any of the tasks that are dependent on the i-*th* task, then *on the fly* elections are started for the tasks that are dependent on the one whose foreseen finish time has changed, and so on, until no changes occur in the foreseen finish time of considered tasks or the executors of all tasks that are still waiting for execution are elected;

II) If the execution of the *i*-th task finishes later than estimated – Then all candidates can make new promises. Besides the elections of the new executors of the tasks that depend on the i-*th* task, the system also does *on the fly* elections for the executors of the other tasks that are waiting for execution. If there is no candidate that can satisfy specified requirements of any of the remaining tasks, then the system cancels the execution of the remaining tasks and informs the user that the job's DARs cannot be satisfied.

Example 5.3 Consider a certain job that comprises three tasks (task 2 is dependent on task 1 and task 3 is dependent on task 2). The executors of each task are elected: service I is the executor of task 1, service V is elected the executor of task 2 and service IV is the executor of task 3 (as represented in the first timeline of Figure 47).

Service I finishes the execution of task 1 much earlier than foreseen. As the foreseen execution finish time of task 1 was used as the *ExecutionStartTime* requirement of task 2, Service V would wait for a long time to start the execution of task 2 (represented in the second timeline of Figure 47). Then, the community scheduler starts an *on the fly* election for the executor of task 2, using the real execution finish time of task 1 as the value of the *ExecutionStartTime* requirement of task 2.

Service VI wins the new election for the executor of task 2, promising to finish the execution of task 2 earlier than it was originally promised by Service V (service VI can provide such promise on execution time finish due to the change on the *ExecutionStartTime* requirement of task 2). As the foreseen execution finish time of task 2 has changed, the community scheduler starts a new election for the executor of task 3 (considering a new value for the *ExecutionStartTime* requirement of task 3). Service II is elected the new executor of task 3 and the new foreseen required time to execute the entire job (represented in the third timeline of Figure 47) is much smaller than the original one.

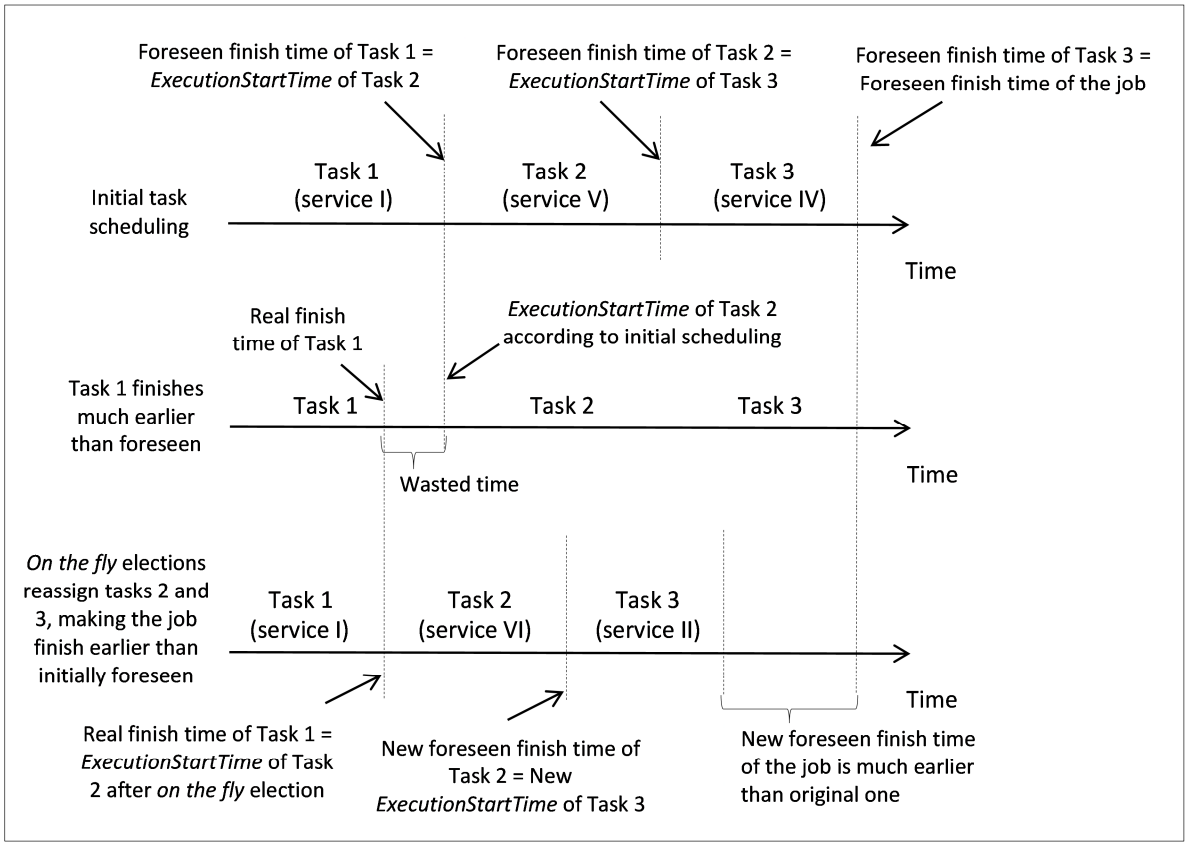

Figure 47 - On the Fly Elections Impact on Job's Finish Time - Example

5.3 Election-Inspired Scheduling and Alternative Sets of Tasks for the Same Job

Election inspired scheduling can also be used when more than one set of tasks is defined for a single job (set of tasks generation is discussed in Section 4.3). As discussed in that section, more than one set of tasks may be defined due to the possibility that data sets be placed in more than one manner, replicated and/or in different data services, or due to the use of the `OR` connector in DARs specification.

In such situation, each set of tasks is individually scheduled (i.e. the executor of each task is selected). When all elections are finished, the system selects the set of tasks that would be executed.

If there are one or more tasks in the set of tasks for which the system could not assign an executor, then such set of tasks is discarded.

If a single set of tasks is completely assigned for execution, then such set of tasks is executed. On the other hand, if more than one set of tasks is completely assigned for execution, then the system chooses the set of tasks with the highest value for the *Score of the Set of Tasks*.

The *Score of the Set of Tasks (SST)* has two components, one based reputation values and the other based on estimated completion times (the same way the candidate's *Election's Score* of Section 5.1.3 has). Such components are (i) the *Normalized*

Reputation of Executors (NRepExec) and (ii) the *Normalized Job Execution Completion Time* (NJobExecTime).

The *Score of the Set of Tasks* of the i-th set of tasks (SST_i) is computed by the following equation:

$$SST_i = \upsilon * NRepExec_{i,} + \omega * NJobExecTime_i$$

The calibration factors υ and ω are the same used while computing the Election's Score of candidates (Section 5.1.3).

Definitions

Let $AvgReputation_i$ be the average value for the reputation (on maintaining promises on tasks' execution) of the services selected to execute the tasks of i-th set of tasks. Let HAvgReputation be the highest value of AvgReputation among the ones of all considered set of tasks. The Normalized Reputation of Executors of the i-th set of tasks ($NRepExec_i$) is the relation between $AvgReputation_i$ and HAvgReputation:

$$NRepExec_i = \frac{AvgReputation_i}{HAvgReputation}$$

Let $JobExecTime_i$ be the foreseen execution completion time of the job while executing the i-th set of tasks (i.e. estimated finish time of the last task of the set to finish). Let LJobExecTime be the lowest value of JobExecTime for all considered set of tasks. The Normalized Job Execution Completion Time of the i-th set of tasks ($NJobExecTime_i$) is the relation between LJobExecTime and $JobExecTime_i$.:

$$NJobExecTime_i = \frac{LJobExecTime}{JobExecTime_i}$$

Example 5.4 Consider a certain job for which two set of tasks were generated: one with five tasks (that operate over a partitioned table) and another with a single task (which operate over a non-partitioned table). For each set of tasks, the community scheduler elects the executor of each task as if the other set of tasks does not exist.

Table 11 presents the reputation on *maintaining commitments to satisfy tasks requirements* for each of elected executors of tasks from the first set of tasks. The last column of Table 11 presents the average value of the reputation of elected executors (*AvgReputation*). The foreseen job's execution time while executing the first set of tasks is 135 seconds (JobExecTime = 135).

Table 11 – First Set of Tasks - Reputation of Elected Executors - Example

	Task					AvgReputation
	1	2	3	4	5	
Reputation of Elected Executor	0.6	0.7	0.5	0.7	0.5	0.5

The reputation value of the executor elected for the single task of the second set of tasks is 0.7 (AvgReputation = 0.7) and the job's execution time while executing the first set of tasks is 165 seconds (JobExecTime = 165).

Table 12 presents the values of NRepExec, NJobExecTime and SST for each set of tasks (we consider $\upsilon = \omega = 0.5$). The second set of tasks has a higher value SST value and, therefore, is chosen for execution.

Table 12 – NRepExec, NJobExecTime and SST for Distinct Set of Tasks - Example

	NRepExec	NJobExecTime	SST
First Set of Tasks	$\frac{0.5}{0.7} = 0.7$	$\frac{135}{135} = 1.0$	0.85
Second Set of Tasks	$\frac{0.7}{0.7} = 1.0$	$\frac{135}{165} = 0.8$	0.90

5.4 Reputation on Maintaining Commitments to Satisfy Tasks' Requirements

In election inspired scheduling, each user's command is transformed into tasks that may have several requirements. Candidate services must agree to satisfy all specified requirements of the task that is being scheduled. But elected candidates can fail (intentionally or not) to satisfy specified requirements.

The reputation *R* of a data service on maintaining its commitment to satisfy specified requirements while executing tasks is scaled to $[0,1]$. When the value of *R* for a certain data service is close to 1, then such data service almost always satisfies the specified requirements that it committed itself to fulfill. Hence, the community scheduler has a great confidence on the data service's capacity to maintain its commitments.

Definition

Let k represents a Success Factor $(k \in \{0,1\})$ *that indicates if a certain Data Service (S) fulfilled specified requirements of a given task (k = 1 → the service fulfilled specified requirements; k = 0 → the service did not satisfy specified requirements). The reputation R of S at time t considers the value of k for each of the j tasks executed by the service, as specified in the following Equation.*

$$R_{(S,t)} = \frac{1}{\sum_{i=1}^{j} d_i} \sum_{i=1}^{j} d_i k_i$$

In the previous Equation, *d* represents a time discount function. Such function is used to differentiate old values of *k* from more recent ones, as the service's behavior can change over time (e.g. due to changes in the environment). On the other hand, if the service does not execute any task, then its reputation does not change – the system maintains its last reminder about the service's behavior.

Besides that, in the previous equation we consider the use of information about the last *j* tasks executed by the service. In real implementations, the number of tasks (i.e. *j*) to consider must be adjusted according to the system's type. For instance, in some systems, data about tasks executed at the last hour may be sufficient, while for others systems, the system administrator may choose information about tasks executed during the last entire month.

We consider the time discount function defined by Huynh, Jennings & Shadbolt (2006). Therefore, for a time window (Δt) from the time when the task that had the i-*th* task was executed and the current time, the discount function *d* can be defined by the following Equation.

$$d_i = e^{\left(\frac{-\Delta t}{\omega}\right)}$$

In the above definition of *d*, a scaling factor *w* is used to allow the use of distinct time units and intervals. For instance, if the time unit used is *minute* and a Success Factor obtained ten minutes ago should have only 15% of the effect of a recently obtained Success Factor, then *w = -10 / (ln 0.15)*.

The system administrator can configure the system to alert him when the reputation of a any service is undesirable (i.e. bellow a limit value defined by the administrator).

Example 5.5 Consider two data services *S1*and *S2* that should execute 10 tasks. Each task has just one requirement and there is an interval of 1 minute between tasks execution. Table 13 presents the service that executed each task and if the service satisfied or not the task's requirement. Service *S1* executes tasks 1, 2, 9 and 10, and satisfies the requirement of tasks 1, 9 and 10. Service *S2* executes all the tasks from 3 to 8, and fulfills the requirement of three tasks.

Table 13 - Tasks Executed by Services S1 and S2 - Example

		Timestamp / Task									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Executed by	**S1**	X	X							X	X
	S2			X	X	X	X	X	X		
Requirement Satisfied?		Yes	No	Yes	Yes	No	Yes	No	No	Yes	Yes

In order to evaluate each service's reputation maintaining its commitment to fulfill task level requirements, we consider that a minute interval represents a fall of 20% on the importance of the obtained Success Factor. Figure 48 presents the reputation of services *S1* and *S2* at the end of each timestamp from 1 to 10 (i.e. after the execution of the task that was submitted at the timestamp).

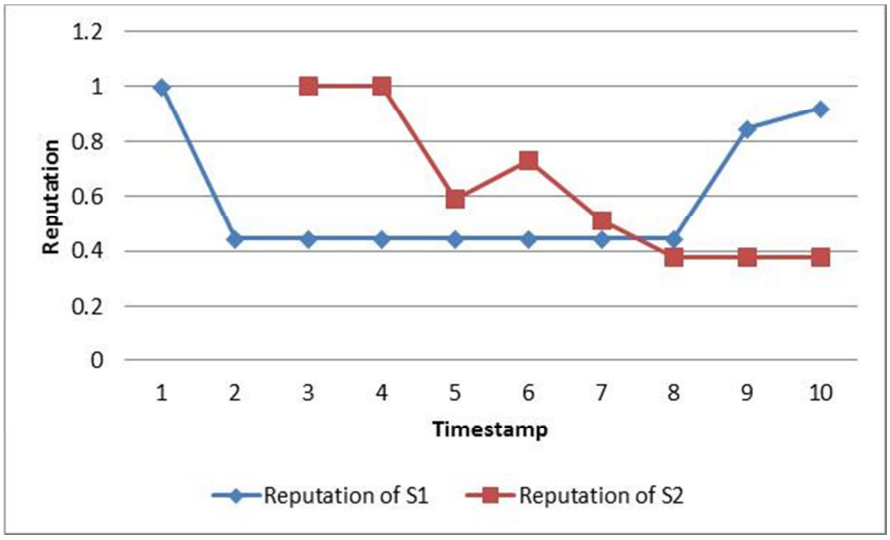

Figure 48 - Reputation of services S1 and S2 – Example

The reputation values of Figure 48 were evaluated considering just the execution of tasks 1 to 10. Therefore, in Figure 48, service *S2* has no reputation at timestamps 1 and 2. Both *S1* and *S2* satisfied the requirement of the first task they executed. Therefore, both services have a reputation of 1 just after executing the first task. Services' reputation decrease just after the first requirement they do not fulfill. A service's reputation increases when the service satisfies a requirement. Service *S1* does not execute any task from timestamp 3 to 8, hence its reputation is the same on such period. The reputation of service *S2* remains the same in the period from 9 to 10, when the service does not execute any task.

5.5 Using Promises and Reputation to Estimate Tasks' Execution Time Interval

During tasks' executor elections, candidates can make promises on the necessary time interval to execute tasks. But candidates can break (intentionally or not) their promises, taking a time interval to execute a task distinct from the one that they promised to take. Section 5.5.1 presents how to estimate the necessary time to execute a task at a certain candidate, considering candidate's promises and reputation. Then, Section 5.5.2 presents how we evaluate candidates' reputation on maintaining their promises on tasks' execution time interval.

5.5.1 Estimating Task's Execution Time Interval

Candidates can make promises on the necessary time to execute tasks. The reputation of a data service on *maintaining its promises on tasks' execution finish time interval* is used to estimate the execution time of a certain task by *calibrating* candidate's promise.

The estimated task's *execution time* (*et*) value is computed as defined by the following Equation, where $et_{i,j}$ represents the execution time of task i at candidate j, $P_{i,j}$ represents the candidate's j promise on the execution time interval of task i, and $R_{j,t}$ represents the reputation on *maintaining promises on tasks' execution time interval* of candidate j at time t.

$$et_{i,j} = P_{i,j}(1 + R_{j,t})$$

Example 5.6 Consider the election with two candidates: services *S1* and *S2*. Candidates' promises are 98 seconds and 107 seconds for services *S1* and *S2*, respectively. The first candidate's reputation on *maintaining promises on tasks' execution time interval* is 0.3, while the reputation value for the second candidate is 0.1, as specified in Table 14. In such situation, service S2 is the one that provides the lowest execution time interval, as the foreseen task's execution time in *S2* (117.7 seconds) is lower than the one at *S1* (127.4 seconds).

Table 14 – Estimating Task's Execution Time Using Reputation - Example

	Candidate's Promise (seconds)	Candidate's Reputation	Foreseen Task's Execution Time (seconds)
S1	98	0.3	127.4
S2	107	0.1	117.7

5.5.2 Reputation on Maintaining Promises on Tasks' Execution Time Interval

We propose a reputation measure on *how much* a service fails to accomplish its execution time promises.

The reputation R of a data service on maintaining its promises on tasks' execution finish time interval is scaled to $]-1, \infty)$. Negative values for reputation represent the service would finish task's execution before the time it has promised (a value of -1 means that task's execution would finish instantaneously). On the other hand, positive values indicate that the service would finish task's execution after the time it has promised to finish (a value of ∞ means that task's execution never ends). When the reputation value is zero, the scheduler believes that the service would finish task's execution exactly at the time it has promised to finish.

Definition

For each data service S, at time t, the value R of service's reputation on maintaining promises on tasks' execution time interval is evaluated considering

service's promises (P) on the past j elections the service won and the real task's execution time interval (E), as represented in the following Equation.

$$R_{(S,t)} = \frac{1}{\sum_{i=1}^{j} d_i} \sum_{i=1}^{j} d_i \left(\frac{E_i - P_i}{P_i}\right)$$

In the above Equation, *d* represents the same time discount function we used when defining the reputation of a data service on maintaining its commitment fulfill specified requirements. Again, the time discount function is used to make older data about service's behavior less important than newer one. The above Equation considers the last *j* elections in which the service made a promise and won the election. In real implementations, the number of elections to consider may depend on the system's characteristics. Besides that, the initial reputation value of each service may be fixed by the system administrator.

Example 5.7 Consider a certain service *S1* that won ten elections in which it made a promise. Table 15 presents the promise values and the actual tasks' execution finishing time. There is an interval of 1 minute between tasks execution. Service *S1* did not win the elections of tasks 7, 8 and 10.

Table 15 - Service's Promises and Tasks' Execution Finish Time Interval

	Timestamp / Task												
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
S1 Promise	5,0	3,0	7,0	8,0	8,0	8,5			5,0		3,0	6,0	5,0
Task's Finish Time Interval	5,0	5,0	8,0	8,1	8,3	9,0			4,0		2,0	4,0	4,5

Figure 49 presents the reputation of *S1* on maintaining its promises on tasks' execution finish time interval. We consider that each minute between the task's execution and the current time represents a fall of 20% on the importance of the task's promise when evaluating services reputation.

Initially, *S1* maintained its promise and terminated the execution of task 1 by the same time interval it promised to terminate. However, in tasks 2 to 6, *S1* took more time to terminate the task than it had promise to take. Hence, from time 2 to 6, service's reputation is positive. Then, *S1* took less time to execute tasks 9, 11, 12 and 13 than it promised to take. Therefore, at timestamp 9, service's reputation value begins to drop down, and becomes negative at timestamp 11. The lowest reputation value is obtained at timestamp 12. As the execution time interval of task 13 is closer to the service's promise than the execution time interval of task 12 was to the corresponding promise, service's reputation at timestamp 13 is closer to *zero* than it is at timestamp 12.

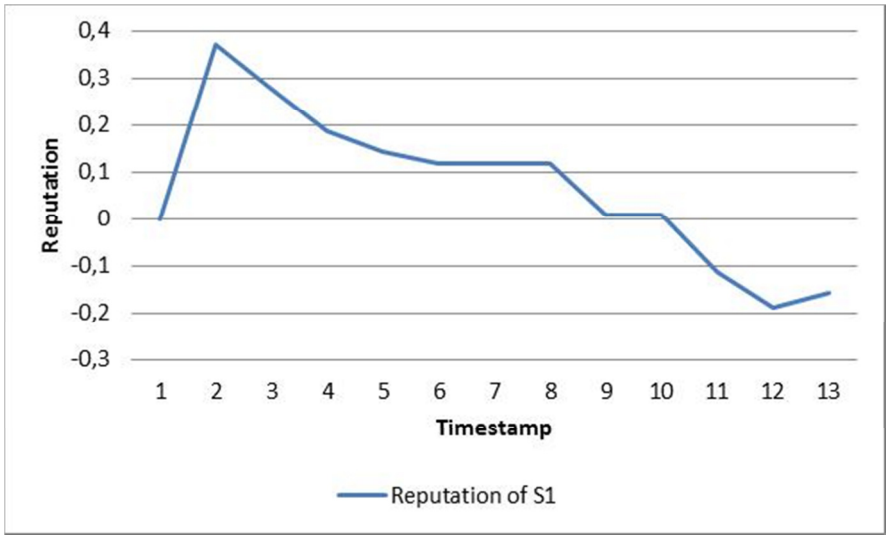

Figure 49 - Reputation of *S1* on Maintaining Promises – Example

5.6 What-if Elections and Dynamic Replication for QoE

When a certain job cannot be executed, the system may start *what-if elections*: data services that do not store the data required to execute (one or more of) job's tasks, are invoked to evaluate if they would execute considered tasks in case they store required data.

A what-if election considers all the job's tasks just like a normal election. However, it just verifies if there is a schedule in which all tasks are assigned, and does not effectively requests tasks execution.

When a certain service that does not store the required data to execute a task wins the task's what-if election, then it is considered that the creation of a data replica at such service would bring a *benefit* to the system (i.e. increase the provided level of QoE, as it would enable the execution of job that was not accepted for execution). If the total benefit (β) of the creation of a certain data replica at a certain service reaches a threshold value, then such data is selected for replication, and the system administrator is alerted.

Computing the Benefit of Replica Creation

In order to compute the total benefit (β) of a replica creation, the QoE oriented database system should differentiate the benefit for old jobs from the ones for newer jobs. Therefore, a time discount function may be used (the same way it was used when computing services' reputation):

$$\beta = \sum_{i=1}^{j} d_i$$

In the previous Equation, *d* represents the time discount function. The total benefit (β) of a certain replica creation is the sum of a time decay function's result for every time

that the replica creation brought any benefit to the system. We use the exponential decay function:

$$d(t) = d_0 * e^{-\lambda * t}$$

In such function, d_0 represents the initial function value, λ represents the decay constant and *t* represents a time window.

Filtering What-if Candidates based on Replica Synchronization Requirements

Data replicas may have distinct freshness requirements: a replica may be (i) just a snapshot of the master data at a certain moment; (ii) periodically updated with changes that occur in the master table (i.e. data updates occur asynchrony in the master table and in the replica); and (iii) always synchronized with the master table (i.e. data updates occur in the master table and in the replica as part of the same transaction – typically controlled by a two-phase commit protocol).

In this thesis, we do not study replica synchronization mechanisms, which are already studied in other works and are present in current database management systems like Oracle 11g R1 Enterprise Edition [Oracle, 2010] and SQL Server 2008 [SQL Server, 2010]. If the system administrator identify that the underlying resources (e.g. network resources) cannot guarantee the required replica synchronization level between some services, it can configure the system to do use such services in what-if elections.

5.7 Reputation and Resource Availability Monitoring

Some of proposed requirements impose restrictions on data availability. In Section 5.1 we presented that resource availability predictions can be used to filter the pre-candidate services for executing availability related tasks: services whose predicted availability does not meet specified requirements are not even considered for the election of an availability related task.

Nevertheless, availability prediction methods can fail and, then, data services availability must be periodically checked. There are several mechanisms that can be used to monitor resources availability, including the use of a *telnet* client to connect to remote data services or the use of a heartbeat daemon that periodically verifies node's availability. The mechanism to be used and the resource verification interval depend on the system's architecture (i.e. if a distributed or parallel system) and configuration.

If the system detects that a resource that should be available in a certain period (i.e. is executing a task that has availability-related requirements) is in fact unavailable, then it updates such service's reputation on *maintaining commitments to satisfy tasks' requirements*. Besides that, if unavailability persists for a certain period, then the system takes corrective actions, running an *on the fly election* (Section 5.2) for the job(s) that are being affected by resource unavailability, and also alerts the system administrator.

5.8 Conclusion

In this chapter, we presented the Election-inspired reputation aware QoE-oriented scheduling strategy. In such strategy, some data services are considered as pre-candidates to execute a task. Then, the pre-candidates should evaluate if they can or cannot satisfy task's requirements. If a pre-candidate evaluates that it can satisfy task's requirements, then it can present itself as a candidate. Candidates can also make promises on task's execution time.

This chapter also discussed how to evaluate a candidate's reputation on *maintaining commitments to satisfy specified requirements*. Such reputation is used to classify candidates in terms of their capacity to fulfill task's requirements. We also presented how the community scheduler foresees tasks' execution time, considering both candidates' promises and their reputation on *maintaining promises on tasks' execution time interval*. We presented how to select election's winner considering both candidate's reputation on *maintaining commitments to satisfy specified requirements* and the foreseen task's execution time (i.e. to select a dependable candidate that provides a low task's execution time). The use of the reputation system increases the system's capability to maintain its commitment to the user (i.e. to satisfy the DARs that the system agrees to satisfy). Therefore, it increases the QoE level the system provides.

We also presented the use of *on the fly* elections during the execution of jobs with several tasks in order to improve the provided QoE level. We detailed the use of the election inspired scheduling when there are several alternative sets of tasks for the same job.

Besides that, we also discussed resource availability monitoring and how *what-if* elections can be used to detect what data replica would improve the level of QoE provided by the database system.

In the following chapter, we discuss how tasks are evaluated and managed at data services.

6 Tasks Evaluation and Management at Data Services

Data services are responsible for tasks execution. In order to be assigned as the executor of a certain task, the data service should commit itself to satisfy the task's requirements. The service should also specify a foreseen execution time for the task.

In Section 6.1 we discuss the participation of the data service in elections. We present how data services can decide on presenting themselves as pre-candidates and how tasks scheduler evaluates if it can or cannot satisfy task level requirements (decide on being a candidate or not). Besides that, we discuss how the scheduler specifies its *promise* on tasks' execution time interval.

In order to decide on being a candidate, a service may need to estimate the execution time of a query or to estimate the necessary time to copy remote data. In Section 6.2 we discuss how we estimate the necessary time to transfer some piece of data between hosts. Then, in Section 6.3 we present a strategy to estimate the execution time of database queries based on query's cost and system load.

Finally, Section 6.4 presents the chapter's summary and some final comments.

The main contributions of this chapter are: (i) multi-target intra-site query scheduling policy, (ii) query execution time estimation method, and (iii) method for automatic adjustment of the multi-programming degree for fulfilling task level requirements.

6.1 Participating in Elections

Data services can become pre-candidates in elections voluntarily or when selected by the community scheduler. In both cases, pre-candidates should evaluate if they can or cannot satisfy task's requirements. Besides that, candidate services should make *promises* on the required time to execute the task.

In Section 6.1.1 we discuss when a data service should present voluntarily itself as a pre-candidate to participate in elections. Then, in Section 6.1.2 we present how the tasks scheduler evaluates if the service or cannot satisfy specified task level requirements, and how the tasks scheduler decides on the value to promise as the required time to execute the task.

6.1.1 Presenting Itself as a Pre-Candidate

A data service may present itself as a pre-candidate to participate in elections. This is an indication that the service *wants* to evaluate if it can or cannot satisfy a task's requirements and make some promises on the required time to execute the task even

though the community scheduler evaluates that the service does not have high changes to win the election. Such intention is valid for the elections that occur during a certain time window.

Participating in an election requires some processing time of pre-candidates (e.g. to evaluate task's requirements satisfaction). Although service's administrators can implement any local policy to choose when the service should present itself voluntarily as a pre-candidate to execute a task, we propose that the service should only present itself voluntarily to participate in elections when service's load remains below a certain threshold value for a certain time (the values for the load metric and for the time window should be tuned by the system administrator).

Example 6.1 In Linux-based systems, the data service may present itself to participate as a pre-candidate in all elections that occur in the next minute when the *load average* metric remains below the value of 0.3 for 5 minutes.

6.1.2 Evaluating Tasks' Requirements and Making Promises

During election-inspired scheduling, the community scheduler informs to selected pre-candidate services about the task whose executor is being elected and about such task's requirements. Then, pre-candidate services should verify if they can execute such task and satisfy its requirements while still fulfilling the requirements from tasks the service already committed itself to fulfill.

Table 16 summarizes the tests that should be done for each task level requirement. If the tasks scheduler foresees it can fulfill all the requirements of the considered task, then it can present its data service as a candidate to execute the task. Table 17 presents the description of the methods/properties used in Table 16 for testing requirements.

Table 16 - Evaluating Requirements Fulfillment

Task Level Requirements	Testing Requirements
Task.Relation(ρ).Freshness	Data to which the local replica of ρ corresponds in the master table ≥ Task.Relation(ρ).Freshness
Task.ExecutionStartTime	Task.ExecutionStartTime ≤ Task.ForeseenInitialTime
Task.ExecutionDeadline	Task.ExecutionDeadline ≥ Task.ForeseenFinishTime
Task.ExecutionFinishTime	Task.ExecutionFinishTime ≥ Task.ForeseenFinishTime
Task.TemporaryRelation(*relation_identifier*).AvailableFrom Task.TemporaryRelation(*relation_identifier*).AvailableUntil	*Period* = [Task.TemporaryRelation(*relation_identifier*).AvailableFrom, Task.TemporaryRelation(*relation_identifier*).AvailableUntil]
Task.Results.StoreAtTemporaryRelation	Memory.HasSpace(Task.TemporaryRelation(*relation_identifier*).Size; *Period*)

Task Level Requirements	Testing Requirements
Task.TemporaryRelation(*relation_identifier*).AvailableTime	Not Exists (Scheduled.Downtime(*Period*))
Task.TemporaryRelation(*relation_identifier*).AvailabilityPercentage	(Scheduled.Downtime(*Period*).Size / *Period*.Size) <= Task.TemporaryRelation(*relation_identifier*).AvailabilityPercentage
Task.Results.StoreToFutureUse	Memory.HasSpace(Task.Results.Size)
Task.RefreshMode	Queue.Place(Task) && Dataset.CanRefresh(Task.RefreshMode; *Period*)
Task.ExecutionDate	Task.ForeseenExecutionDate ≥ Task.ExecutionDate
Task.ExecutionPriority	Queue.Place(Task)
Task.ExecutionMode	Queue.Place(Task)

Table 17 – Testing Methods/Properties Description

Method	Description
Memory.HasSpace(*Relation Size*; *Time Window*)	Verifies if the system has enough free space in memory (primary or secondary) to store a relation. It uses information about the required space and the time period on which the space is required.
Queue.Place(*task*)	Verifies if the system can place the task in tasks queue while satisfying the requirements of the newly placed task and of the other tasks that were already in the queue. Returns `TRUE` if task's requirements can be fulfilled (i.e. the service can be a candidate) or `FALSE` in case requirements cannot be fulfilled
Task.ForeseenInitialTime	Returns the foreseen initial execution time of a task. It is set by the *Queue.Place* method and considers a certain placement of the task in tasks queue.
Task.ForeseenFinishTime	Returns the foreseen finish execution date and time of a task. It is set by the *Queue.Place* method and considers a certain placement of the task in tasks queue.
Task.ForeseenExecutionDate	Returns the foreseen execution date(s) of a certain task. It is set by the *Queue.Place* method and considers a certain placement of the task in tasks queue.
Scheduled.Downtime(*Time Window*)	Verifies if the system has a scheduled downtime period in a certain time window
Scheduled.Downtime(*Time Window*).Size	Returns the size of the scheduled downtime period in a certain time window

Method	Description
Dataset.CanRefresh(*RefreshMode, Time Window*)	Verifies if the system can guarantee the required refresh mode for a certain dataset in a certain time window. It uses information provided by system's administrator about constraints on data replication modes (discussed in Section 5.6)

The method *Queue.Place* (Table 17) is a key method for task's requirements evaluation. Each tasks scheduler manages two queues: one for very small tasks and another for the other tasks. Such size-based organization aims to provide an 'express queue' to very small tasks. The *Queue.Place* aims at placing a task in the corresponding tasks queue in order to finish tasks execution as soon as possible, but while satisfying the requirements of such task and of other tasks that were already placed in the queue.

The *Task.ForeseenFinishTime* property stores the task's foreseen execution finish (date and) time considering a certain placement of the task in the tasks queue. In normal elections, task's initial time is determined by the task's position in the queue and the task's execution finish time is estimated as the sum of two components: (i) the time required to transfer the necessary data to execute the task (input data transfer time) and (ii) the time required to execute task's database command locally (in Section 6.2 we present how we estimate the data transfer component and in Section 6.3 we discuss database command's execution time estimation). But if the task is being evaluated for a what-if election, then service's promise considers only the second component: the time required to execute task's database command.

The *Task.ForeseenFinishTime* property is used to compute the *candidate's promise* on the necessary time interval to execute the task:

$$Promise = Task.ForeseenFinishTime - Election's\ Timestamp$$

The *Task.ForeseenInitialTime* is used to estimate the timestamp on which a task's execution begins. It is determined by the task's position in the queue.

The *Scheduled.Downtime* method verifies if the system has a scheduled downtime in a certain time period. If the service is suitable to frequent unscheduled downtime periods and it accepts for execution many tasks that have availability requirements, then the service would probably have a low reputation on maintaining commitments to satisfy tasks' requirements.

In the following sections, we discuss method to estimate the required time to do data transfer between sites and to execute a query at the local site database.

6.2 Data Transfer Time Estimation

In order to execute a task, a service may need to copy remote data to the local host (as such data is required to task's execution). During task's requirement evaluation, tasks scheduler must estimate the required time to execute this operation (i.e. data copy between hosts) in order to estimate the required time to execute a task.

Data transfer time depends on several aspects, like network latency and bandwidth, I/O throughput, network card speed and data set size. The size of the data set that should be copied between hosts (from a remote host to the service or from the service to a remote host) is obtained using database statistics. All other components are estimated according to the network benchmarks strategy proposed by Antunes & Furtado (2007): tasks scheduler transfers (periodically or on demand, according to the made by the system administrator) data sets of distinct sizes between sites, measuring the required time to transfer such data. Then, such data transfer times benchmark is used to estimate the time required to copy task's input data from remote sites to the local host.

6.3 Query Execution Time Estimation

In order to make a promise on the necessary time to execute a task and to evaluate requirements fulfillment, the tasks scheduler must have a strategy to estimate task's execution time. In this Section, we present query execution time estimation strategies.

Typical query execution times may vary from just a few milliseconds to several minutes. In order to deal with such variation, tasks scheduler uses two queues: an *express* queue for very small tasks (e.g. whose execution takes less than 1 second) and another one for the other queries.

Only a few small tasks can be executed concurrently (low multi-programming level – MPL - limit). However, the number of normal and long-running tasks executed concurrently is adjusted by a specific time estimation method. Such queue management strategy is represented in Figure 50. The use of small tasks queue increases system's throughput (a large number of small tasks is executed in a small time window), but the MPL limit for small tasks should not be so large that the execution of small tasks significantly impact in the execution time of normal and long-running queries.

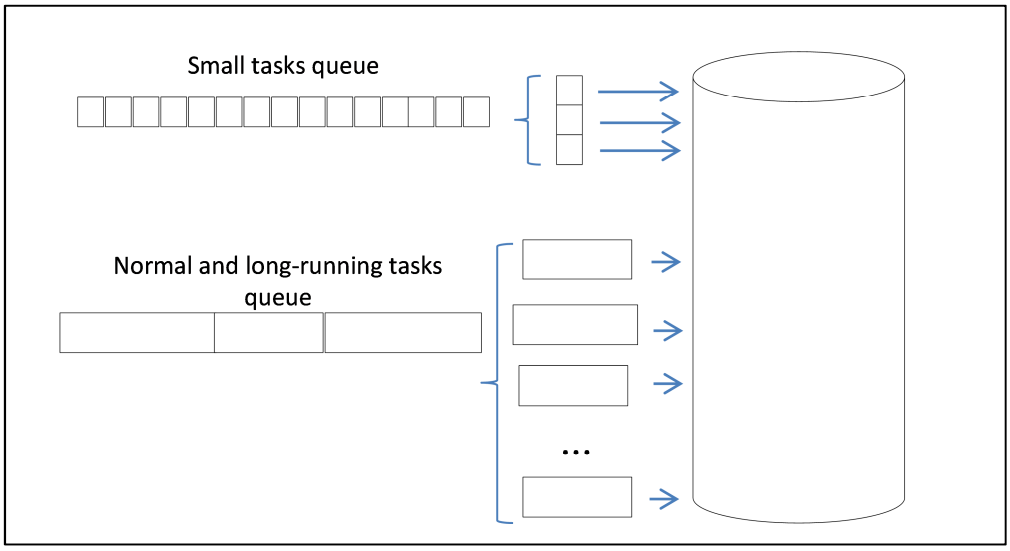

Figure 50 - Multiple queue management

First, Section 6.3.1 presents how to estimate execution time of small tasks. Then Section 6.3.2 presents how to estimate the execution time of normal and long-running database queries.

6.3.1 Small Tasks Execution Time Estimations

A fixed (typical) value is defined for the duration of small tasks execution by the DBMS. As small tasks execute too quickly, the typical error for this strategy is relatively small (for instance, if small tasks queue manage the executions of queries that typically take less than 1 second to execute and the typical value chosen for small tasks duration is of 500 milliseconds, then the maximum forecast error would be lower than half a second).

The total execution time of a small task should consider the required time to execute the query by the DBMS but also consider the time that the task waits in the queue.

Consider that *typ* is the value the typical execution time of a small task. Consider a task q that is placed in the small tasks queue after n tasks. The following equation presents how to compute the total execution time *tot* of q in such situation:

$$tot = typ * (n + 1)$$

Small tasks are auto-detected by the system. It periodically verifies (by executing some sample queries) which is the query execution cost that leads to an execution time that is near the typical duration of small tasks. Then, all tasks with a foreseen execution cost smaller than the obtained value are considered as small tasks.

6.3.2 Estimating Normal and Long-running Queries Execution Time

Execution time estimation is not a common feature of current database management systems. On the other hand, most current query optimizers inform the user about some kind of *execution cost* for the query. Such execution cost is used together with system's workload in execution time estimation.

Consider that, at a certain point in time, the system is executing a certain number of concurrent commands (multiprogramming level - MPL). To such point in time there is a corresponding workload cost (and workload execution phase). Whenever a new command execution starts or ends, there is a workload execution phase change, as represented in Figure 51.

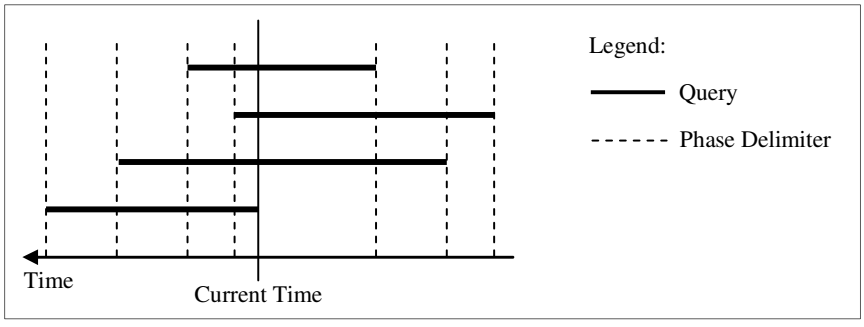

Figure 51 - Workload Execution Phases - Example

Consider that there is a conversion function (δ) that returns the estimated execution time for a certain estimated execution cost (it is important to notice that there is no exact relation between execution cost and execution time, and that execution time depends on several factors, including external loads, machine's processing power and main memory size).

The conversion function can be used to estimate the time required for a phase change (considering the total phase cost as input for the function).

Estimating Query Execution Finish Time When Changing the MPL

For simplicity, consider that the execution cost of a command is equally distributed during all the period of command's execution and that in a certain time window the system processes the same amount of cost units from all the queries that it is executing.

Suppose a certain system that is executing a single query q whose execution cost is c_q. Using the conversion function (δ), one can estimate such query's execution time. The execution of q starts at time t_0. After a certain time point during the execution of q (t_1), the system starts the execution of a new query (q_2), increasing the system's MPL to 2 (*NewMPL*). Also consider that δ^{-1} can be used to estimate the execution cost processes in a certain time window.

Figure 52 presents some pseudo-code to represent the algorithm used to estimate the time required to finish the execution of q after t_1 (*RemainingTime*).

```
ProcessedCost = δ^-1(t1 - t0)
RemainingCost = cq - ProcessedCost
RemainingTime = δ(RemainingCost x NewMPL)
```

Figure 52 - Estimating a query finish time after changing the MPL

Now, consider that, at a certain time point t_2 after t_1 and before the end of the execution of q and of q_2, the system starts the execution of a new query (q_3), as represented in Figure 53.

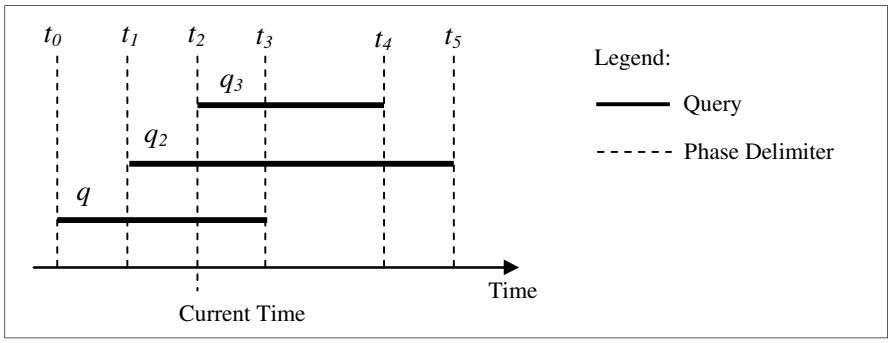

Figure 53 – Phase Changes When Increasing the MPL - Example

At t_2, the system can estimate the remaining execution time of each query (i.e. the time when each phase change would occur). In order to do that the first step is to estimate how much of q and q_2 were already processed. This is a generalization of the first line of the pseudo-code from Figure 52 and is represented in Figure 54.

```
For Each Phase_Already_Finished
  TotalProcessedCost = δ^-1(t_End_Phase - t_Begin_Phase)
  For Each Query_Executed_In_Phase
    ProcessedCost_Query += (TotalProcessedCost / MPL_In_Phase)
  Next
Next
```

Figure 54 - Estimating Processed Cost

Then, the system should estimate the remaining execution cost of each query (as it does for q in the second line of the pseudo-code from Figure 52). Such code is represented in Figure 55.

```
For Each Query
  RemainingCost_Query = TotalCost_Query - ProcessedCost_Query
Next
```

Figure 55 - Estimating Remaining Execution Cost

At that point starts the estimation of the next phase changes (i.e. the execution finish time of ongoing queries). Such process is represented in Figure 56 and begins with the selection of the next query to finish. Then, the estimating time and executed cost of the next phase change is computed. The remaining cost of each query is updated, and one query is removed from the pending list. These actions are repeated until all phase changes are identified (i.e. the remaining execution time of each query is estimated).

```
While MPL > 0
      /* Get Next Query to Finish */
      LowestRemainingCost = HighValues
      For Each Query in ExecutingQueryList
            If RemainingCost_Query < LowestRemainingCost then
                  LowestRemainingCost = RemainingCost_Query
                  NextQueryToFinish = Query
            End if
      Next

      /* Get the time of next Phase Change */
      CostSubsequentPhase = LowestRemainingCost * MPL
      TimeSubsequentPhase = δ(CostSubsequentPhase * MPL)

      /* Update costs and times of pending queries */
      For Each Query in ExecutingQueryList
            RemainingCost_Query -= LowestRemainingCost
            RemainingExecutionTime_Query += TimeSubsequentPhase
      Next

      /* Execution of NextQueryToFinish would finish in
            RemainingExecutionTime_Query */
      Remove NextQueryToFinish from ExecutingQueryList

      MPL -= 1
Wend
```

Figure 56 - Estimating Future Phase Changes

Choosing the Conversion Function

Heiss & Wagner (1991) models the relation between concurrency level and the performance of a transaction processing system as a polynomial function of degree 2 with a parabola that opens downward. In such case, after saturation an increase in the system's load causes a drop in throughput.

We also model the conversion function δ as a polynomial function of degree 2, but our parabola opens upward. Initially, an increase in system's load would cause a small increase in the execution time. Above the saturation point, a small increase in system's load would cause a great variation in each query's foreseen execution time.

When a phase change occurs due to a query execution finish, then the real time necessary for the phase change is used to calibrate the function's coefficients. This is done by a quadratic regression considering the foreseen execution cost for the last Workload Execution Phase and the real execution finish time of the Workload Execution Phase, besides some other measures obtained in previous phase changes that occurred due to a query execution finish.

6.4 Conclusion

In this Chapter we discussed how data services decide on presenting itself voluntarily as a pre-candidate to an election, and how they decide to be a candidate or not in an election. We detailed task scheduling and task level requirements evaluation in data services and presented a strategy to estimate the execution time of database queries considering both query cost and system load.

We discussed the tests that a tasks scheduler should do for each type of task level requirement in order to verify if it can fulfill task's requirements or not. We presented how the service would specify its promise on task's execution time interval.

Besides that, we presented a strategy to estimate the execution time of a query considering its execution cost and the system's load. Such strategy is conceived to be used by database *external schedulers*. This way, the tasks scheduler can be implemented as an external scheduler, which enables its use together with currently available database management systems.

In the following chapter, we discuss the use of specialized metrics to evaluate the level of QoE that a database system provides to users.

7 Measuring the QoE Provided by Database Systems

This chapter is concerned with the appropriate metrics to evaluate Quality of Experience. *Traditional* performance indicators (e.g. throughput or execution time) do not measure the level of QoE the system provides for users: they usually measure how *fast* a system is but not how many requirements a system fulfills or how *satisfactory* a system is. Therefore, *traditional* performance indicators are not the most adequate to measure the *performance* of a QoE-oriented database.

As an illustrating example, consider two queries submitted simultaneously and that query Q_1 must be completed within 1 second and that this is a tight schedule, and query Q_2 must be completed within 1 hour and that is a relaxed schedule. In a *best effort* system they will run simultaneously. Assume that Q_1 does not complete within 1 second in that scenario. In a QoE-oriented system, Q_2 may be scheduled to be executed after Q_1 if that allows Q_1 to meet its 1 second goal. From a throughput perspective postponing execution of Q_2 for 1 second may not improve the metric (i.e. the throughput), however, from a QoE perspective, it means that Q_1 can execute and is successful as well as Q_2.

We propose four specialized indicators that can be used to measure distinct aspects of a QoE-oriented database. The first three of them (*Acceptance Rate*, *Commitment Maintenance Rate* and *Success Rate*) provide measures of specific operations (e.g. system's capability to maintain its commitment on satisfying certain DARs), while the last one (*QoE Level* indicator) can be used to estimate the level of QoE that the system provides. Such KPIs can be used in any QoE-oriented database that follows the model we proposed in Chapter 3 (no matter if it is a centralized, parallel or distributed database), and it can be used to compare the level of QoE provided by a QoE system versus the level provided by a *best effort* system.

Even though users cannot explicitly specify requirements in *best effort* oriented systems, the data access requirements intention may be assumed for comparison purposes with QoE systems. Hence, proposed KPIs may also be used to compare the QoE level provided by *best effort* systems and by QoE oriented systems.

Besides that, the system uses the KPIs' values to alert the administrator when the QoE level provided to a certain database user (or transaction) remains too low for long periods. This can enable administrators to take corrective actions before users complain about system's performance.

In Section 7.1, we define the *Acceptance Rate Indicator*. In the proposed model for QoE oriented databases, the system can reject to execute a command informing the user that it cannot satisfy specified requirements. The *Acceptance Rate Indicator* KPI provides a measure on how many of the submitted jobs (i.e. statements or blocks of statements) the system agreed to execute. While it is important that the system informs the user of jobs that it cannot satisfy, if the system rejects the execution of a too high number of jobs, the user would probably become unhappy.

Then, in Section 7.2 we propose the *Commitment Maintenance Rate Indicator*. When a QoE oriented system agrees to execute a certain user's command, it should satisfy all the requirements associated with such command. But the system may fail to satisfy some of the specified DARs (e.g. due to a wrong prediction of future conditions, a command execution deadline cannot be satisfied), which would lead to user's disappointment. The *Commitment Maintenance Rate Indicator* provides a measure on the number of jobs that had their DARs satisfied, considering just the jobs with associated DARs that the system's agreed to execute.

In Section 7.3 we define the *Success Rate Indicator*. Consider a user who submits several commands that have explicitly defined DARs. The system foresees it cannot satisfy the requirements of some of the submitted commands (and such commands are not executed). The system also fails to satisfy the requirements of some of the commands that it agreed to execute. Finally, the user may feel that just a small number of submitted commands were effectively executed and had their requirements satisfied. The *Success Rate Indicator* provides a measure on the number of jobs that had their DARs satisfied, considering all submitted jobs that have associated DARs.

In Section 7.4 we present the *QoE-Level Indicator*. The QoE level a system provides depends mostly on whether users' expectations are met or not. But the system may not be able to satisfy all users' expectations, since some of them may be infeasible, considering existing resources. The way that users become aware that their expectations are not going to be met can also influence users' satisfaction degree. The *QoE-Level Indicator* aims to measure the level of QoE that the system provides considering several factors, including the number of jobs with explicitly defined DARs that the system accepts to execute and the number of jobs whose DARs the system satisfies.

In Section 7.5, we present some examples and discuss obtained values for each of the proposed indicators.

Then, Section 7.6 presents the use of proposed indicators to alert administrators when the system provides a level of QoE that is not acceptable.

Section 7.7 closes the chapter presenting its summary and some comments.

Hence, the main contribution of this chapter is the proposal of some specialized performance indicators that can be used to measure distinct characteristics of QoE-oriented database systems, and to compare the QoE levels provided by distinct systems (which may be *QoE-oriented* or *best effort* oriented).

7.1 Acceptance Rate Indicator (AR)

Users may submit several commands to the system, each one with several requirements. For each command, the system analyzes if it can or cannot satisfy the specified DARs. The system only agrees to execute commands whose DARs it can satisfy. The *Acceptance Rate Indicator* presents a measure on the rate of jobs with explicitly defined DARs that the system agrees to execute.

Let NJ represent the number of jobs that have at least one DAR and were submitted to the system in a certain time period (Δt). Let AJ represent the number of jobs with at least one DAR that were submitted to the system in Δt and

the system agreed to execute. The Acceptance Rate (AR) Indicator represents the relation between AJ and NJ, as represented in the following Equation:

$$AR = \frac{AJ}{NJ}$$

where $0 \leq AR \leq 1$.

A high Acceptance Rate value indicates the system agreed to execute almost all the jobs (i.e. commands or blocks of commands) that had associated requirements. On the other hand, if the system rejects almost all the jobs with associated DARs, then the AR value gets close to zero. Among the causes of low AR values, we can cite high system loads and high number of unfeasible requirements specified by users. Both situations can indicate that the system does not have the necessary resources to satisfy user requirements.

When the system accepts almost all users' jobs (high Acceptance Rate), users may feel that the system is *very capable,* and that it would solve all users' problems. But this may be wrong… The system may fail to satisfy the DARs of accepted jobs.

If the system rejects some user jobs and satisfies the DARs of almost all the jobs it accepts, users' would probably be happier than in the previous case.

On the other hand, if the system has high *risk aversion* (or low resources) and rejects almost all users' jobs (low Acceptance Rate) users would feel frustrated (some of them would feel so even though the system satisfies the DARs of the jobs it accepts).

7.2 Commitment Maintenance Rate Indicator (CMR)

When the system determines that it can satisfy all DARs of a certain command, the system accepts such command for execution. But what if the system is slower than predicted or something in the environment changes (which is especially feasible in distributed information systems like data grids) and the system fails to satisfy a DAR? Probably, the user who submitted the corresponding command would become very unhappy, as he/she was expecting that the DARs would be satisfied (after all, the system agreed to satisfy all specified DARs when the user submitted the corresponding command).

The *Commitment Maintenance Rate* is a measure on the system's capability to maintain its commitments (i.e. satisfy the DARs from jobs it agreed to execute).

Let AJ represent the number of jobs that had associated DARs and the system agreed to execute. Let SJ represent the number of jobs with at least one DAR that were submitted to the system in Δt and whose DARs the system satisfied. The Commitment Maintenance Rate (CMR) Indicator represents the relation between the number of jobs that had all DARs satisfied (SJ) and the number of jobs that had associated DARs and the system agreed to execute (AJ).

$$CMR = \frac{SJ}{AJ}$$

where $0 \leq CMR \leq 1$.

A high value of Commitment Maintenance Rate is desirable as it can indicate that the system is highly dependable and is satisfying all the requirements that it commits itself to satisfy. Low values of CMR would certainly lead to low QoE and can be caused by several reasons, like a highly dynamic environment (where conditions are constantly changing) or by the use of bad algorithms to foresee the possibility to satisfy specified requirements.

If a system administrator wants to increase the values of Commitment Maintenance Rate, he/she can change parameters of the system's algorithm or the algorithm itself that decides concerning acceptance of users' jobs: for instance, the system can become more conservative and decline to accept a large number of concurrent queries or decline to accept too tight requirements. But if the used algorithm has a high risk aversion, the system can refuse a high number of users' jobs, which would not lead to high levels of QoE (i.e. high values of CMR, but low values of AR simultaneously).

7.3 Success Rate (SR) Indicator

When a user submits a command for a database system and explicitly defined a data access requirement, he/she expects that such command would be executed and that specified the DAR would be satisfied.

The *Success Rate Indicator* aims at providing a reference between the number of jobs with DARs the system satisfied and the number of jobs with DARs submitted by users.

The Success Rate (SR) Indicator represents the relation between the number of jobs that had all DARs satisfied (SJ) and the number of commands with DARs that were submitted by users (NJ).

$$SR = \frac{SJ}{NJ}$$

where $0 \leq SR \leq 1$.

Values of SR close to 1 are highly desirable, as they indicate that most of submitted jobs were executed and the corresponding DARs were satisfied. When SR is equal to 1, the system provides a high QoE. But when some of the submitted jobs do not have their DARs satisfied, the value of SR cannot be used as the measure of provided QoE. In Section 7.5, we present distinct situations that have the same value of SR but provide distinct levels of QoE to users. In the following Section, we present our proposal of indicator for measuring the QoE level a database system provides.

7.4 QoE-Level (QoEL) Indicator

Satisfaction is a subjective measure. However, in order to compare the levels of QoE provided by distinct systems (and scheduling strategies), we must have a somewhat rigorous indicator. Therefore, we define the QoEL indicator, which aims to provide information on the QoE-level provided by a QoE-oriented database.

In the proposed strategy for QoE-oriented databases, users submit database commands or blocks of commands with associated requirements for the system, which should evaluate if it can or cannot satisfy the specified requirements. The system only agrees to execute jobs with DARs that it can satisfy. This situation is somewhat similar to the situation of contracting a personalized service (e.g., home renovation or tailor made clothes).

Consider a person who wants to contract a service of home renovation. Such person specifies some requirements (e.g. deadline) that should be met by the service provider. The person would only hire a service provider that agrees to do the service while satisfying all specified requirements. Let us consider that the person searches for such a provider for some time, and then finds a provider who agrees to execute the required job. Then, the person hires such provider. This would certainly give some satisfaction to the person. Besides that, the person will certainly be happier when the provider delivers the hired job in accordance with specified requirements. On the other hand, if the person never finds a service provider that agrees to execute the specified job, he/she can feel somewhat frustrated, but can also change some requirement and start looking for a service provider again. The frustration of not finding a service provider that agrees to execute a certain job would indeed be smaller than the disappointment that the person would feel if he/she would hire a service provider, wait a long time for the service to be delivered and, then, the service would not be delivered as engaged.

Therefore, there are two important events that can lead (or not) to satisfaction: (i) service hiring; and (ii) service delivering. We believe that the execution of a command or block of commands (i.e. job) with associated DARs in a QoE-oriented database is similar to such situation. The *service hiring* event is related to the moment when the system agrees (or not) to execute a job. *Service delivering* represents the moment when the system finishes job execution satisfying (or not) associated DARs.

The QoE-Level Indicator (QoEL) is dependent on two factors: (i) service hiring (H) and (ii) service delivering (D) and is computed by the following Equation. Each factor may have a distinct importance to each person.

$$QoEL = \alpha * H + \beta * D$$

where $(\alpha + \beta) = 1;\ \alpha \geq 0; \beta \geq 0$

We use α and β as factors that calibrate events' importance. In most cases, service delivering would provide greater satisfaction (or disappointment) than service hiring (i.e. $\beta >> \alpha$).

The *service hiring* event can provide satisfaction when the system agrees to execute submitted jobs (positive factor). On the other hand, *service hiring* can provide

some degree of dissatisfaction when the system does not agree to execute a job (negative factor). Therefore, we can define *H* as dependent on *NJ* (i.e. number of jobs with DARs that were submitted by users, as defined earlier in this Chapter) and *AJ* (i.e. number of jobs with at least one DAR that were submitted to the system and the system agreed to execute):

$$H = \alpha_1 * AJ - \alpha_2 * (NJ - AJ)$$

where α_1 and α_2 are used to calibrate the importance of each factor. $\alpha_1 + \alpha_2 = 1;\ \alpha_1 \geq 0;\ \alpha_2 \geq 0$.

Similarly, the *service delivering* event can provide satisfaction when service is delivered as expected (positive factor) or dissatisfaction when the service is not delivered as expected (negative factor). Therefore, we define *D* as dependent on *AJ* and *SJ* (i.e. number of jobs that had all DARs satisfied).

$$D = \beta_1 * SJ - \beta_2 * (AJ - SJ)$$

where β_1 and β_2 are used to calibrate the importance of each factor. $\beta_1 + \beta_2 = 1;\ \beta_1 \geq 0;\ \beta_2 \geq 0$.

7.5 Using QoE-related Indicators to Evaluate Systems - Examples

In this Section we present some examples on the use of the above defined KPIs. In all the examples, we consider the same number of submitted commands and the same calibrating factors.

Consider that during a time interval of 10 minutes, users submit 1500 commands with DARs to the system (*NJ = 1500; Δt = 10min*). Suppose service delivery provides greater satisfaction (or disappointment) for users than service hiring, and calibrate the system with $\alpha = 0.4$ and $\beta = 0.6$.

In terms of service hiring, we should calibrate two factors α_1 and α_2. Consider that accepting a job results in some degree of satisfaction, but rejecting a job does not result in much dissatisfaction (i.e. users accept that the system is *doing its best* to satisfy DARs). Therefore, we choose $\alpha_1 = 0.8$ and $\alpha_2 = 0.2$.

We should also define the values of the factors related to service delivery (i.e. β_1 and β_2). Consider that, when the system fails to satisfy a DAR that it promised to satisfy, it causes more disappointment than the satisfaction it provides when delivering a job satisfying the specified DAR. Hence, we would consider $\beta_1 = 0.3$ and $\beta_2 = 0.7$.

In the following, we present five distinct examples. Table 18 presents a summary of the parameter considered in proposed situations and the values obtained for each of the proposed indicators.

Table 18 - KPIs - Examples

Example	Accepted Jobs (AJ)	Jobs with Satisfied DARs (SJ)	Acceptance Rate (AR)	Commitment Maintenance Rate (CMR)	Success Rate (SR)	QoE Level Indicator (QoEL)
7.1) Best Effort Oriented System	1.500	100	1.00	0.06	0.06	-90
7.2) Conservative QoE Oriented System	200	200	0.13	1.00	0.13	-4
7.3) Daring QoE Oriented System	1.400	700	0.93	0.50	0.47	272
7.4) Balanced QoE Oriented System	1.240	1.200	0.83	0.97	0.80	575
7.5) Conservative QoE Oriented System II	740	700	0.49	0.95	0.47	285

Example 7.1 – Best Effort Oriented System

Consider a *best effort* system. The system accepts all users' commands that have an associated DAR, even though such DAR is not explicitly defined (*best effort* systems do not deal with DARs). It achieves a high value of AJ (AJ = 1,500). Unfortunately, it cannot satisfy all such commands and a low value of SJ is obtained (SJ = 100).

In such situation, a high Acceptance Rate is obtained (1.00). But the values of CMR and SR are too low (0.06 for both indicators). The system may present high throughput or low response time, but it satisfied only a few DARs. Hence, the system provides a low QoE level, which is represented by the value of QoEL (-90).

Example 7.2 – Conservative QoE Oriented System

Now, consider a QoE-oriented database system. Such system is calibrated in order to be very *conservative* when evaluating if it can or cannot satisfy commands' DARs (i.e. high risk aversion). Therefore, the system refused to execute most users' commands: it executes only 200 commands (AJ = 200). With such configuration, the system fulfills all DARs of the jobs it executes (SJ = 200).

Hence, the system appears quite dependable to users as it fulfills all DARs that it committed itself to satisfy. But, although a high Commitment Maintenance Rate is obtained (CMR = 1.00), users may feel that just a small number of their commands were executed (SR = 0.13), which leads to a low level of QoE (QoEL = -4).

Example 7.3 – Daring QoE Oriented System

The QoE-oriented system uses a new method to estimate if it can or cannot satisfy users' DARs. Using this method, the system agreed to execute 1.400 commands (AJ =

1400). It is a lower number of executed commands than in Example 1 but a higher number of executed commands than in Example II. Consider that in such configuration the system satisfied all DARs from 700 jobs (SJ = 700).

The Acceptance Rate value of this Example is smaller than in Example I. It has a smaller value for Commitment Maintenance Rate than the one of Example II. But the values for Success Rate (0.47) and QoE-Level (272) are higher in Example III than the ones of previous examples. This is because the configuration of Example III leaded to a better balance between command acceptance and requirements fulfillment than the ones obtained in previous examples.

Example 7.4 – Balanced QoE Oriented System

Now consider a situation where the number of accepted jobs (AJ = 1240) is somewhat smaller than the one of the previous example, but almost all accepted jobs had their DARs satisfied (SJ = 1200). About 80% of users' commands that were submitted had their DARs satisfied (SR = 0.8). This situations leads to a higher QoE level (QoEL = 575) than the previous examples.

Example 7.5 – Conservative QoE Oriented System II

In examples I to IV, an increase in the value of Success Rate provided an increase in the value of QoE-Level, which indicates that there is a relation between SR and the provided QoE level. But situations with the same value of SR can have distinct values of QoEL. This happens because it is assumed that users give distinct importance to the result of each *phase* of command execution (i.e., success or failure in service hiring and delivering).

Consider a configuration where the number of accepted commands is 740 (AJ = 740) and the number of jobs whose DARs where fulfilled is 700 (SJ = 700). The Success Rate of such configuration is equal to the obtained in Example III (SR = 0.47), but the value of QoEL (285) is higher than the one of Example III (272). Although the system had satisfied the same number of DARs in examples III and V, in Example III it failed to satisfy its promises much more times than in Example V (i.e., 700 failures in Example III and 40 failures in Example V).

7.6 Using QoE-related Indicators to Alert Administrators

In previous chapters, we presented several situations where the QoE oriented database system alerts the administrators, including: (i) when the reputation of a certain service is undesirable (Section 5.4); (ii) when the system detects that replication can improved the provided level of QoE (Section 5.6); and (iii) when undesirable unavailability is detected (Section 5.7). In this section, we discuss the use of proposed QoE-related indicators to alert administrators when the system is providing an undesirable level of QoE.

In several situations, systems' administrators do not know that users are unsatisfied with the system until it is *too late*. For instance, consider a web-based store. At a certain moment, the response time of the system becomes too high (for instance,

because of an inefficient database query included by a programmer in a certain release of the application or because the number of uses is too high for existing infrastructure). In such situation, users may become unsatisfied with the system. Nevertheless, it may take a long time for users to inform the store's owner that they are unsatisfied with the system's performance. In fact, most users can stop using the application and never inform the store's owner that they are unsatisfied with the system's performance.

Proposed specialized performance indicators are also used to alert system's administrator when the system is providing low levels of QoE, even before a large number of users is affected or before users start complaining about the system. Therefore, examples of some possible alerts are:

- The value of Acceptance Rate in a certain time window is below the acceptable level – usually, it indicates that users' requests are unfeasible with existing resources. Some possible actions are: (i) database tuning; (ii) client application tuning/refactoring; (iii) add new hardware; (iv) users may need to be trained in how to use available DARs. AR would also be low if the strategies used by data services to foresee future conditions are too *conservative* (data services are refusing to be candidates to execute tasks even though they have resources to execute them).
- The value of Commitment Rate in a certain time window is below the acceptable level – the system is not being able to confirm its commitments. Some possible reasons are: (i) the system is suffering interference of other software that uses the same infrastructure; (ii) strategies used by data services to foresee future conditions are not accurate.
- The value of Success Rate is below the acceptable level – there may be one or more of the problems related for above listed two indicators.
- The value of Success Rate is high (near 1.0), but the value of QoEL is near zero – system utilization is too low: high SR values indicate that (almost) all submitted jobs are being executed, but a QoEL near zero indicates that a low number of jobs is being executed.

7.7 Conclusion

Traditional performance indicators are not capable to measure the level of QoE a system provides. Therefore, they cannot be used to compare the *performance* of database systems in terms of QoE. In this Chapter, we present four specialized performance indicators to be used in QoE-oriented databases. Three of them (i.e. *Acceptance Rate*, *Commitment Maintenance Rate* and *Success Rate*) are oriented for specific aspects of DAR evaluation and fulfillment. The forth indicator (QoE-Level) aims to provide a measure of the level of QoE a system provides.

Therefore, proposed indicators can be used to measure the performance of QoE-oriented databases, which is not only useful to compare database systems, but mostly to help identify, thorough experimentation, the techniques and algorithms that can provide higher levels of satisfaction to users.

In fact, as data access requirements may exist even though users are not capable to explicitly define them to a database system, proposed indicators can even be used to estimate the levels of QoE that *traditional best effort*-oriented systems provide.

Proposed indicators are also used to alert systems' administrators about the level of QoE the system is providing to users. This permits that administrators take corrective actions before users become too unsatisfied.

In the next chapter, we present some experimentally obtained results on the use of a QoE oriented database system.

8 Experimental Evaluation

In this chapter, we present the results of the experimental evaluation of the QoE mechanisms that were proposed in this thesis. The objective is to show that election scheduling, reputation and the use of data access requirements (DARs) are useful to improve the Quality of Experience provided by database systems (offering added control over how things execute), and to analyze how parameters such as reputation vary with varying conditions. Next we will discuss methodology, metrics and scenarios used in the experiments.

Methodology

In order to assess the QoE-related mechanisms, we designed a set of experiments that use some of the most important concepts that were proposed in this thesis, provide quantitative results and show their importance. The approach was based on a prototype of the proposal that we implemented, and lab experiments over benchmark data. We used two benchmark databases (TPC-H [TPCH, 2010] and TPC-W [TPCW, 2010]), and two database management systems (Oracle 11g R1 Enterprise Edition [Oracle, 2010] and SQL Server 2008 Express Edition [SQL Server, 2010]).

The experiments were designed by setting up three main experimental scenarios where the mechanisms would be useful to provide QoE expected behavior to the user. The mechanisms were compared with best effort (no QoE) counterparts and, when relevant, with scheduling approaches such as round-robin or on-demand. Besides such scenarios, we also conducted some experiments to evaluate specific aspects of proposed strategies. We also defined a set of metrics that were evaluated, and then we analyzed the results and concluded on the relevance of the approaches that were proposed in this thesis.

Metrics

- Execution time – Job execution time is measured starting at the moment of job submission and ending on job execution finish time. Since a job may have to wait in a queue before it is executed, this metric includes the queue wait time;
- Success rate (SR), acceptance rate (AR), commitment maintenance rate (CMR) – success rate is the ratio of submitted jobs whose DARs were satisfied, AR is the ratio of submitted jobs that the system agreed to execute, and the commitment maintenance rate measures how many of the accepted commands had their DARs satisfied. These metrics are formally described in Chapter 7;
- QoE-Level (QoEL) – this metric aims to express the level of QoE provided by the database system. It considers three values (formally described in Section 7.4):

- NJ: number of jobs with DARs specified;
- AJ: number of jobs with DARs which the system agreed to execute, promising to satisfy the DARs;
- SJ: number of jobs whose DARs were satisfied.

These metrics are weighted using a weighted sum as described in Chapter 7 (we use in this chapter the same weigh values used in the examples of Section 7.5).

- Reputation – this metric indicates an expectation about data service's behavior in terms of maintaining its commitments. Reputation is between 0 and 1. Highest reputation is 1 and lowest reputation is 0. As data services fail to maintain their commitments, their reputation decreases; on the other hand, as services stick to their commitments, their reputation increases; finally, aging reduces the weight of commit or fail events in the determination of reputation of a data service. These concepts were defined and discussed in detail in Section 5.4.

Scenarios

The proposed QoE mechanisms can be applied in quite different scenarios to provide qualities such as execution time control, availability or others. We have setup three different contexts to test the mechanisms. Besides these, we also added experiments testing certain specific aspects of the proposed mechanisms. The three main scenarios are:

1. A globally distributed data warehouse was designed as a set of sites with a varying number of machines that register local regional sales. There are regions for Africa, America, Asia and Europe, with different sales volumes and computing resources. We consider that users can place DARs together with the queries and assess how the system is able to adapt to improve the processing over two main DARs: execution time and availability. There is also a query workload where queries execute over more than one site.
   1.1. Execution constraints over distributed query execution: In the first case a site will be unable to meet the timing requirements specified in DARs, therefore refusing to accept the queries that would otherwise be over the time limit. The QoE system will decide to create a replica of the data from the slow region into a better equipped region, then the queries will be able to meet the required timing and there will be a balancing of execution over the sites;
   1.2. Availability and freshness: In the second case we test what happens when a site holding a region's data is unavailable. Queries that require that region's data will not be able to run. The scheduling system will detect the unavailability and decide to copy the site contents into other site(s). From then on the queries will be able to run.
2. A parallel data warehouse was designed to run over a set of off-the-shelf computers, data replication allows inter-query parallelism and intra-querying parallelism is enabled by the slicing of the main table (i.e. TPC-H's `LINEITEM`) into 100 pieces.
   2.1. Execution time: the 100 pieces of the main table were divided into nine database machines, and the workload has long-running and short-running queries. Most queries will fail to honor their specified deadline when using either round-robin or on-demand scheduling. We show that execution time

DARs over the proposed QoE-aware system allow the system to meet the deadlines of most queries simply by choosing the best moment to execute the heaviest queries, which would otherwise fail their deadlines and also make most of the remaining queries fail them as well.

2.2. Autonomics: In this setup we will show that our approach is able to automatically size a system to needs. Starting with a single node with all the data, the system will determine that too many queries will be rejected due to node overload, so that if and when there are new nodes, the system will automatically copy fragments into the new node(s). We will show that the system will be able to adapt by incrementally adding more resources (if those are or become available);

3. An OLTP web server that should provide good response time for both long running and short running queries is evaluated. We considered a set of user transactions that are submitted to the system considering several distinct submission rates. Workload transactions have both execution time and execution priorities constraints. We will show how execution constraints will enable most short transactions to be executed even though the system cannot execute all large transactions. Then, system administrator is alerted and, after physical tuning, the entire workload can meet the specified requirements.

Besides these three main scenarios, the following experiments were made to analyze specific features of the approach:

- Reputation tests: in this setup we evaluated how the system adapts itself when participating data services misestimate conditions, making promises that cannot be fully accomplished. We replicated data across nine data services and submitted a workload of hundreds of jobs. We configured three of the data services so that they would accept to execute every job and promise to finish each of them immediately – simulating an overly *optimistic* condition. We analyze how reputation varies along time in this scenario and how the system is able to adapt in the presence of data services that do not estimate well;
- Queue organization analysis: the scheduler proposed and implemented in the prototype is able to use multiple queues in order to differentiate on job sizes. This experiment analyses different queue decision parameters. It was necessary to find out the most appropriate parameters for the queues. In such context, we also analyze the quality of time estimations, considering distinct job sizes.

The remaining of this chapter is organized as follows: Section 8.1 presents the experimentally obtained results on the use of QoE mechanisms over the global warehouses environment. In Section 8.2 we present the results of the second tested scenario: parallel warehouse. Section 8.3 presents the results obtained in the tests that ran over the OLTP scenario. In Section 8.4 we present the analysis of specific features tests. Finally, Section 8.5 concludes the chapter.

A detailed description of the used testedbed environments is included in the Appendix A.

8.1 Scenario I: QoE in Distributed Databases

In this scenario, we evaluate the use of proposed techniques over distributed warehouses. We conducted two main sets of tests: the first one (presented in Section 8.1.1) evaluates execution time constraints and data replication for distributed query execution, while the second set of tests (Section 8.1.2) evaluates the use of DARs to improve data availability and enable query execution even in the presence of constant site failures.

In this scenario, we used TPC-H's database and queries, and Oracle 11gR1 DBMS.

8.1.1 Execution constraints over distributed query execution

In this set of tests, we consider a distributed warehouse context, composed by three main sites: Europe, Africa and Asia. Each site stores data about sales in its region. Europe users are querying data about the sales of Africa and Asia. A single community scheduler is used, while the Asia site has 5 data services and Africa has 3 data services. Each data service has its own tasks scheduler. Although the processing power of Asia is 66% greater than the one of Africa, the size of the database stored at each site is almost the same. Such scenario is represented in Figure 57.

Europe users periodically submit a job workload composed by 6 jobs. Each job is a query of TPC-H: the workload is composed by queries 1, 17 and 2, each one repeated two times (in the specified order). Due to table partitioning (which is detailed in Appendix A), TPC-H's queries 1 and 17 are transformed into 100 tasks each (50 for each site), while query 2 of TPC-H is transformed into two tasks (one for each site). There is an interval of 20 seconds between each job submission. Each job accesses both data from Asia and Africa. Appendix A presents a detailed description of the testbed environments used in the experiments.

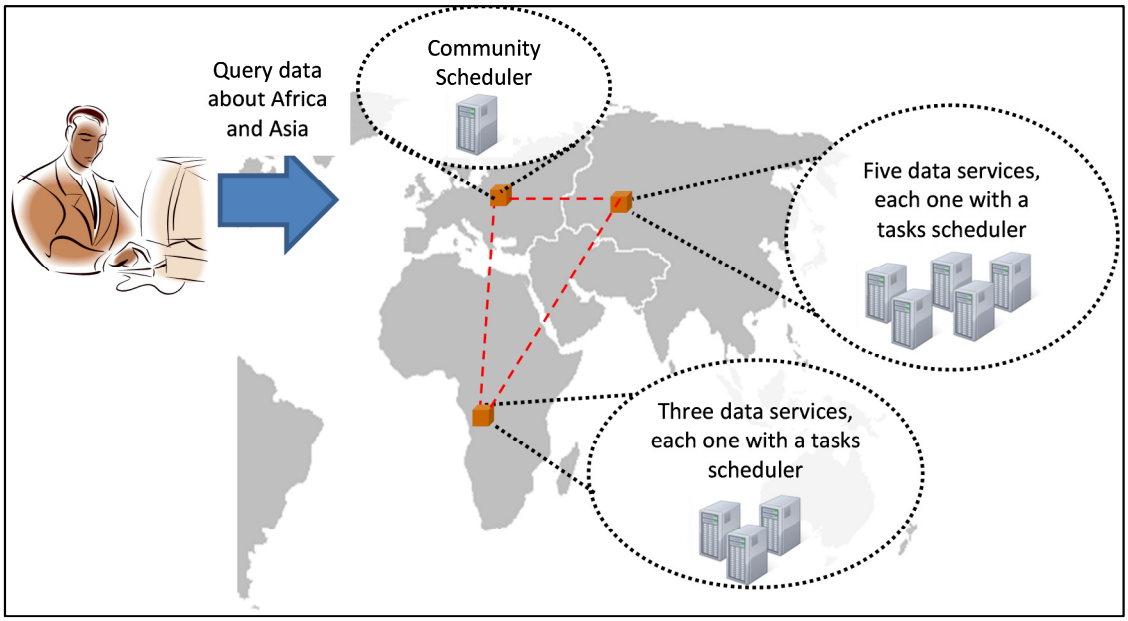

Figure 57 - Scenario I: testing execution constraints over distributed query execution

Distributed query execution without DARs

First, we ran the considered workload using no DARs. Some jobs take a long time to execute, which makes the system fail specified constraints, making users unhappy.

Distributed query execution with DARs

Next we ran the same workload using a 10 minute execution deadline interval DAR and election-based scheduling, and considering dynamic replication between Africa and Asia sites. Figure 58 presents the clause that is added to each job to specify the deadline. What-if elections (section 5.6) are enabled to identify when data replication would improve the QoE level the system provides.

```
REQUIREMENTS
 DEADLINE 600
```

Figure 58 - REQUERIMENTS clause especifying 10 minutes deadline

Experimental Results

Figure 59 presents the measured execution time of each job in three distinct configurations: (i) when no DAR is used; (ii) when DARs are specified but replication has not occurred; and (iii) after dynamic replication.

When no DAR is used, three jobs take more than 10 minutes to execute.

Then, DARs and what-if elections (for dynamic replication) are applied. Before replication, the system does not execute jobs 2 and 5, as it estimates that their execution would take more time than the specified deadline. Figure 59 also presents the execution time of each job in this case. Each job is partially (distributed) executed at Africa and Asia sites. In such configuration, the execution time of job 6 falls down to barely 28% of what it takes to execute such job when no DAR is used.

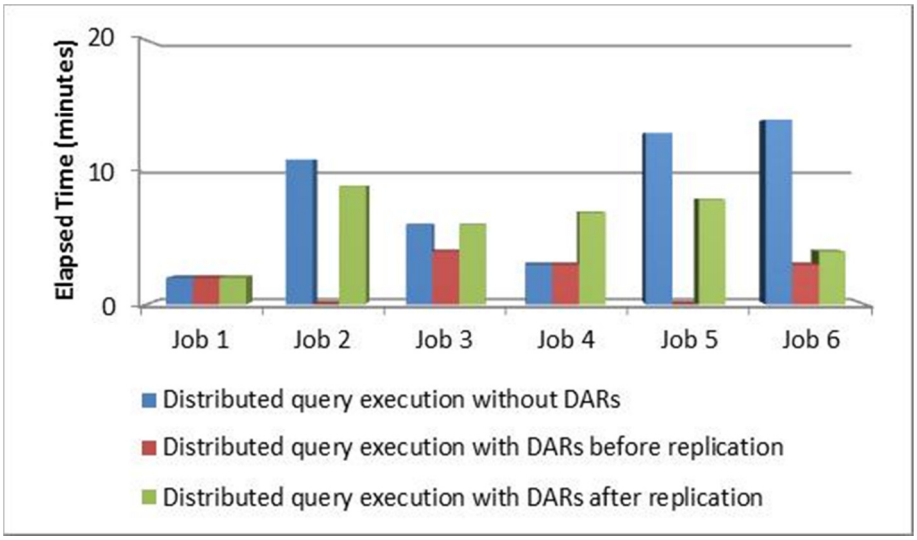

Figure 59 - Job mean execution time - Distributed query processing

In Figure 59, the series “Distributed query execution with DARs after replication” refers to the times taken by jobs after the data sets are replicated. Replication is suggested to the administrator by the system based on a replication benefit decision strategy described in section 5.7.

The replication benefit decision is taken by the system based on what-if elections and execution information. In order to compute the total benefit of replica creation, the system sums the contributions of the replica for individual job executions considering what-if scenarios, where it evaluates what would happen if certain replicas existed.

In this experiment both jobs 2 and 5 access tables `LINEITEM` and `PART`, and workload is periodically executed in intervals of an hour. The system should alert the administrator when the total benefit of replica creation (measured by the what-if elections as described in section 5.6) reaches the threshold value, which was set to 5 in these experiments (this parameter should be specified by the administrator). Such value is computed based on the number of times that the system would satisfy a DAR of a job that the system rejected to execute in a what-if scenario with replication. In this computation an exponential time decay function is used to differentiate old executions from newer ones (the importance of a job rejection is reduced 40% at each hour).

Figure 60 presents the replica creation benefit value computed automatically at each hour based on what-if analysis. In such figure we detail the contribution of each workload execution to the computation of the total replica benefit value, and for each workload execution we also show its influence decay on the total replica benefit value (each quota line in the figure). The total benefit line is the sum at each instant of the individual quotas. After five workload executions, the system alerts the administrator that replica creation should occur.

Figure 60 - Benefit of replica creation

After the administrator is advised, tables `LINEITEM` and `PART` are replicated. Figure 59 also presents the execution time of each job execution after table replication. In such configuration, all jobs can be executed by their specified deadlines.

Figure 61 presents the values of AR, CMR and SR for the three configurations. When no DARs are specified, the system accepts to execute every job (AR = 1.0). But just half of them are executed by the desired deadline interval (CMR = SR = 0.5). When execution deadline DARs are specified, the system rejects the execution of two jobs (AR = 0.67) but the number of jobs executed within the desired deadline is increased (SR = 0.67). After replica creation, all jobs are executed within the desired deadline (SR = 1.0).

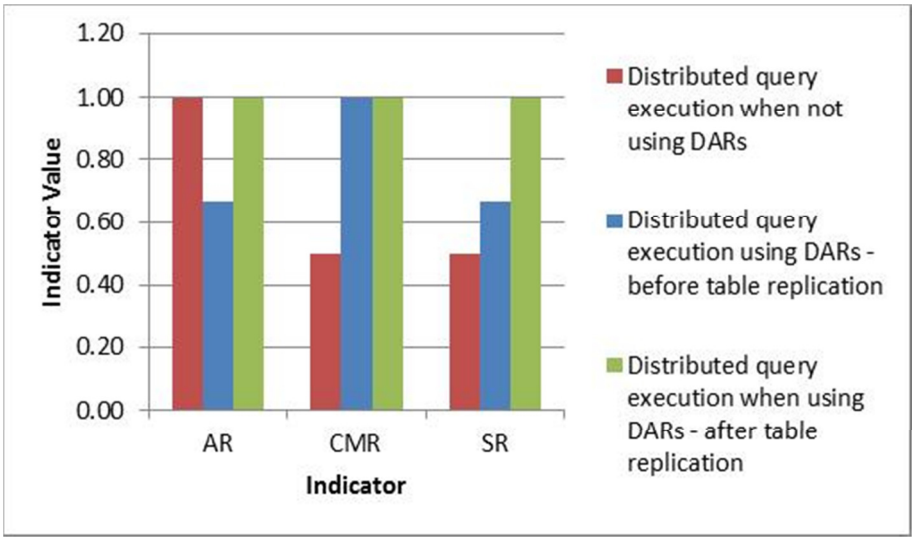

Figure 61 - AR, CMR and SR - KPI values when using and when not using DARs

These results have shown that the proposed QoE approaches are useful to help provide a better service to users in a distributed (global) data warehouse context. With the necessary DARs, the system was able to avoid a bad service to the user (according to his requirements) and it has also adapted to provide the best possible service.

8.1.2 Availability and freshness in the global warehouse

In this tests set, users from Europe are querying data about sales in America and Africa. America's site has five data services while Africa's site has three data services. Each data service has its own tasks scheduler. A single community scheduler is used, as represented in Figure 62.

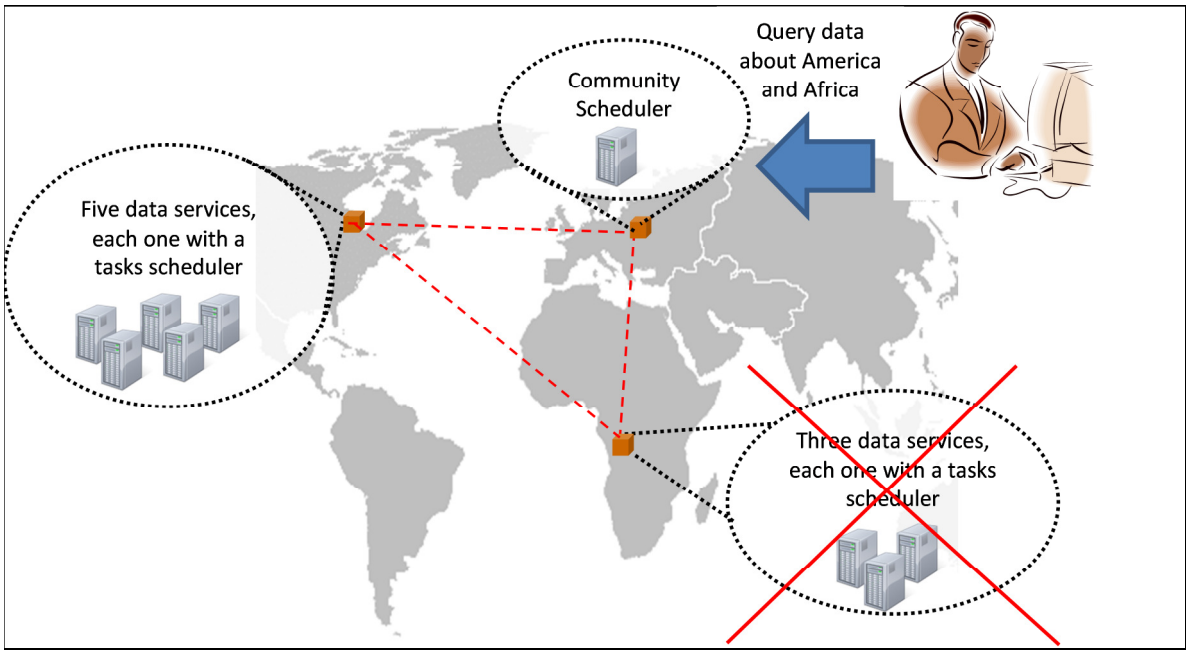

Figure 62 - Scenario I: testing availability and freshness DARs in distributed warehouses

The site which holds Africa's data is suffering from constant failures, which leads to data unavailability to remote sites.

Europe users periodically submit a job workload composed by 6 jobs. As in the previous set of tests, each job is a query of TPC-H and the workload is composed by queries 1, 17 and 2 (each one repeated two times in this order). Jobs related to queries 1 and 17 are transformed into 100 tasks each (50 for each site), while jobs related to query 2 of TPC-H are transformed into two tasks (one for each site). There is an interval of 20 seconds between each job submission. Each job accesses both data from America and Africa. Appendix A presents a detailed description of the testbed environments used in the experiments.

Distributed query execution without DARs

User submits a workload to be executed while Africa's site is unavailable and none of workload's jobs can be executed.

Distributed query execution with DARs

In this case, the system administrator specifies an availability DAR of 99.9% for tables stored at Africa's site from 8AM until 7PM at London. Figure 63 presents the DAR's specification, considering the `LINEITEM` table.

```
ALTER TABLE LINEITEM
  ADD REQUIREMENTS
    AVAILABLE DURING 99.9 PERCENT
    EVERY WEEK FROM MONDAY TO FRIDAY
    IN PERIOD FROM '08AM' TO '07PM'
    SYNCHRONOUS
```

Figure 63 – Availability requirement example – LINEITEM table

Experimental Results

The system starts to monitor the availability of Africa's site; testing the site availability every 1 minute (such interval can be configured). We implemented the availability monitoring (discussed in Section 5.7) as a *telnet* test. At the end of the day, several unavailability periods are detected, as represented in Figure 64 (where 1 indicates that tested site is available and -1 indicates that the site is unavailable).

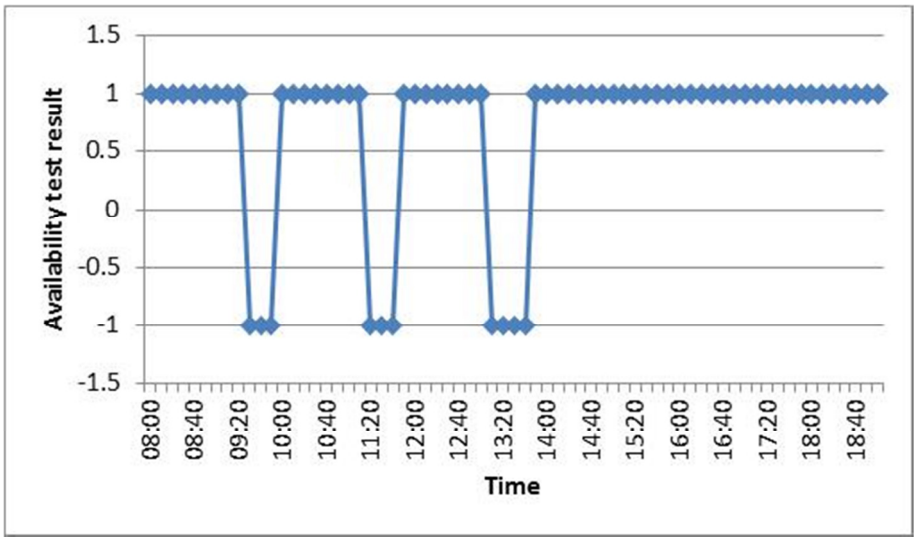

Figure 64 - Availability tests results

At the end of the day, the unavailability period reaches almost 14%. As a consequence, table replication takes place. Africa's tables are replicated to America's site. But if Africa becomes unavailable to America's users, then there is no guarantee that data stored at America about Africa is up to date.

In order to use such Africa's data stored at America for query execution users specify the data freshness requirement that is acceptable. Figure 65 presents an example on the use of such DAR, which specifies that any replica of `LINEITEM` table that corresponds to July 1st, 2010 in the master table can be used to answer the query. In an application program, users may specify the freshness requirement when querying for a report (e.g., ask for up to date data or inform the acceptable freshness).

```
REQUIREMENTS
    FRESHNESS OF LINEITEM HIGHER THAN '2010/07/01'
```

Figure 65 – Data Freshness requirement – Example for LINEITEM table

With the use of availability and data freshness requirements, the system can execute users' jobs even though Africa is unavailable. Figure 66 presents the execution time of each job in such situation. One series represents the no-DARs case – no job is executed since Africa is unavailable – the other series represents the use of DARs in this context – in this case all jobs are executed (all jobs are executed at America's site, as Africa is unavailable and the data stored at America's site satisfies specified freshness requirement).

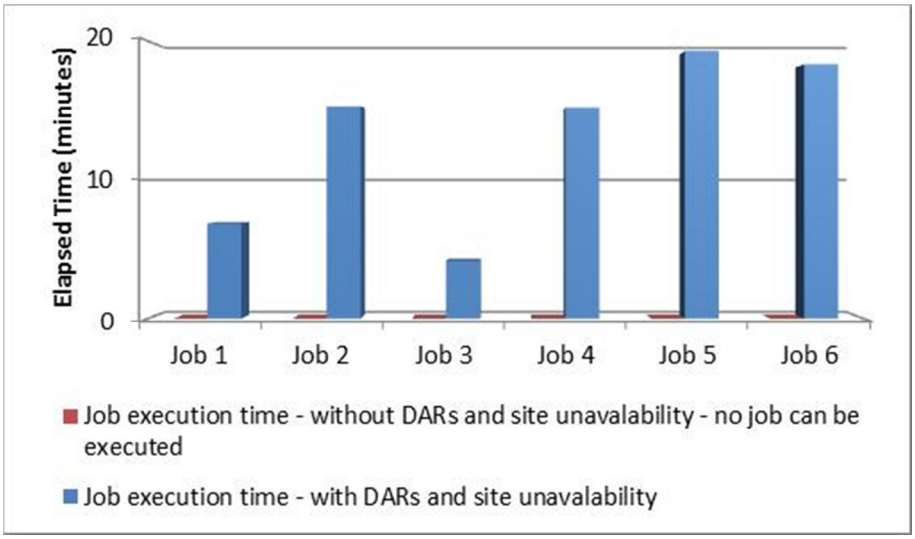

Figure 66 - Distributed query execution time - With and without DARs

These results have shown how DARs are useful to improve data availability, enabling users to generate reports whose execution would otherwise fail due to frequent unavailability of some site.

8.2 Scenario II: Parallel Warehouses and QoE

In this Section, we consider the scenario of a 20GB parallel warehouse built over a cluster of off-the-shelf computers. In this scenario, we used TPC-H's database and queries, and Oracle 11gR1 DBMS. TPC-H's `LINEITEM` table is horizontally partitioned into 100 fragments by ranges of the `L_ORDERKEY` column. `LINEITEM` partitions and the other TPC-H tables are replicated at cluster's nodes.

The workload is composed by queries 1, 11, 5, 7, 14, 17 and 2 of TPC-H. Each of these queries is a job and may be transformed into several tasks (depending if it

accesses or not partitioned data). Jobs were submitted to the system with a 25 seconds interval.

In the first tests set (Section 8.2.1), DARs are used to indicate to the system when jobs can be executed. Without DARs, long-running jobs would fail their deadlines and also cause that short-running jobs to fail theirs. When users' expectations are expressed with DARs, the system chooses the best moment to execute each job, satisfying all specified DARs and increasing the provided level of QoE.

In the second tests set (Section 8.2.2), we use the proposed strategies to decide on data placement over a database cluster. A small cluster, composed by three nodes is used to execute users' jobs. But such system cannot fulfill all specified requirements. Then, two new nodes are added to the system, which indicates when table replication can be used to increase the requirements fulfillment rate.

8.2.1 Choosing when to execute jobs based on DARs

In this tests set, we use nine database nodes, each one with its own tasks scheduler, and one community scheduler, as represented in Figure 67.

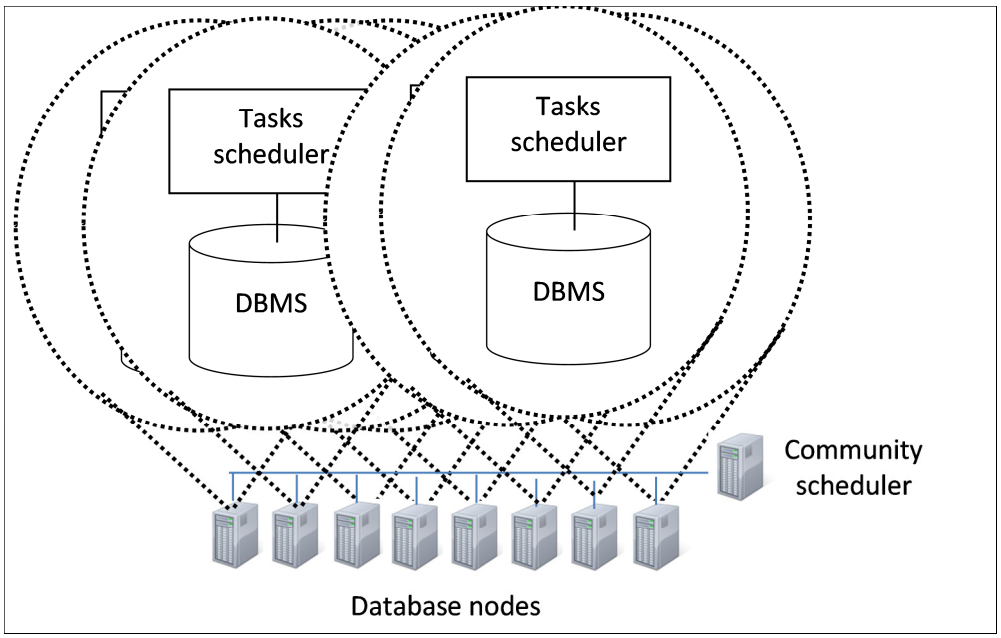

Figure 67 - Scenario II: Evaluating DARs at cluster of off-the-shelf computers

Using best-effort scheduling strategies

First, we ran users' workload using two distinct best-effort strategies:

- Round-robin – each job is transformed into tasks, and those tasks are assigned to database nodes in a round-robin fashion. This strategy aims at assigning the same number of tasks to each database node;
- On-demand – each job is transformed into tasks that are assigned to nodes when the nodes can execute them (i.e. the number of tasks being executed by the node

is bellow a threshold number). This strategy aims at increasing the load balance level (as no node remains idle when there still exist tasks to execute).

Using DARs to improve the level of QoE

Then, we ran our QoE-oriented scheduling strategy, considering that each job has execution related DARs, as the ones exemplified in Figure 68. Such DARs indicate that the system should finish job execution in no more than 15 minutes, otherwise the job should be executed at night (after 19 PM of the current day and before 08 AM of the day after) and its results must be stored and available for users during the next day.

```
REQUIREMENTS
    (DEADLINE 900)
    OR
    (START AFTER '2010/12/15 19:00',
    FINISH BEFORE '2010/12/16 08:00',
    EXECUTE DISCONNECTED RESULTSET IDENTIFIED AS TMP_SALES,
    AVAILABLE DURING 100 PERCENT
        IN PERIOD FROM '2010/12/16' TO '2010/12/17'
```

Figure 68 - Specifying multiple DARs to jobs - Example

Experimental Results

Figure 69 presents the execution time of each job when using the three considered scheduling strategies (i.e. round-robin, on-demand and QoE-oriented using DARs). When executing users' workload using best-effort oriented techniques, 71% of the jobs take more than 15 minutes to execute. But when using the proposed election-based QoE-oriented scheduling strategy together with the proposed DARs, the system immediately starts the execution of five types of jobs, which are executed in much less than 15 minutes (as shown in the figure), and lets two longer types of jobs to be executed at night. Figure 70 presents the execution time of those long-running jobs, that are executed in disconnected mode and whose results are stored for future access by users.

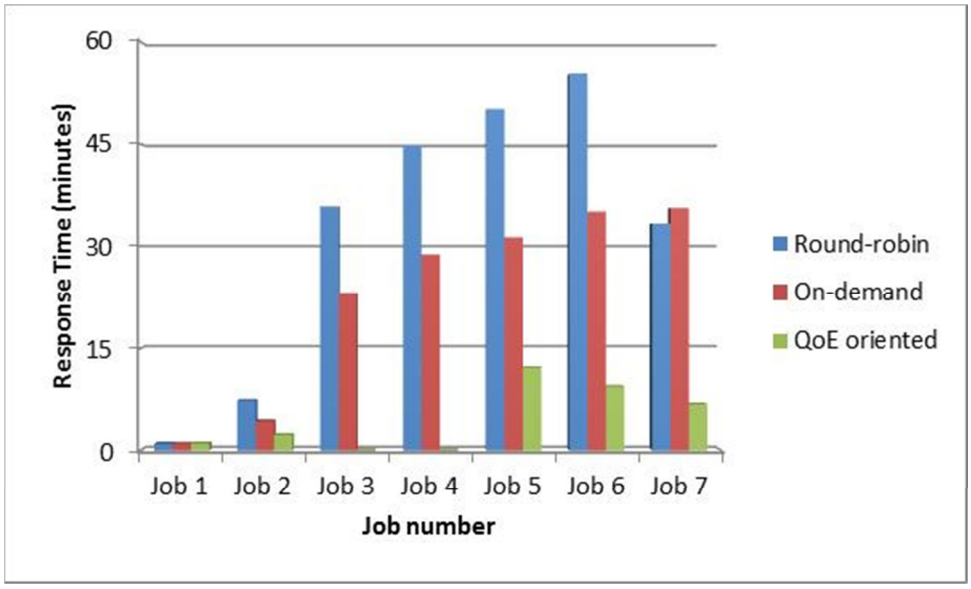

Figure 69 - Mean execution time for each job using several scheduling strategies

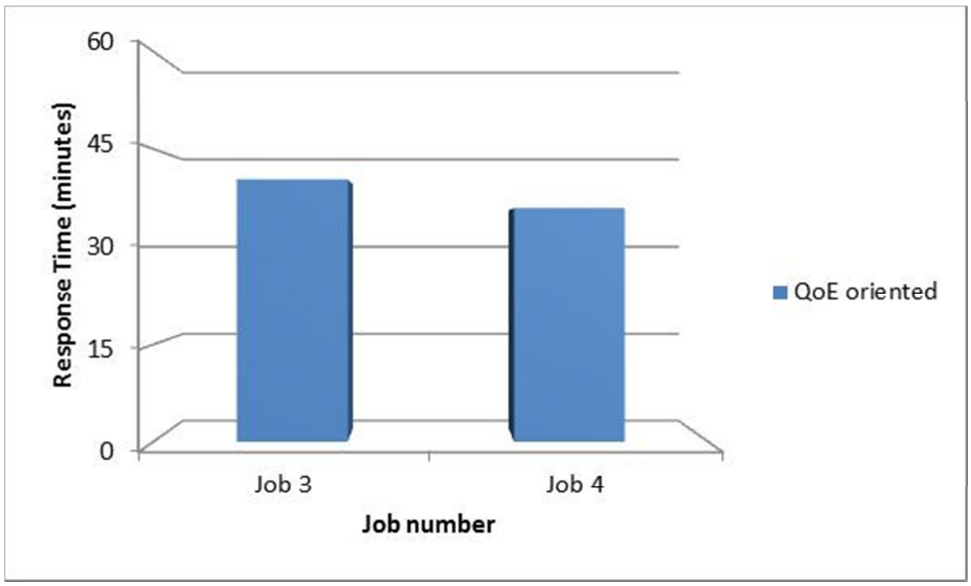

Figure 70 - Execution time of long-running jobs

Table 19 presents the values of AR, SR and QoEL for each configuration. All strategies executed the entire workload (AR = 1.0). But the QoE oriented scheduling strategy did not immediately start the execution of jobs of two types, which let it fulfill the 15 minutes deadline of the remaining five types of jobs and executed the two long-running ones in alternative time windows (SR = 1.0). On the other hand, best-effort strategies immediately start the execution of every job, failing to fulfill the deadline of three of the five short-running jobs due to the execution of the two-long running jobs (SR = 0.28). In such configuration, the QoEL obtained for the QoE oriented environment was 7 times greater than the one obtained when using best-effort scheduling strategies.

Table 19 - AR, SR and QoEL for several configurations

Scheduling Strategy	Acceptance Rate (AR)	Success Rate (SR)	QoE Level Indicator (QoEL)
Round-robin	1.00	0.28	0.5
On-demand	1.00	0.28	0.5
QoE oriented	1.00	1.00	3.5

This set of tests has shown how the QoE oriented system improves users' satisfaction with DARs. By considering alternative DARs specified by users, the system could choose the best moment to execute each job and meet users' expectations.

8.2.2 Autonomic behavior: placing data in database clusters

In this test set we evaluate the capability of the system to automatically size itself to user needs. The system is initially composed by three database nodes and a community scheduler. Each database node has its own tasks scheduler and database management system. `LINEITEM` partitions and the other TPC-H tables are replicated at database nodes. User's workload is composed by 4 jobs, each one being a TPC-H query. The queries used are 1, 11, 14 and 2. Each job has an execution deadline of 5 minutes.

Then, new nodes are added to the system. At each moment, the system automatically evaluates if the additionally available resources would be useful to increase requirements satisfaction rates (based on what-if elections). Based on that, the system makes suggestions to the database administrator about data replication.

Experimental Results

In the three nodes configuration, some of workload's jobs cannot be executed by specified deadlines, as represented in Figure 71.

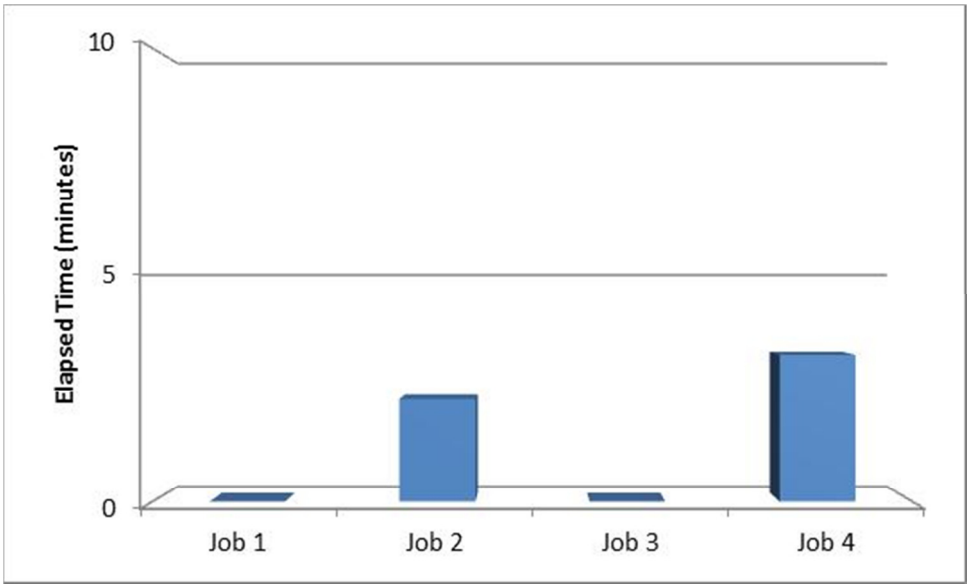

Figure 71 - Job execution time in three nodes configuration

Then, two new empty nodes are added to the system and the system is configured to consider such nodes for replication. Then, the system evaluates that data replication can be used to improve requirements fulfillment rate using the what-if elections strategy (defined in Section 5.6).

Job 1 accesses the `LINEITEM` table. Considering that the replication can take place when the replica benefit is over 3, then replication is advised after 4 workload executions, as represented in Figure 72 (in the exponential time decay function, the importance of a replica creation is reduced 40% at each hour, as used previously in this chapter). This figure represents the total benefit value of replication and the quota contributed by each what-if evaluation on workload executions, including the decay of those contributions. The total benefit value shown in the figure is the sum of the individual contributions at each moment, considering the decay as well.

Figure 72 - Benefits of replica creation

As the `LINEITEM` table (used by job 1 and whose replication was indicated by the system as presented above) is partitioned, the table partitions are distributed across the two empty nodes and each new node will store 50 partitions from the initial 100 ones). With such new configuration, the system uses 5 nodes to execute the workload and is capable of satisfying the deadline requirement of Job 1, as represented in Figure 73 (besides satisfying the DARs of jobs 2 and 4, which were already satisfied in previous configuration). Job 3 deadline is still not satisfied yet in this configuration. The user would have to reconsider the deadline of that job, or to consider adding alternative requirements to the job (i.e. allowing the job run at an alternative time), adding additional nodes or using some other strategy to run that job as well.

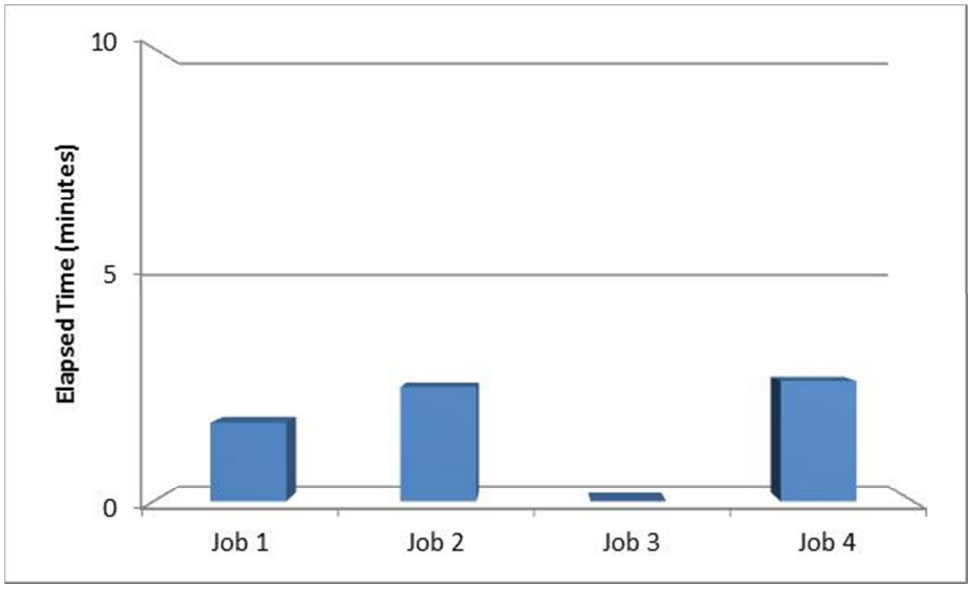

Figure 73 - Job execution time in five nodes configuration

In this set of tests, we have shown how proposed election-inspired mechanisms and DARs can be used to suggest data placement over a parallel infrastructure. The suggested placement (which is not full replication) improved the number of DARs the system satisfied, therefore increasing the level of QoE provided by the system.

8.3 Scenario III: DARs for QoE in OLTP Applications

In this section, we consider the scenario of a centralized web-based OLTP application. We use a database machine and an application server, which submits several transactions to the database machine according to distinct submission rates and using distinct scheduling strategies. Such scenario is represented in Figure 74. First, we use best-effort scheduling, which makes most transactions fail their requirements. As an alternative, we also used an admission control system, which rejects the execution of most transactions due to the requirements failing. Then, we use our QoE oriented strategies that increases the QoE level the system provides. It increases the number of satisfied DARs and informs the system administrator about transactions whose requirements cannot be satisfied. After database physical tuning, all requirements can be satisfied. Besides that, in such scenario we also evaluated the use of high priority transactions, which reduces the mean execution time of such type of transactions when compared to their counterparts that have normal priorities.

In this scenario, we used TPC-W's database and queries, and SQL Server 2008 DBMS.

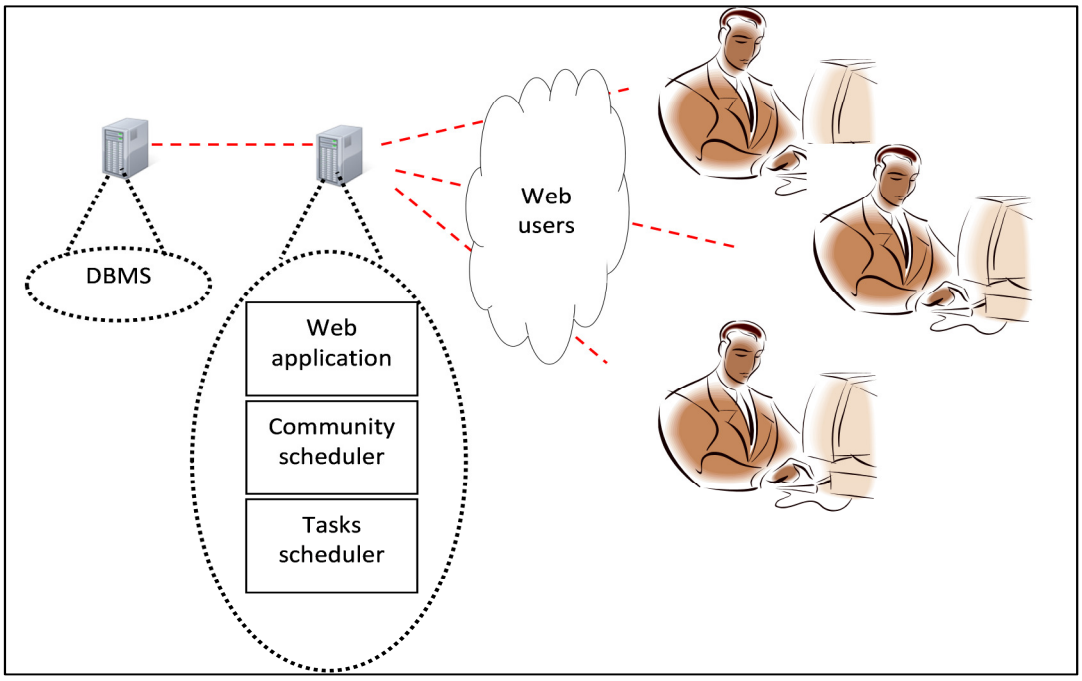

Figure 74 - Scenario III: centralized web-based OLTP application

Mix of Transactions

Our mix of transactions is inspired in TPC-W's browsing mix. We implemented four transactions: Home, Product Detail, Order Display and Admin Request (Appendix A details the SQL commands of each transaction).

The Initial Order Display transaction is much longer than the other ones and is a block of statements with sequentially requirements. Table 20 presents the percentage of each transaction in the mix and the foreseen execution cost of each transaction obtained by the transaction's foreseen execution plan provided by the DBMS. In terms of execution time in the considered environment, the execution time of the Product Detail transaction varies from just a few milliseconds (when the user has no orders) up to almost 50 seconds, while the typical execution time of the other transactions is of about just a hundred milliseconds.

Table 20 – Initial Transaction's Foreseen Execution Cost and Mix of Transactions

Transaction	Foreseen Execution Cost	Percentage of the Highest Foreseen Execution Cost (%)	Percentage of Executions in Used Mix (%)
Home	0.027	0.05	57
Product Detail	0.007	0.02	40
Order Display	31.716	100.00	2
Admin Request	0.007	0.02	1

In our experiments, we considered the Admin Request as a high priority transaction, while the other transactions are of normal priority. All the transactions have an execution deadline of 30 seconds.

Evaluated Scheduling Strategies

We used three scheduling strategies:

- Best Effort (BE) – it simulates a traditional environment, where every user query is submitted to the database and executed as soon as possible. The application uses as command execution timeout the desirable transaction execution deadline. We tested this strategy with several limits on the number of queries being concurrently executed by the DBMS (i.e. multi-programming level - MPL). In the best effort approach, the MPL limit is implemented as the maximum number of active connections in the connection pool (in Appendix A we present the used connection pool). We present here the results obtained when using the value of 600 as the maximum MPL allowed;
- Admission Control System (ADC) – We implemented a prototype of the admission control strategy proposed by Schroeder et al (2006b). Such strategy aims at maintaining a low MPL in order to achieve specified deadlines (as discussed in Chapter 2). We tested several limits to MPL and present here the results obtained when using 20 as the maximum allowed MPL;
- QoE oriented scheduling – We used our scheduling strategy. As discussed in Chapter 6, our system uses an *express* queue for small tasks and another for the other tasks. There is a limit on the number of small tasks that can be executed simultaneously (MPL limit) and another for the number of other tasks tan can be executed simultaneously. In the following, we present the results obtained when the MPL limit for small tasks is 20. The MPL limit for large tasks is automatically adjusted.

Experimental Results

Figure 75 presents the acceptance rate for each scheduling strategy and transaction submission rate. The Best Effort strategy provides an acceptance rate of 1.0 (as it executes all incoming queries). The Admission Control application, on the other hand, refused to execute a large amount of transactions, leading to low values of AR. The number of refused transactions is much greater in the ADC strategy than it is when using our QoE strategy.

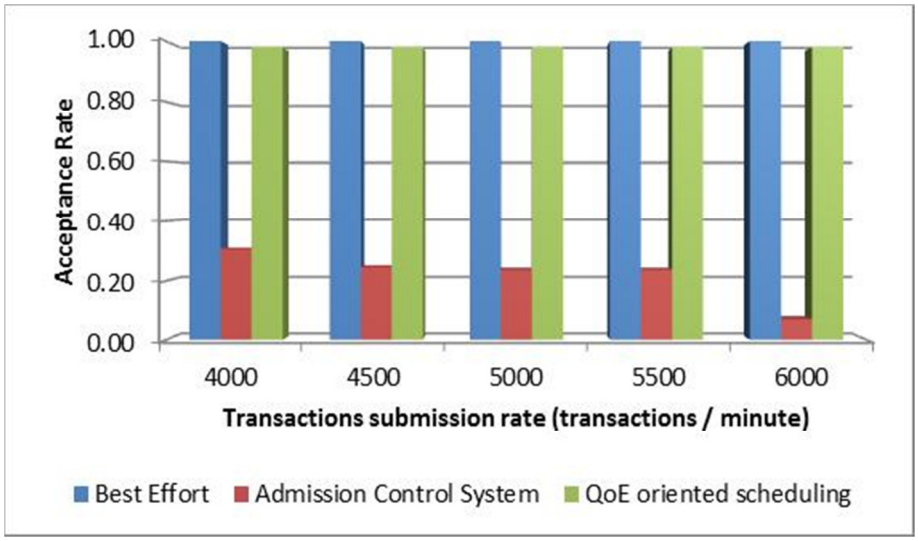

Figure 75 – Acceptance Rate for distinct Scheduling Strategies

The QoE oriented system achieved high acceptance rate for all types of transactions, except for the long running Order Detail transaction, which had an acceptance rate of about 2%. In the ADC case, the system refused to execute several transactions of all the types, as presented in Figure 76.

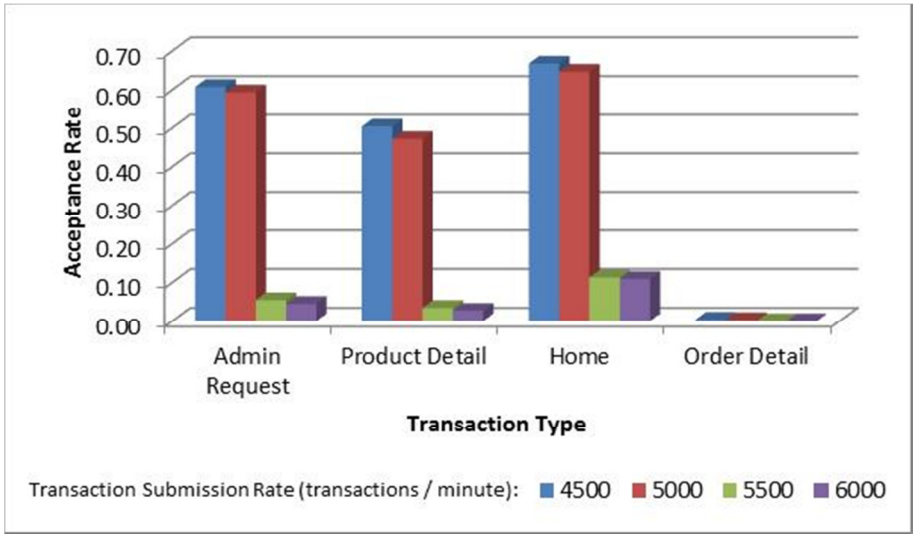

Figure 76 - Acceptance Rate for distinct types of transactions - ADC Application

In fact, the ADC strategy was somewhat conservative, and achieved a commitment maintenance rate of almost 1.0 in all tested transaction submission rates. The CMR of the Best Effort strategy is equal to its success rate (as such strategy executes all submitted transactions). The CMR of the QoE oriented strategy was close to 0.99.

The success rate obtained when using Best Effort, Admission Control System (ADC) and QoE oriented strategies is presented in Figure 77. The QoE oriented approach achieved much higher success rates than the other strategies. This happens

because BE executed every transaction and ADC refused to execute a too large number of transactions.

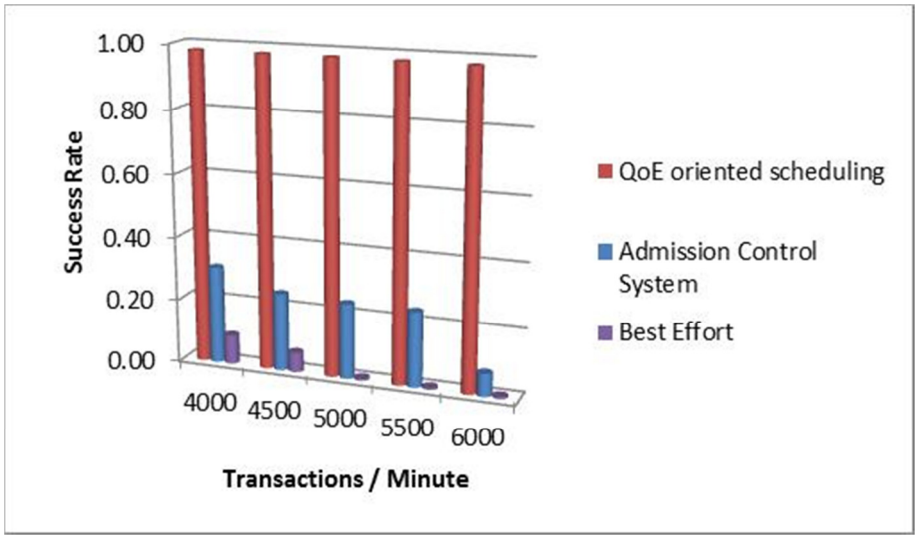

Figure 77 – Success Rate for distinct Scheduling Strategies

The distinct behavior of studied strategies also impacted in the QoEL they provide to users. Figure 78 presents the QoEL for distinct strategies and several workload submission rates. The BE strategy provided low QoEL in all configurations, as the users expect that the system would satisfy the requirements, but the number of satisfied requirements is relatively small. The ADC strategy, on the other hand, refused to execute a too high number of tasks, leading to even worse results in terms of QoEL. The QoE oriented system achieves both high number of accepted transactions and high levels of commitment maintenance rates, which led to the highest levels of QoEL.

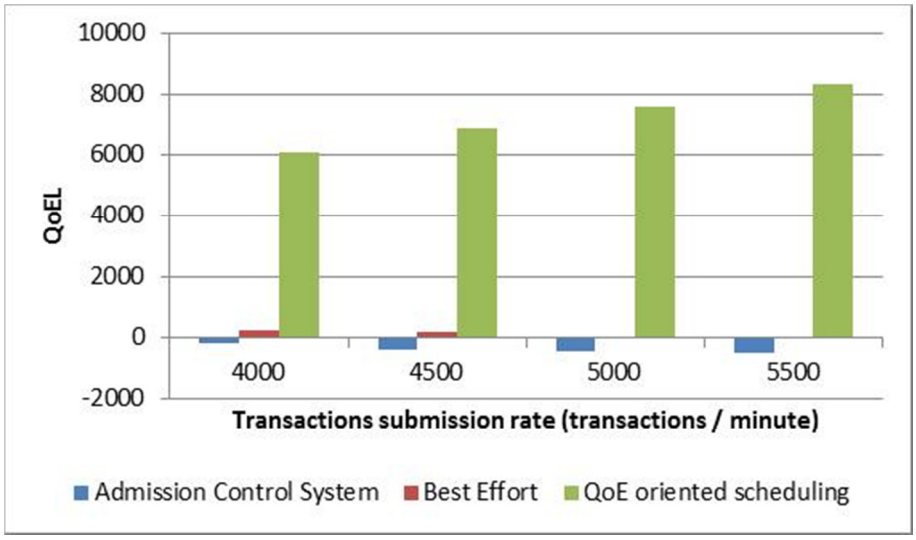

Figure 78 – QoEL for distinct Scheduling Strategies

But the QoE oriented scheduling rejected to execute a high number of Order Detail transactions. As the AR for such type of transactions was lower than 10% in the monitored period, the system administrator is informed that available resources are not capable to deal with such kind of transactions in the current configuration. Then, physical database tuning takes place (a new index is created by the administrator).

Figure 79 presents the QoEL of Order Detail transactions before and after index tuning. After tuning actions, the system is capable to deal with a much larger number of Order Detail transactions, and the QoEL that was negative turns to be positive.

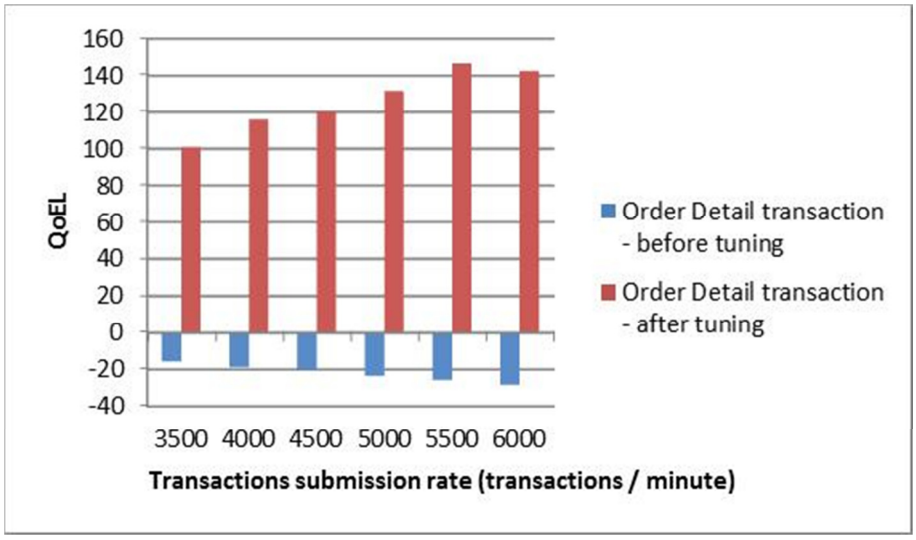

Figure 79 - QoEL for order detail transactions

In such context, we also evaluated the use of the execution priority requirement. The Admin request transaction corresponds to just 1% of transactions execution requests. We evaluated the system's behavior when such transactions of the Admin request type are of normal priority and when they are marked as having high priority. The mean execution time of high priority Admin requests is just of about 52% of the mean execution time of normal priority Admin requests.

8.4 Evaluating Specific Features

Some specific features of the QoE approach and of its application in different scenarios were also tested as parts of the experimental analysis. In this section we first present the results of experiments made to evaluate the use of reputation-based mechanisms (Section 8.4.1) – reputation is a core aspect of the QoE approach, with relevance in multi-node and multi-site environments - and then the results of tests made to evaluate queue management and time estimation capabilities (Section 8.4.2). Queue management is relevant for scheduling in data services – especially in OLTP environments – and time estimation is necessary in any environment (either OLTP or OLAP) to estimate the execution time of tasks, as part of scheduling decisions.

8.4.1 Reputation Tests

Reputation classifies the capability of an entity to commit to its promises and duties and is used in decisions on whether to use the entity to execute a duty. In this set of tests, we tested the proposed reputation approach when applied in tasks scheduling. We demonstrate that the use of the proposed mechanisms is useful even when it is not possible to accurately foresee future conditions.

We used 9 Data Services (each one with its own task scheduler) and a single Community Scheduler. In order to verify the use of proposed reputation-based mechanisms, 3 of the used data services are modified so that they do not estimate correctly the required time to execute tasks: they are transformed into *optimistic* schedulers, accepting to execute all tasks and making promises to finish task execution immediately. In order to remove from the community scheduler all mechanism that can correct wrong estimations done by tasks schedulers, we disabled the use of the *reputation on maintaining promises on execution time interval* by the community scheduler.

The workload is composed of 400 jobs, each one is created by transforming query 1 of TPC-H to access a single partition of the `LINEITEM` table. Therefore, each `LINEITEM`'s partition is accessed by 4 jobs. Jobs were submitted in a 4 jobs per second rate. Each job has an Execution Deadline requirement of 30 seconds.

Experimental Results – Without using reputation based mechanisms

First, the community scheduler is adjusted to assign tasks execution by just considering candidates' promises on the required time to execute each task. In this test, services can be elected to execute a task no matter what their reputation is.

Only the three services that do not estimate well the required execution time executed tasks, as represented in Figure 80. They promise instant execution, therefore they are always chosen. This configuration had high load misbalancing.

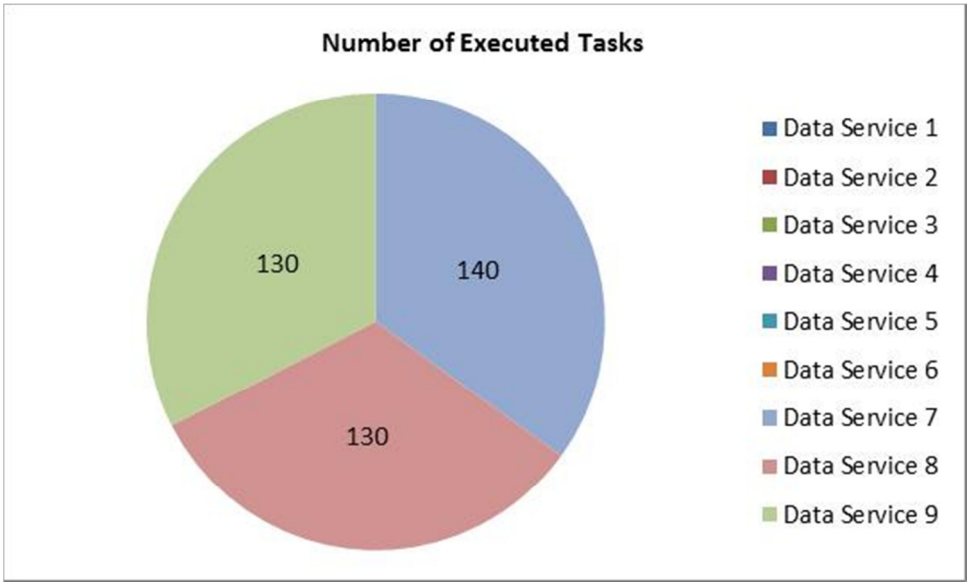

Figure 80 - Number of executed tasks per data service – without using reputation based mechanisms

The misbalancing and the wrong predictions made the system fail to satisfy the requirements of most jobs. Figure 81 presents the acceptance rate, commitment maintenance rate and success rate of the entire workload execution and of four time intervals of workload execution. All submitted jobs were accepted (AR = 1 in all the period), but the system was unable to fulfill several requirements (CMR < 0.25 during the entire period of load execution), which leaded to low levels values for the success rate indicator (final SR = 0.19). The obtained value for QoEL is 38.5.

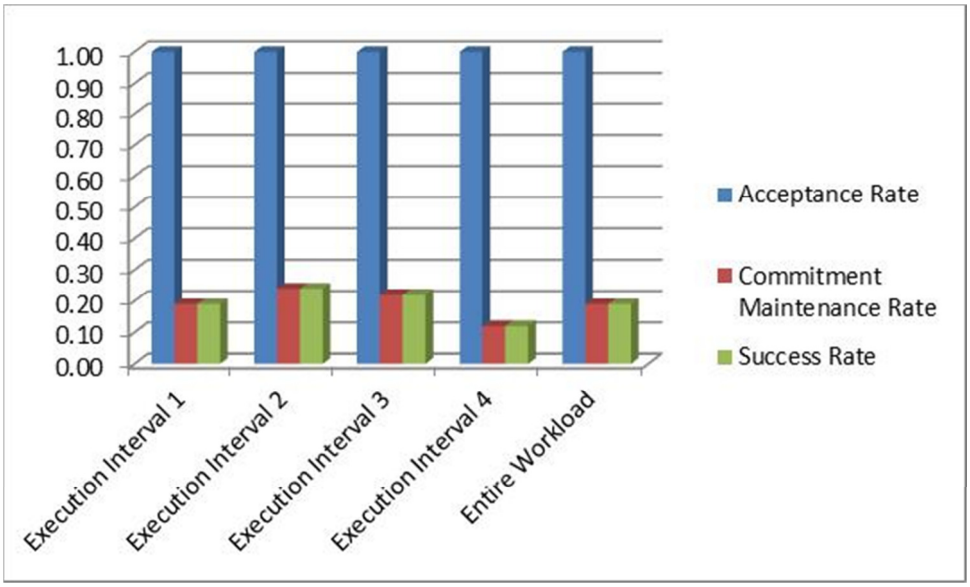

Figure 81 - AR, CMR and SR – without using reputation based mechanisms

We also measured the reputation on maintaining commitments to satisfy tasks of each data service (represented in Figure 82). The reputation value of the services that did not execute any task remained in 1.0, while the reputation value of the other (*optimistic*) services fluctuated around the value of 0.2 (near the achieved success rate) value.

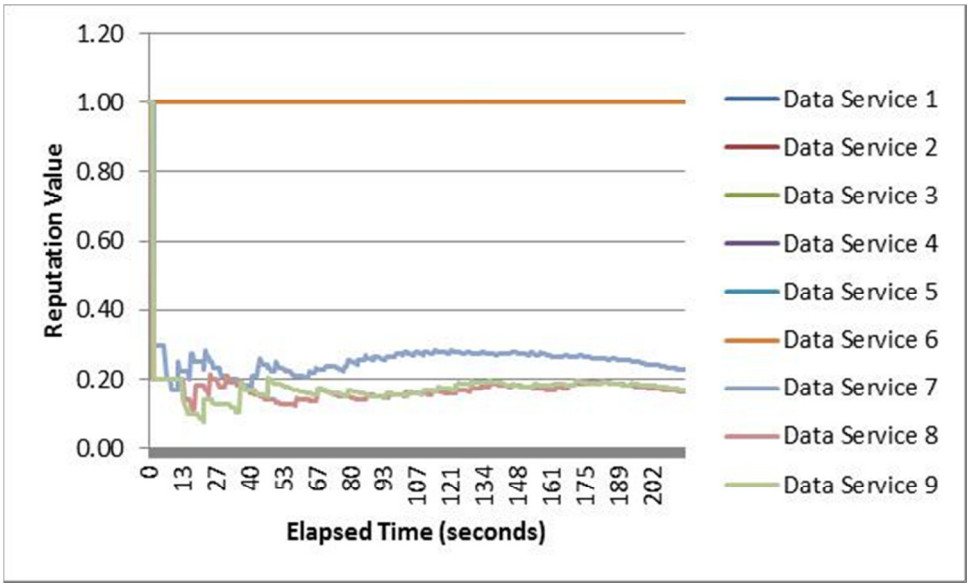

Figure 82 - Reputation on maintaining commitments to satisfy tasks – without using reputation based mechanisms

Experimental Results – Minimal reputation requirement

Then, Next we configured the community scheduler to use the minimal reputation mechanism proposed in Section 5.1: in our implementation, if a service reputation on maintaining commitments to satisfy tasks falls under 0.9, then for each 10 promises of the data service, only one is considered. Besides that, the community scheduler was adjusted to consider both reputation value and time estimations when electing a winner to execute a task: election's score (Section 5.1.3) is computed using $\upsilon = \omega = 0.5$. The results we show next prove that this configuration improved the QoE the system provides significantly: the new value is 132.3 (almost 350% the value of previously obtained QoEL).

In Figure 83 we present reputation on maintaining commitments to satisfy tasks of each data service. Optimistic services (those that accept all tasks for execution, promising to finish then immediately) had a first reputation fall to bellow the limit of 0.9. At that point the community scheduler started refusing most promises from such data services (9 in every 10 promises, as we described above). Some time later, two of those lower-rated data services were actually able to raise their reputation score from executing few easy jobs, their reputation increasing over the 0.9 threshold value. As soon as they went over the 0.9 threshold, those two services won several subsequent elections. Of course moments later the community scheduler noticed that those data services were not accomplishing the specified deadlines (i.e., they won several elections in a 30 seconds period), and their reputation had another great fall.

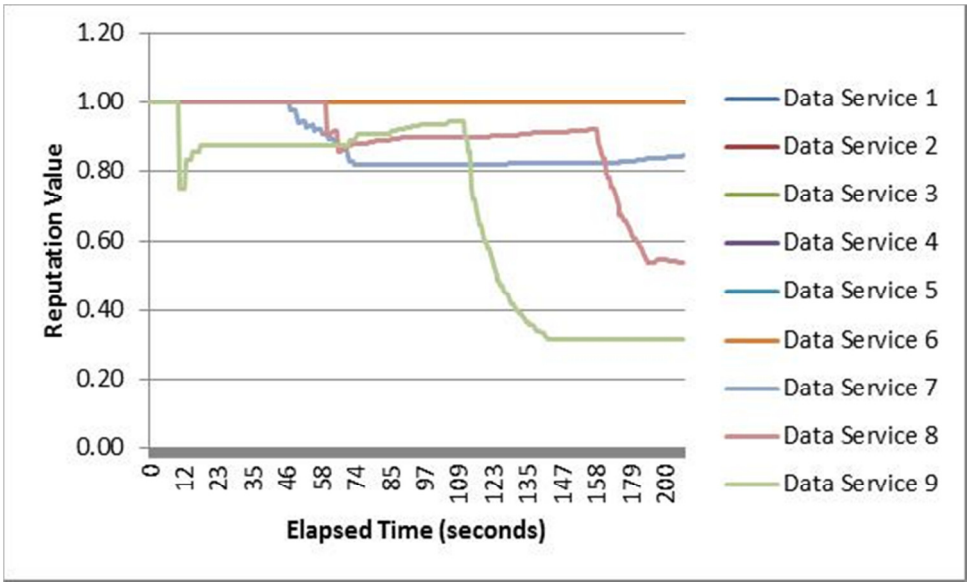

Figure 83 - Reputation on maintaining commitments to satisfy tasks – using minimal reputation requirement

Figure 84 presents the acceptance rate, commitment maintenance rate and success rate of the entire workload execution and of four time intervals of workload execution. At the beginning of workload execution, all jobs are accepted and almost all executed. But then, the optimistic nodes become overloaded and the commitment maintenance rate falls significantly (intervals 2 and 3). But at the last 20% of execution time (see time 150 of Figure 83), the reputation value of *optimistic* nodes is much lower than the 0.9

threshold value and they are allowed to participate only in 10% of elections. In such period, the system regrets to execute many jobs, but achieves a high CMR value.

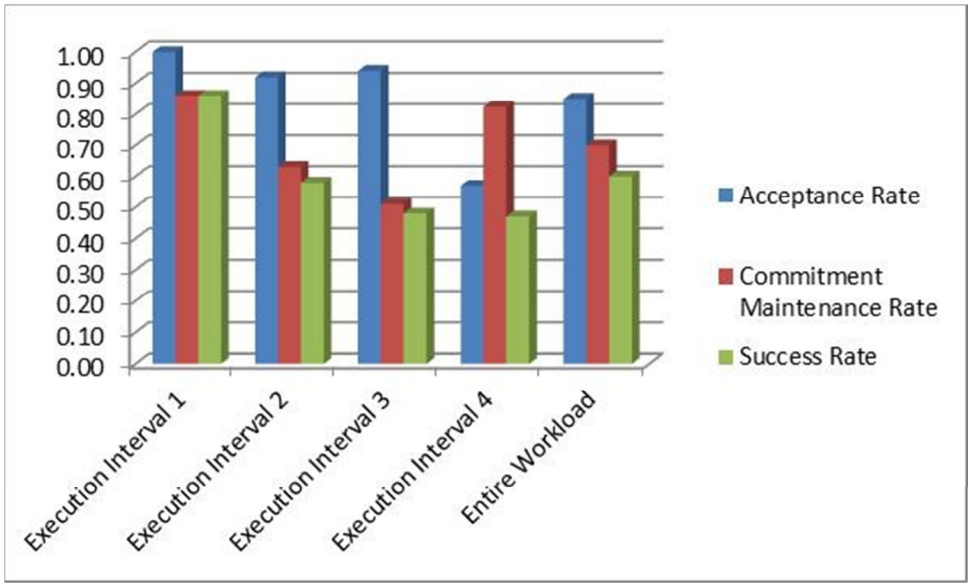

Figure 84 - AR, CMR and SR – using minimal reputation requirement

Figure 85 presents the number of tasks executed by each data service. Due to the minimal reputation control, the workload distribution across participating services was much more balanced than in previous test.

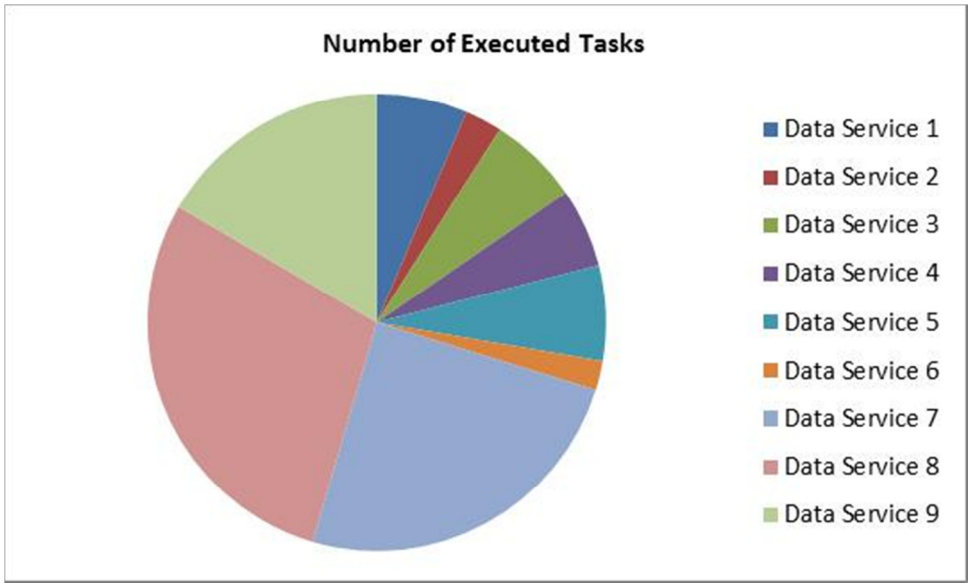

Figure 85 – Number of executed tasks – using minimal reputation requirement

Now, we analyze the behavior of the system in terms of task assignment during workload execution. Figure 86 presents the number of tasks executed per data service in distinct time intervals. Initially, all tasks are assigned to the *optimistic* services (most of them to service 7). Then, in the second time interval, reputation of service 7 goes down the acceptable limit (see time 15 of Figure 83), and it does not execute any tasks. But most of them are assigned to another optimistic service (service 9), which did not

executed many tasks in the first time interval. The *normal* services (which are not optimistic) execute some tasks. In time interval 3, the reputation of service 9 is beyond the acceptable limit (see time 100 of Figure 83), and it does not execute new tasks. The third *optimistic* service (data service 8) executes most of the tasks. Finally, in execution interval 4, reputation values are somewhat stabilized (see period after time 150 of Figure 83). Then, in such interval, tasks distribution per data service is much more balanced, as represented in Figure 86.

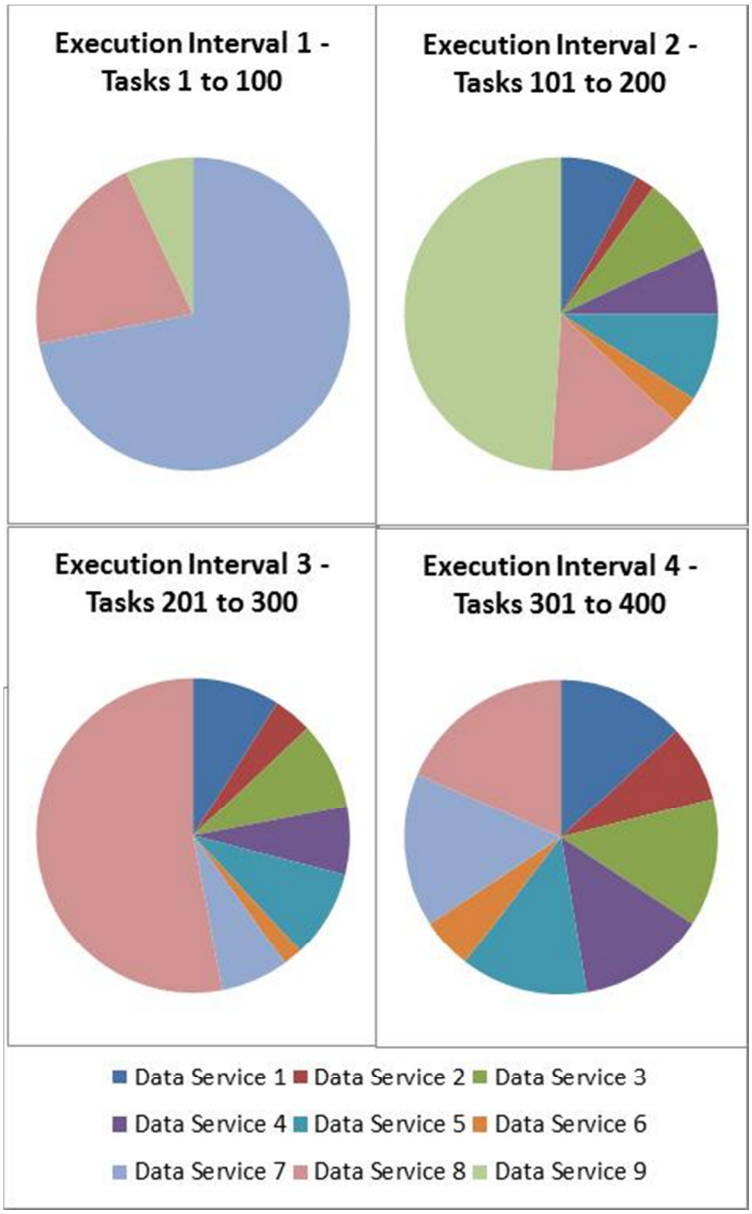

Figure 86 - Number of executed tasks per data services in distinct time intervals

In order to improve the level of QoE provided, next we configured the system to use the restriction on the number of victories in sequence (defined in Section 5.1).

Experimental Results – Minimal reputation requirement and restriction on the maximum number of victories in sequence

The use of the requirement of a minimal reputation value in order to participate in an election improved scheduling quality. But it takes some time for the system to react to changes in nodes behavior. To modify this, we also incorporated into our system a restriction on the maximum number of victories in sequence (as discussed in Section 5.1). In our test, when a data service wins three elections in a row, it cannot participate in another election for 5 seconds. The community scheduler was adjusted to elect winners based on services' reputation. This configuration increased even more the value of the provided QoEL to 138.6.

A key benefit of the restriction on the maximum number of victories in sequence is that it reduces the possibilities of a great number of commitment failures due to the change in behavior of a single service. Besides that, it also contributes to improve load balancing between services. In Figure 87, we present the number of tasks executed per data service. Due to proposed mechanisms, the number of tasks executed by optimistic nodes was equivalent to the one of some of the *normal* (non-optimistic) nodes.

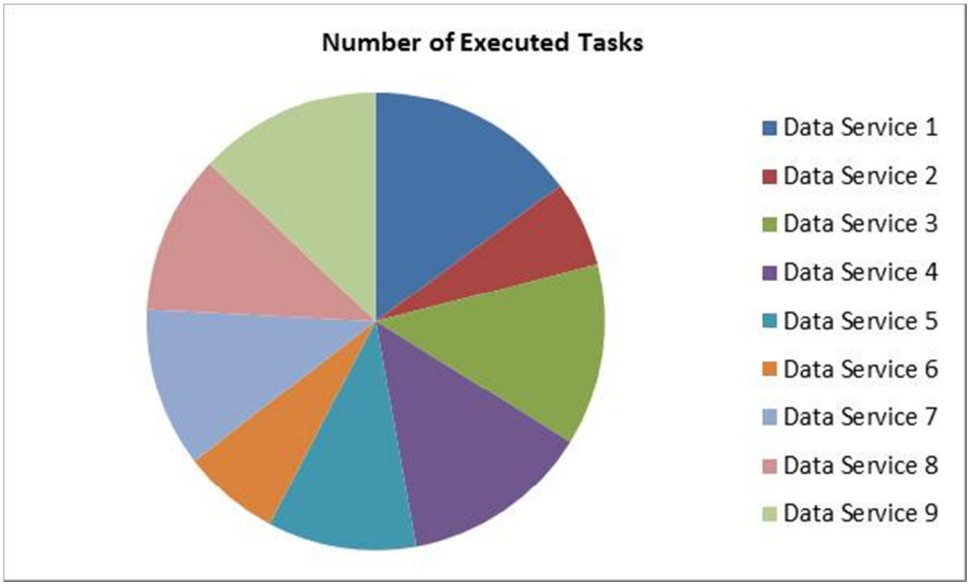

Figure 87 – Number of executed tasks – using minimal reputation requirement and restrictions on the maximum number of victories in sequence

Indeed, the system behavior became even more stable over time. Figure 88 presents the obtained values for AR, CMR and SR. AR and SR remained almost the same during the entire workload execution. Workload execution CMR was of 1.0. The system was able to satisfy the requirements of all the jobs it accepted to execute, which means that data services' reputation was of 1.0 even for the optimistic services.

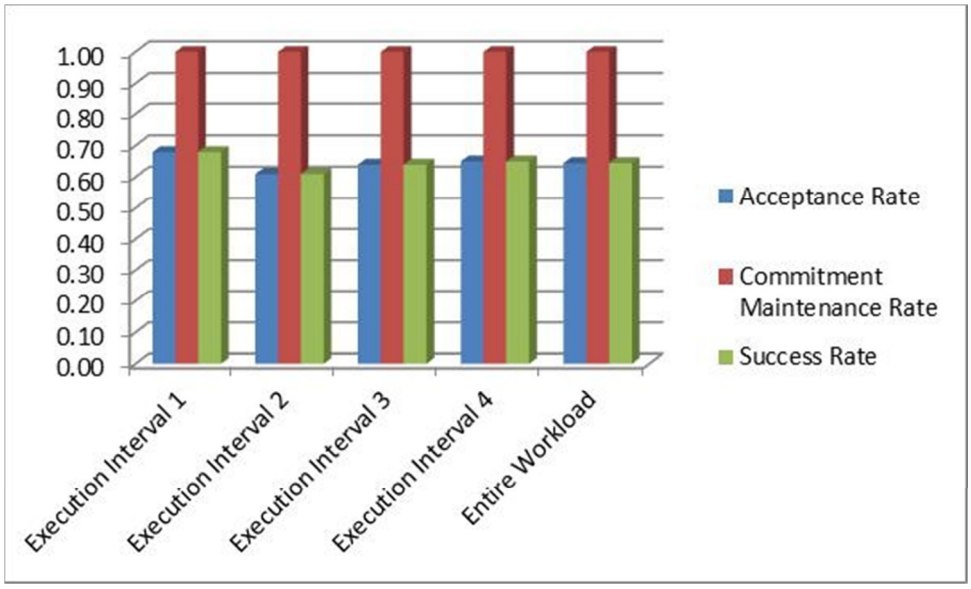

Figure 88 - AR, CMR and SR – using minimal reputation requirement and restrictions on the maximum number of victories in sequence

In this set of tests, we presented how proposed mechanisms can provide high QoE levels even though some services do not estimate well the required time to execute tasks. We presented that the minimal reputation limit increases the QoE level by reducing the impact of wrong behavior after some time. We also presented that the restriction on the maximum number of election victories in sequence prevents that some wrong estimations influence the level of QoE provided by the system.

8.4.2 Queue Management and Time Estimation Analysis

When resources are limited when compared with task needs, large tasks not only fail their own deadlines as they influence the commitment rates of other tasks as well. For this reason the tasks scheduler mechanisms proposed in Chapter 6 treat very small tasks in distinct ways than it deals with normal and long-running tasks. Very small tasks have a separate queue and their own time estimation mechanism.

Estimation time is an important aspect of scheduling in our QoE proposal. In this section we also discuss estimation time error as part of the study on alternative queue approaches.

In order to evaluate the use of specific mechanisms for very small tasks, we made a set of tests that we describe in this section. Such tests were also used to evaluate system's time estimation capabilities.

Such tests were made using the same TPC-W benchmark and workload used in the scenario of centralized database systems (Section 8.3). We experimentally evaluated three basic configurations for tasks scheduler:

- *No Small Tasks Queue* (NSTQ)- there is a single tasks queue and none of the transactions is considered a *small task* (i.e. uses the small tasks queue);
- *Small Tasks Queue-Conf.1* (STQ-Conf1) – Only the transactions that have the lowest foreseen execution costs are considered as small tasks. We have set the

number of such transactions to 2. Therefore, the smallest tasks - Product Detail and Admin Request transactions - are considered as *small tasks* and have their own *express* execution queue (and database connections);

- *Small Tasks Queue-Conf.2* (STQ-Conf2) – Transactions that have execution time smaller than a certain threshold value (we have set this execution time parameter to 1 second) are considered as small tasks. In such case, Product Detail, Admin Request and Home transactions are *small tasks* and have their own execution queue (and database connections);

Each of the above alternatives was evaluated considering several limits on the number of small tasks commands that can be concurrently executed by the DBMS (i.e. multi-programming level - MPL).

In the following, we discuss some of the obtained results for each of the above presented configurations, when the MPL limit for *small tasks* is fixed as 20.

Experimental Results – Queue Management and Time Estimation

In Figure 89 we present the acceptance rate (AR) for each of considered configuration alternatives. When considering the entire workload, the STQ-Conf2 had the highest values of AR, while the NSTQ configuration achieved the lowest values. The configurations that used small tasks queue had an almost constant value of acceptance rate, while the acceptance rate in the NSTQ configuration with high load situations was slightly smaller than the one obtained with lower loads.

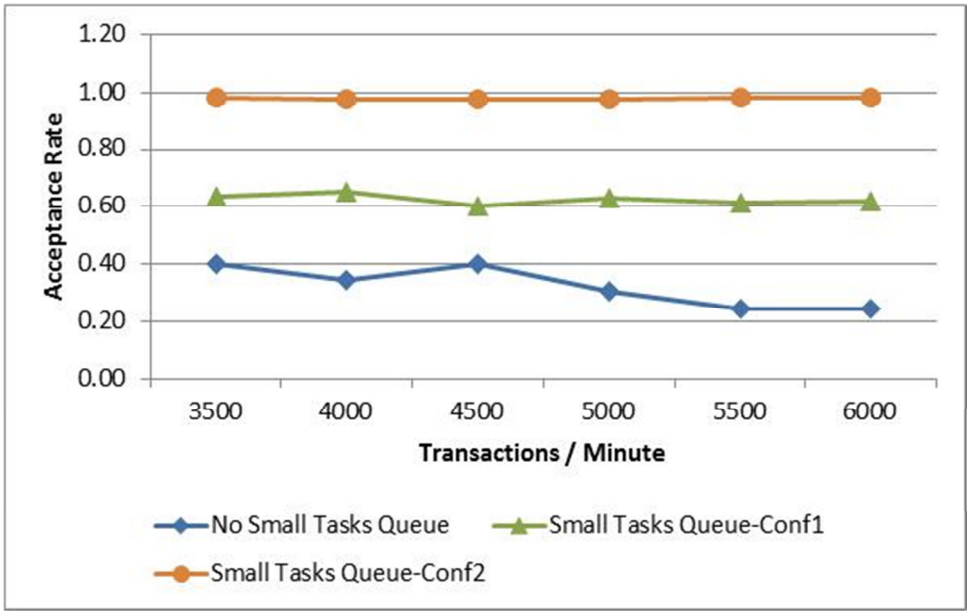

Figure 89 - Acceptance Rate - Alternatives on the use of Small Tasks Queue

The distinct behavior is closely related to the strategy used to estimate the execution of time and schedule transactions. Figure 90 presents typical execution time (measured from the moment of job submission until its execution is completed) and foreseen time errors for each type of transaction and distinct configurations (under the 5.500 transactions per minute workload submission rate).

In the NSTQ configuration, tasks with a (relative) small execution cost remain in the tasks queue waiting to be executed after the end of (relative) big tasks (derived from the Order Detail interaction). Therefore, in such configuration Admin Request, Product Detail and Home interactions take a few seconds to be executed. Besides that, the scheduler makes some mistakes when foreseeing the execution time of tasks derived from such transactions. But the foreseen execution time error for the long Order Detail job is relatively small.

In the STQ-Conf-1 configuration, tasks derived from the Admin Request, Product Detail transactions have an 'express queue' and do not compete for database connections with the long Order Detail job. In such configuration, the execution time of such jobs is significantly reduced to less than a second.

In the STQ-Conf-2 configuration, the mean execution time of the Home transaction is reduced to less than a second. Although the execution cost of the Home transaction is about 4 times greater than the ones of the Admin and Product Detail transactions, the Home transaction can also be placed in the small tasks queue together with the other inexpensive transactions. This result indicates that the selection of jobs that can use the small tasks queue can be done considering the average job execution time.

As the number of small jobs is much higher than the number of long jobs and due to the difference in the foreseen execution time obtained under the distinct configurations, the AR of the NSTQ configuration is much small than the one of the other methods.

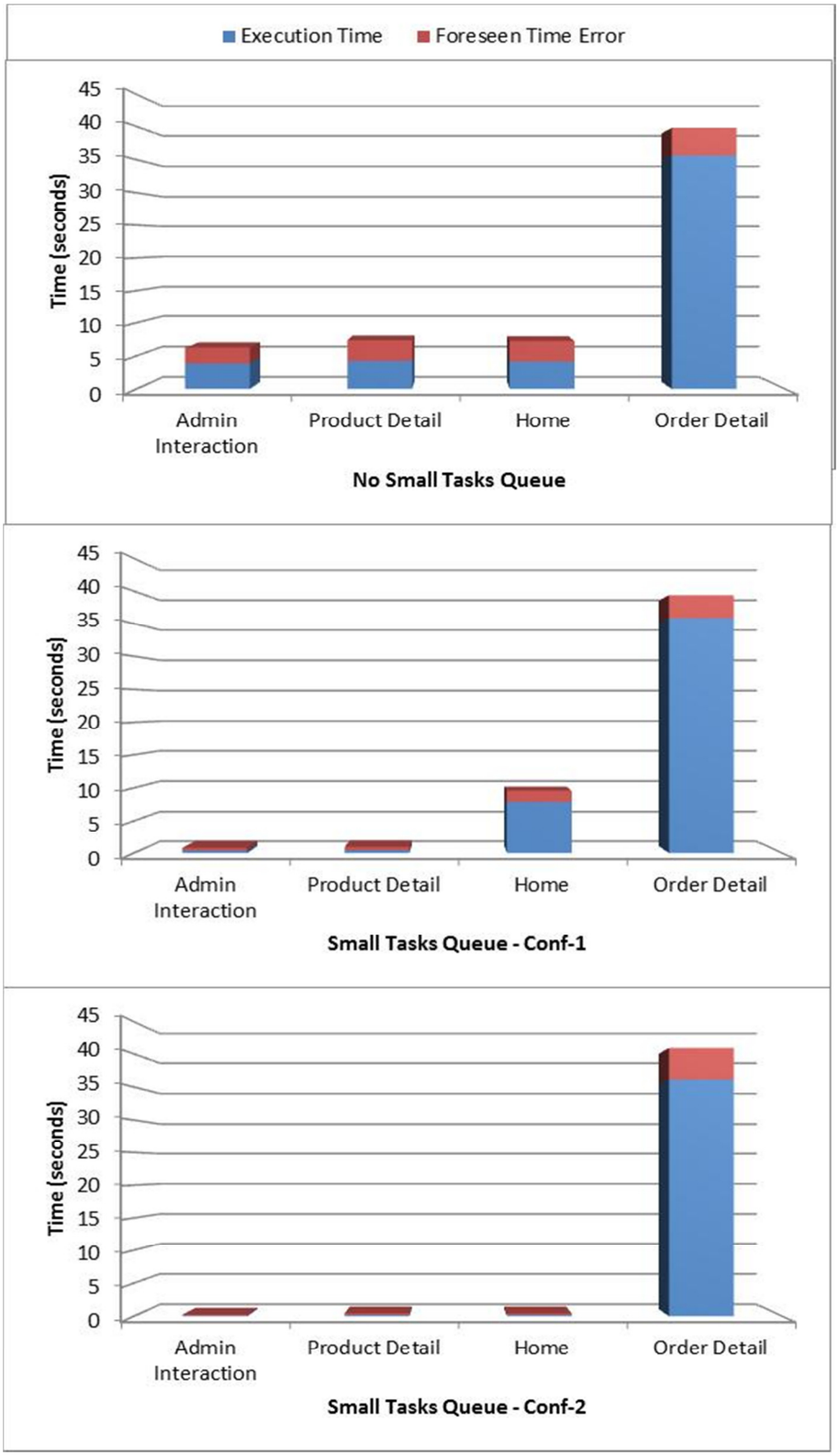

Figure 90 – Execution Time and Execution Time Forecast Error for each type of transaction - Alternatives on the use of Small Tasks Queue

Therefore, the use of the express queue made the system differentiate from short-running tasks from the other ones. This could improve the number of small tasks

executed by the system, as represented in Figure 91, while reducing the number of long-running tasks executed by just a few, as represented in Figure 92.

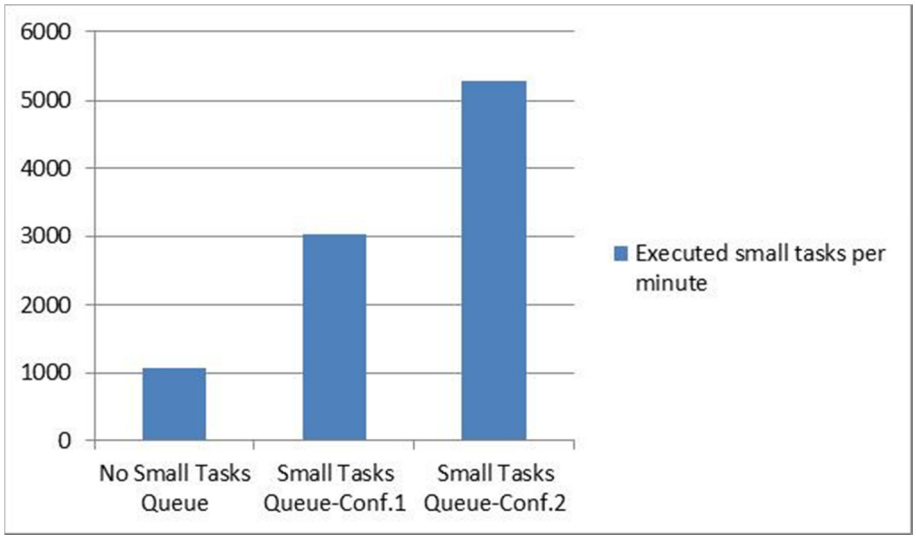

Figure 91- Number of Small Tasks transactions executed in the 5,500 tasks per minute submission rate

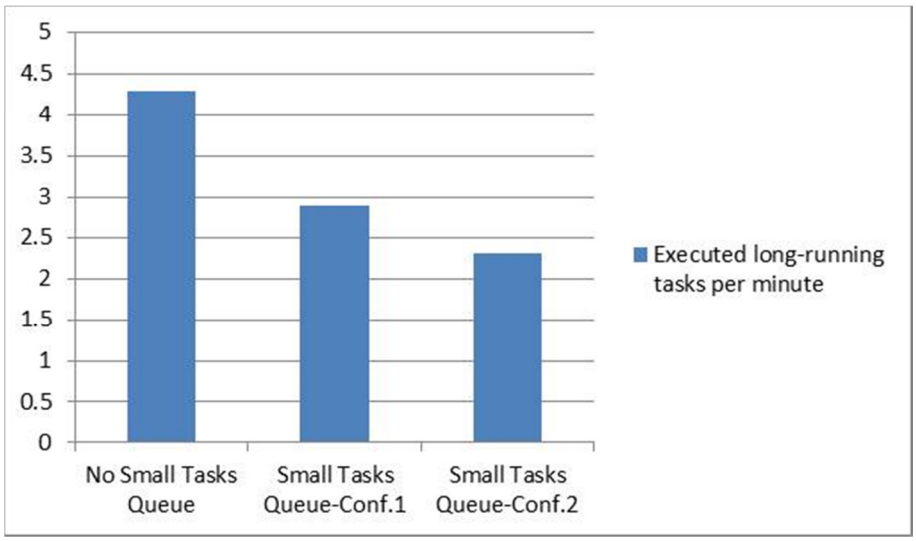

Figure 92 - Number of Order Detail transactions executed in the 5,500 tasks per minute submission rate

Such distinct number of executed transactions makes that each configuration achieves highly different values of QoEL. Figure 93 presents the value for QoEL indicator for each of the tested configurations and workload submission rates. The use of the time-related criteria when placing a task in the small tasks queue has proven to be the one that leaded to the best results in terms of level of QoE.

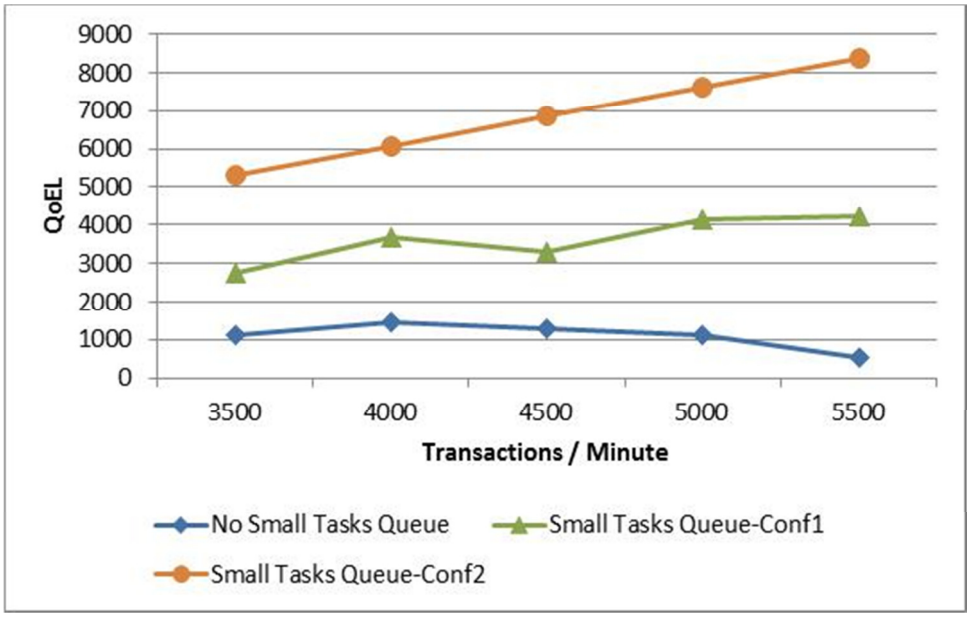

Figure 93 – QoEL - Alternatives on the use of Small Tasks Queue

In this set of tests, we presented how the use of an express queue for small tasks can improve the level of QoE the system provides. We also presented that proposed strategy leads to good query execution time estimations.

8.5 Conclusion

In this chapter, we presented the use of QoE related techniques in three distinct scenarios: (i) globally distributed data warehouse; (ii) parallel cluster-based data warehouse; and (iii) centralized OLTP application. Besides that, we also evaluated some specific features: reputation based mechanisms, queue management and runtime estimations.

In the global warehouse scenario, we discussed how DARs and dynamic replication can be used to improve the QoE level provided by the system. We also used DARs to improve data availability and enable users to execute workloads even when some sites are unavailable.

In the parallel warehouse scenario, we presented how best-effort approaches, which do not take into account user requirements, can lead to users' dissatisfaction. When using such approaches, the system fails to satisfy the requirements of several jobs. Then, DARs were used to dynamically adjust the execution time of long-running jobs, which enabled requirements fulfillment of both long-running and short-running jobs. Such scenario was also used to present how proposed strategies can be used to automatically suggest data placement over parallel databases.

In the centralized database scenario, we compared our approach with best-effort and admission control approaches, and presented how our approach can lead to the highest levels of QoE. Besides that, we presented how our proposals can be used to alert system administrators about the need of tuning actions. The use of execution priority

requirements was also evaluated and proven to be relevant in order to reduce the execution time of selected transactions.

We also made some tests to evaluate specific features proposed in the thesis. We evaluated the use of reputation-based mechanisms in the presence of nodes making wrong estimations (accepting to execute every task and informing that they would finish their execution immediately) and showed how proposed mechanisms can be used to improve scheduling decisions quality. Besides that, we also evaluated time estimations and queue management under the presence of very small and long-running transactions. We showed that the proposed multiple queue mechanism leads to good results both in terms of provided level of QoE and in runtime estimations quality.

Therefore, experimentally obtained results confirmed the usefulness and adequacy of proposed strategies.

In the following chapter, we present final conclusions and future work.

9 Conclusions and Future Works

Quality-of-Experience is a measure of a person's satisfaction while using a certain service or system. In terms of database systems, we propose that those can be translated into constraints on execution characteristics that can be expressed by users. We then propose how database systems could incorporate mechanisms specially oriented to provide high levels of Quality-of-Experience to users.

In order to support such proposal, in this work we have:

- Proposed the use of user-defined *Data Access Requirements* (DARs) in order to provide to the user a way to specify how he/she expects the system to behave;
- Presented a set of types of DARs that are useful for a wide range of applications and exemplified the use of each of proposed type of DAR;
- Presented SQL extensions that enable DARs specification in SQL;
- Proposed some key QoE-oriented components that should exist in QoE-oriented database systems and discussed how they can be used in centralized, parallel and distributed database systems;
- Described how user commands (jobs) and expectations (DARs) can be transformed into tasks (that may be performed by database systems) and task-level requirements, and presented several examples on the use of DARs and on the transformation of user-defined jobs and DARs into tasks and tasks level requirements in the contexts of centralized, parallel and distributed databases;
- Presented a reputation-aware, election-inspired task scheduling architecture that assign tasks execution to data services while maintaining high levels of requirements fulfillment;
- Proposed a strategy that detects when replica creation can improve the levels of QoE the system provides;
- Described how data services can evaluate if specified requirements are feasible or not and tasks can be scheduled for execution while satisfying task level requirements;
- Presented query execution time estimation strategies that can be used by tasks scheduler to evaluate the required time to finish a query execution while considering multi-query influence;
- Proposed a set of specialized indicators that can be used to alert administrators when the system is providing unacceptable levels of QoE and to compare the QoE levels provided by distinct database systems, presenting several examples on the analysis of such indicators;

- Experimentally evaluated several aspects of proposed strategies, presenting their effectiveness and how QoE-oriented strategies can lead to higher QoE levels than traditional approaches.

Based on these mechanisms we have been able to propose how QoE should be added to database systems. On the other hand, because of the amplitude and novelty of this issue, we expect that this work can be used as a starting point for future research in extending these basic findings.

Other domain-specific DARs can be defined and corresponding insurance mechanisms can be designed. For instance, DARs specialized for geographic and multimedia databases and DARs for security-related issues can also be studied.

Another interesting future line of work would be to explore the use of approximate query answering to satisfy queries with time constraint and data accuracy DARs.

A key aspect of the QoE-oriented database systems is prediction mechanisms that may provide estimations on several aspects, including execution time, data transfer times over networks and resources availability. In this thesis we explored times estimations, but further work would be important to improve the estimation capabilities in any of those aspects.

Overall, this thesis has shown how such QoE database systems can be implemented and it has also shown the relevance of such systems. The experiments have proven the usefulness of the proposed approaches.

Appendix A – Experimental Environment Details

In this Appendix, we detail the experimental environments used during the experiments presented in Chapter 8. We present information about used machines, databases and software. The detail presented here can be used to duplicate the experiments.

First, we present some information about the used prototype and other developed software. Then, in A.2, we detail the tests based on the TPC-H benchmark [TPCH, 2010], which includes the scenarios of global and parallel warehouses, and the reputation evaluation tests. Finally, in A.3 we detail the tests based on the TPC-W benchmark [TPCW, 2010]: the centralized database scenario, the queue management and time estimation analysis.

A.1 - QoE-oriented Prototype and other Developed Software

In order to run proposed tests, we developed a set of JAVA applications, which includes:

- Distributed community and tasks schedulers – distinct schedulers that communicate to each other using sockets and implement QoE-oriented strategies proposed in this thesis. Tasks scheduler uses the Jama matrix package [JAMA, 2010] to solve the non-linear regression used to update the *cost-to-time* conversion function (Section 6.3.2). On the start of the tasks scheduler, it executes ten database queries to obtain the initial values to use in the *cost-to-time* conversion function. These schedulers were used in reputation evaluation tests, and global and parallel scenarios;
- Global round-robin scheduler –generates tasks and assigns then to executor nodes, following a round-robin strategy. As soon as a job is submitted to the system, its tasks are generated and are assigned to executor nodes. Each executor node should manage a local tasks queue. Used in the parallel warehouse scenario;
- Global on-demand scheduler - which generates tasks and assign then to executor nodes, following an on-demand strategy. When a job is submitted to the system, its tasks are generated and placed in a global tasks queue. When an executor node is executing than a certain number of tasks (we used three tasks), the community scheduler selects a task from the global queue (in first-in-first-out order) and assigns to such node. Used in the parallel warehouse scenario;
- Local nodes queue management system – run at local nodes and execute tasks that are assigned to the node by global round-robin or on-demand schedulers. Has a local queue with the tasks that are assigned to the node and that are waiting for execution. A maximum value is specified for the MPL (we used 3 as the maximum MPL value). When there are free database connections, a task is removed from the local queue (in

first-in-first-out order) and its execution is started. Used in the parallel warehouse scenario;

- Best effort oriented web application simulator – partially simulates the TPC-W environment, implementing some of its transactions. Transactions are submitted to the system considering a fix mix but distinct submission rates. Each transaction is executed by a separate thread, which gets a database connection from a connection pool. Used in the centralized database scenario;
- Web application simulator with admission control system – partially simulates the TPC-W environment, implementing some of its transactions. Transactions are submitted to the system considering a fix mix but distinct submission rates. Implements the queue management and admission control strategy proposed by Schroeder et al (2006b). Each transaction is executed by a separate thread, which gets a database connection from a connection pool. Used in the centralized database scenario;
- Web application simulator with encapsulated community and tasks schedulers – partially simulates the TPC-W environment, implementing some of its transactions. Transactions are submitted to the system considering a fix mix but distinct submission rates. Community and tasks schedulers (which implements strategies proposed in this thesis) are encapsulated in this system. Used in the centralized database scenario, on queue analysis tests and time estimation tests.

In developed software, all DBMS accesses are done through the use of the *MiniConnectionPoolManager* [Mini, 2010].

A.2 – TPC-H based tests: Global and Parallel Warehouses, and Reputation Evaluation

In the global and parallel warehouses scenarios, and in the reputation evaluation tests, we used the same machine environment. All such test sets were constructed over the TPC-H database. In this Section, we present both the machines and database environments.

Table 24 summarizes the main characteristics of used machines. The number of machines used in each test was presented in Chapter 8.

Table 21 - Database Servers Main Characteristics

Operating System	Microsoft Windows XP Professional
Database Management System	Oracle 11g R1
Processor	Intel Pentium 4 3.00GHz
RAM Memory	2 GB

Figure 95 presents the main tables of TPC-H.

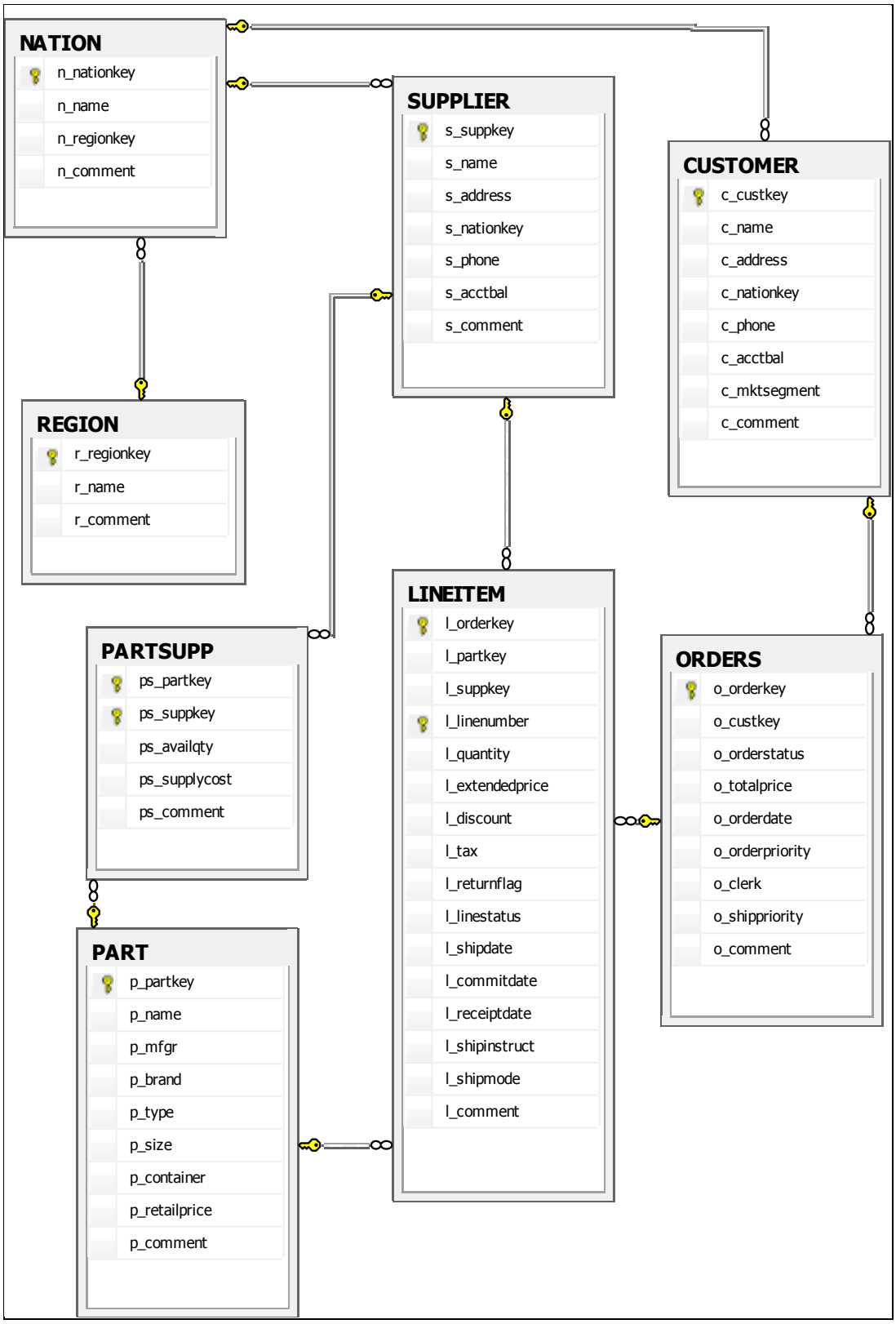

Figure 94 - TPC-H's Tables

We created a 20 GB database. Table `LINEITEM` is partitioned into 100 partitions by ranges of the `L_ORDERKEY` column. Each table has the indexes used to enforce primary key constraints and an index in each column that contains a foreign key

constraint. Table 22 presents the number of lines and storage size of each of used tables. The database is stored in Oracle 11g R1 database management system.

Table 22 - TPC-H Tables – Number of Rows and Allocated Storage Size

Table	Number of Rows	Table Size (MB)
CUSTOMER	3,000,000	462,890
LINEITEM	119,994,608	12,655,681
NATION	25	2
PART	4,000,000	453,125
PARTSUPP	16,000,0000	2,250,000
ORDERS	30,000,000	3,046,875
REGION	5	0.4
SUPPLIER	200,000	26,758

In reputation evaluation tests, all tables are replicated over used nodes. In parallel warehouses tests, tables are also replicated over used nodes (except for the autonomics tests, where tables are only replicated over the initial 3 nodes). In global warehouses tests, each region is composed by disjoint 50 partitions of the `LINEITEM` table, disjoint 8,000,000 lines of the `PARTSUPP` table and a replica of the other tables.

A.3 – TPC-W based tests: Centralized OLTP Database, Queue Management and Time Estimation Analysis

We used the same machine and database environment, both in the centralized database scenario and in the queue management and time estimation tests. In this section, we first present the main characteristics of used machines and then we describe used database environment. We also present the SQL commands used in the transactions mix.

We used the TPC-W toolset to generate about 3.7GB of data for tables `AUTHOR`, `ADDRESS`, `COUNTRY`, `CUSTOMER`, `ITEM`, `ORDER_LINE` and `ORDERS` of TPC-W. Figure 95 presents the main tables of TPC-W.

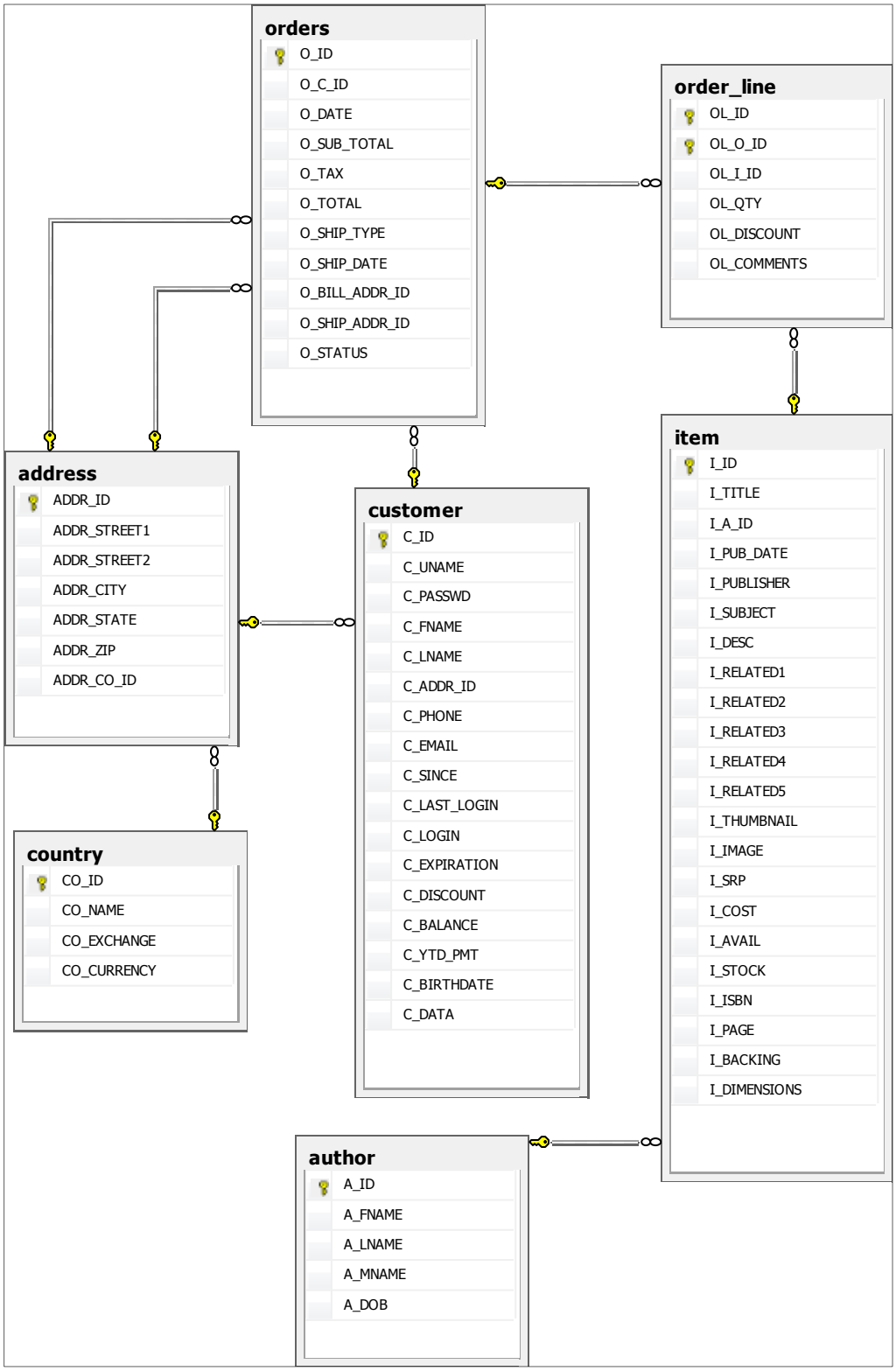

Figure 95 - Main Tables of TPC-W Database

Table 23 presents the number of lines and storage size of each of used tables. The database is stored in a single instance of SQL Server 2008 Express Edition database management system.

Table 23 - TPC-W Tables – Number of Rows and Allocated Storage Size

Table	Number of Rows	Table Size (KB)
ADDRESS	5.760.000	687.944
AUTHOR	2.500	1.040
COUNTRY	92	16
CUSTOMER	2.880.000	1.458.960
ITEM	10.000	6.024
ORDER_LINE	7.775.551	817.992
ORDERS	2.592.000	306.376

Besides the indexes created to enforce primary key constraints, we only created an index in the column `C_UNAME` of table `CUSTOMER`.

We used two machines, one as application server and another as a database server. Table 24 summarizes the main characteristics of server machines.

Table 24 - Database Server Main Characteristics

Operating System	Microsoft Windows XP Professional
Database Management System	Microsoft SQL Server 2008 Express Edition
Processor	Intel Pentium D 3.00GHz
RAM Memory	Total: 700 Gb Available to DBMS: 512 Mb (15% of database size)

Simulated interactions are based in TPC-W v1.8. Table 25 presents the simulated interactions, the SQL commands of each interaction and the mix of executions between interactions. The values for the underlined variables in Table 25 are randomly generated. Used mix is inspired in TPC-W's *Browsing Mix.*

Table 25 - TPC-W-Inspired Application – Interactions, SQL Commands and Mix of Executions

TPC-W Web Interaction	Interaction's SQL Commands	Percentage of Executions in Used Mix (%)
Home	select DISTINCT C_FNAME,C_LNAME from CUSTOMER where C_UNAME=@C_UNAME	57

TPC-W Web Interaction	Interaction's SQL Commands	Percentage of Executions in Used Mix (%)
Product Detail	select distinct * from ITEM,AUTHOR where AUTHOR.A_ID = ITEM.I_A_ID and ITEM.I_ID = @BookID	40
Order Display	select C_ID from CUSTOMER where C_UNAME=@C_UNAME and C_PASSWD=@C_PASSWD declare @O_ID numeric(10) select @O_ID = max(O_ID) from ORDERS where O_C_ID=@C_ID select C_FNAME, C_LNAME, C_EMAIL, C_PHONE, O_ID, O_DATE, O_SUB_TOTAL, O_TAX, O_TOTAL, O_SHIP_TYPE, O_SHIP_DATE, O_BILL_ADDR_ID, O_SHIP_ADDR_ID, O_STATUS, ADDR_STREET1, ADDR_STREET2, ADDR_CITY, ADDR_STATE, ADDR_ZIP, CO_NAME from CUSTOMER, ADDRESS, COUNTRY, ORDERS where O_ID=@O_ID and C_ID=@C_ID and O_BILL_ADDR_ID=ADDR_ID and ADDR_CO_ID=CO_ID select ADDR_STREET1, ADDR_STREET2, ADDR_CITY, ADDR_STATE, ADDR_ZIP, CO_NAME from ADDRESS, COUNTRY, ORDERS where ADDR_ID= O_SHIP_ADDR_ID and ADDR_CO_ID=CO_ID and O_ID=@O_ID select OL_I_ID, I_TITLE, I_PUBLISHER, I_COST, OL_QTY, OL_DISCOUNT, OL_COMMENTS from ORDER_LINE, ITEM where OL_I_ID=I_ID and OL_O_ID=@O_ID	2
Admin Request	select distinct * from ITEM,AUTHOR where AUTHOR.A_ID = ITEM.I_A_ID and ITEM.I_ID = @BookID	1

References

[Abdul-Rahman & Hailes, 2000] Abdul-Rahman, A., & Hailes, S. (2000). *Supporting Trust in Virtual Communities*. HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences.

[Akal et al, 2002] Akal, F., Böhm, K., and Schek, H. (2002). *OLAP Query Evaluation in a Database Cluster: a Performance Study on Intra-Query Parallelism*. In Proceedings of the 6th East European Conference. Lecture Notes in Computer Science, *2435*, pp. 181-184.

[Akinde & Böhlen, 2001] Akinde, M. O. and Böhlen, M. H. (2001). *Generalized MD-Joins: Evaluation and Reduction to SQL*. In Proceedings of the VLDB 2001 international Workshop on Databases in Telecommunications, Lecture Notes in Computer Science, *2209*, pp. 52-67.

[Akinde et al, 2002] Akinde, M. O., Böhlen, M. H., Johnson, T., Lakshmanan, L. V., and Srivastava, D. (2002). *Efficient OLAP Query Processing in Distributed Data Warehouses*. In Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology. Lecture Notes in Computer Science, *2287*, pp. 336-353.

[Alpdemir et al, 2004] Alpdemir, M. N., Mukherjee, A., Gounaris, A., Paton, N. W., Watson, P., Fernandes, A. A., et al. (2004). *OGSA-DQP: A Service-Based Distributed Query Processor for the Grid*. Advances in Database Technology - EDBT 2004, 9th International, pp. 858-861.

[Antunes & Furtado, 2007] Antunes, R. & Furtado, P. (2007). *Hardware Capacity Evaluation in Shared-Nothing Data Warehouses*. Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS 2007), pp.1-6.

[Buyya, Abramson & Giddy, 2000] Buyya, R., Abramson, D., & Giddy, J. (2000). *Nimrod/g: An architecture of a resource management and scheduling system in a global computational grid*. Proceedings Fourth International Conference on High Performance Computing in the Asia-Pacific Region, pp. 283-289.

[Cao et al, 2003] Cao, J., Spooner, D. P., Jarvis, S. A., Saini, S., and Nudd, G. R. (2003). *Agent-Based Grid Load Balancing Using Performance-Driven Task Scheduling*. In Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS), pp.49.

[Cao et al, 2005] Cao, J., Spooner, D. P., Jarvis, S. A., and Nudd, G. R. (2005). *Grid load balancing using intelligent agents*. Future Gener. Comput. Syst. 21, 1, pp. 135-149

[Chervenak et al, 2004] Chervenak, A. L., Palavalli, N., Bharathi, S., Kesselman, C., & Schwartzkopf, R. (2004). *Performance and Scalability of a Replica Location Service*. In Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing, pp. 182-191.

[Dang, Hwang & Lim, 2007] Dang, N. N., Hwang, S., & Lim, S. B. (2007). *Improvement of Data Grid's Performance by Combining Job Scheduling with Dynamic Replication Strategy*. Sixth International Conference on Grid and Cooperative Computing (GCC 2007), pp. 513-520.

[Chaudhuri, Kaushik & Ramamurthy, 2005] Chaudhuri, S., Kaushik, R., & Ramamurthy, R. (2005) When can we trust progress estimators for SQL queries? In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 575-586.

[Chaudhuri, Narasayya & Ramamurthy, 2004] Chaudhuri, S., Narasayya, V., & Ramamurthy. (2004). *Estimating progress of execution for SQL queries*. In Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 803-814.

[Costa & Furtado, 2009] Costa, R. L. C., & Furtado, P. (2009). *Deploying Data Warehouses in Grids with Efficiency and Availability*. In: Complex Data Warehousing and Knowledge Discovery for Advanced Retrieval Development: Innovative Methods and Applications, pp. 208-229. Idea Group Inc (IGI).

[Dehne & Lawrence, 2007] Dehne, F., & Lawrence, M. (2007). *Cooperative Caching for Grid Based DataWarehouses*. Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07), pp. 31-38.

[Elnikety et al, 2004] Elnikety, S., Nahum, E., Tracey, J. & Zwaenepoel, W. (2004). A method for transparent admission control and request scheduling in e-commerce web sites. In Proceedings of the 13th International Conference on World Wide Web (WWW '04), pp. 276-286.

[Foster & Kesselman, 1997] Foster, I., & Kesselman, C. (1997). *Globus: A Metacomputing Infrastructure Toolkit*. The International Journal of Supercomputer Applications and High Performance Computing, 11(2), pp. 115-128.

[Foster, Kesselman & Tuecke, 2001] Foster, I., Kesselman, C., & Tuecke, S. (2001). *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*. International Journal of Supercomputer Applications, 15 (3).

[Foster et al, 2002] Foster, I., Kesselman, C., Nick, J., & Tuecke, S. (2002). *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. Open Grid Service Infrastructure WG, Global Grid Forum.

[Frey et al, 2002] Frey, J., Tannenbaum, T., Livny, M., Foster, I., & Tuecke, S. (2002). *Condor-G: A Computation Management Agent for Multi-Institutional Grids*. Cluster Computing, 5(3), pp. 237-246.

[Furtado, 2004] Furtado, P. (2004). *Workload-Based Placement and Join Processing in Node-Partitioned Data Warehouses*. In Proceedings of the 6th International Conference on Data Warehousing and Knowledge Discovery. Lecture Notes in Computer Science, *3181*, pp. 38-47.

[Furtado, 2004b] Furtado, P. (2004). *Experimental evidence on partitioning in parallel data warehouses*. In Proceedings of the 7th ACM international Workshop on Data Warehousing and OLAP. pp. 23-30.

[Furtado, 2005] Furtado, P. (2005). *Efficiently Processing Query-Intensive Databases over a Non-dedicated Local Network*. Abstracts Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05).

[Furtado, 2005b] Furtado, P. (2005). *Hierarchical Aggregation in Networked Data Management*. Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, pp. 360-369.

[Grimshaw et al, 1997] Grimshaw, A. S., & The Legion Team, C. (1997). *The Legion vision of a worldwide virtual computer*. Communications of the ACM, 40(1), pp. 39-45.

[Gupta, Mehta & Dayal, 2008] Gupta, C., Mehta, A., & Dayal, U.. (2008). *PQR: Predicting Query Execution Times for Autonomous Workload Management*. International Conference on Autonomic Computing, 2008. (ICAC '08), pp. 13-22.

[Haas et al, 1997] Haas, L. M., Kossmann, D., Wimmers, E. L., & Yang, J. (1997). *Optimizing Queries Across Diverse Data Sources*. VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases, pp. 276-285.

[Haddad & Slimani, 2007] Haddad, C., & Slimani, Y. (2007). *Economic Model for Replicated Database Placement in Grid*. venth IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07), pp. 283-292.

[Heiss & Wagner, 1991] Heiss, H.-U., & Wagner, R. (1991). *Adaptive Load Control in Transaction Processing Systems*. VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases, pp. 47-54.

[Huynh, Jennings & Shadbolt, 2006] Huynh, T. D., Jennings, N. R., & Shadbolt, N. R. (2006). *An integrated trust and reputation model for open multi-agent systems*. Autonomous Agents and Multi-Agent Systems, 13(2), pp. 119-154.

[ITU, 1996] International Telecommunication Union. (1996). ITU-T p.800 *Methods for subjective determination of transmission quality* - series p: Telephone transmission quality; methods for objective and subjective assessment of quality.

[ITU, 2007] International Telecommunication Union. (2007). P.10/G.100 (2006) Amendment 1 (01/07): New Appendix I - Definition of Quality of Experience (QoE).

[JAMA, 2010] JAMA: Java Matrix Package, http://math.nist.gov/javanumerics/jama/. Access on December 20, 2010.

[Kamvar, Schlosser & Garcia-Molina, 2003] Kamvar, S. D., Schlosser, M. T. & Garcia-Molina, H. (2003). *The Eigentrust algorithm for reputation management in P2P networks*. WWW '03: Proceedings of the 12th International Conference on World Wide Web, pp. 640-651.

[Kilkki, 2008] Kilkki, K. (2008). *Quality of Experience in Communications Ecosystem*. Journal of Universal Computer Science, 14(5), 615-624.

[Kim et al, 2008] Kim, H. J., Lee, H. D., Lee, J. M., Lee, K. H., Lyu, W., & Choi, S. G. (2008). *The QoE Evaluation Method through the QoS-QoE Correlation Model.*

Networked Computing and Advanced Information Management. Fourth International Conference on Networked Computing and Advanced Information Management, vol. 2, pp. 719-725.

[Koenig & Kale, 2007] Koenig, G., and Kale, L. (2007). *Optimizing Distributed Application Performance Using Dynamic Grid Topology-Aware Load Balancing*. In Proceedings of 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007), pp.1-10.

[Krauter, Buyya & Maheswaran, 2002] Krauter, K., Buyya, R., & Maheswaran, M. (2002). *A taxonomy and survey of grid resource management systems for distributed computing*. Software Practice and Experience, 32, pp. 135-164.

[Lawrence & Rau-Chaplin, 2006] Lawrence, M., & Rau-Chaplin, A. (2006). *The OLAP-Enabled Grid: Model and Query Processing Algorithms*. 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment (HPCS'06), p. 4.

[Li et al, 2005a] Li, W.-S., Batra, V. S., Raman, V., Han, W., & Narang, I. (2005). *QoS-based data access and placement for federated systems*. VLDB '05: Proceedings of the 31st international conference on Very large data bases, pp. 1358-1362.

[Li et al, 2005b] Li, W.-S., Batra, V. S., Raman, V., Han, W., Candan, K. S., & Narang, I. (2005). *Load and Network Aware Query Routing for Information Integration*. ICDE '05: Proceedings of the 21st International Conference on Data Engineering, pp. 927-938.

[Lin, Liu & Wu, 2006] Lin, Y.-F., Liu, P., & Wu, J.-J. (2006). *Optimal placement of replicas in data grid environments with locality assurance*. Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on, 1, p. 8.

[Liu & Wu, 2006] Liu, P., & Wu, J.-J. (2006). *Optimal Replica Placement Strategy for Hierarchical Data Grid Systems*. Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06), pp. 417-420.

[Loukopoulos & Ahmad, 2000] Loukopoulos, T. and Ahmad, I. (2000). *Static and Adaptive Data Replication Algorithms for Fast Information Access in Large Distributed Systems*. In Proceedings of the the 20th international Conference on Distributed Computing Systems (ICDCS 2000), pp. 385.

[Luo et al, 2005] Luo, G., Naughton, J. F., Ellmann, C. J., & Watzke, M. W. (2005). Increasing the Accuracy and Coverage of SQL Progress Indicators. ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05), pp.853-864.

[Luo, Naughton & Yu, 2006] Luo, G., Naughton, J. F., & Yu, P. S. (2006). *Multi-query SQL Progress Indicators*. EDBT 2006, Lecture Notes in Computer Science, 3896, pp. 921-941.

[Marez & Moor, 2007] Marez, L., & Moor, K. (2007). *The Challenge of User-And QoE-Centric Research and Product Development in Today's ICT-Environment*. Observatorio (OBS*), 1(3).

[Martinez-Yelmo, Seoane & Guerrero, 2010] Martinez-Yelmo, I., Seoane, I., & Guerrero, C. (2010). *Fair Quality of Experience (QoE) Measurements Related*

with Networking Technologies. In 8th International Conference on Wired/Wireless Internet Communications, (WWIC 2010), 6074, pp. 228-239.

[Mini, 2010] MiniConnectionPoolManager - A lightweight standalone JDBC connection pool manager, http://www.source-code.biz/snippets/java/8.htm. Access on December 20, 2010.

[Moller, Engelbrecht & Kuhnel, 2009] Moller, S., Engelbrecht, K.-P., & Kuhnel, C. W. (2009). *A taxonomy of quality of service and Quality of Experience of multimodal human-machine interaction*. International Workshop on Quality of Multimedia Experience (QoMEx 2009), pp. 7-12.

[Natrajan, Humphrey, & Grimshaw, 2004] Natrajan, A., Humphrey, M. A., & Grimshaw, A. (2004). *Grid resource management in legion*. In Grid Resource Management: State of the Art and Future Trends, pp. 145-160. Kluwer Academic Publishers.

[Nieto-Santisteban et al, 2005] Nieto-Santisteban, M. A., Gray, J., Szalay, A., Annis, J., Thakar, A., & O'Mullane, W. (2005). *When Database Systems Meet the Grid*. CIDR, pp. 154-161.

[Nokia, 2004] Nokia Corporation. (2004). *Quality of Experience (QoE) of mobile services: Can it be measured and improved?* Telecom Services White Papers.

[Nudd et al, 2000] Nudd, G. R., Kerbyson, D. J., Papaefstathiou, E., Perry, S. C., Harper, J. S., and Wilcox, D. V. (2000). *Pace--A Toolset for the Performance Prediction of Parallel and Distributed Systems*. International Journal of High Performance Computing and Applications 14 (3). pp. 228-251.

[Oracle, 2010] Oracle Database 11g Enterprise Edition, http://www.oracle.com/us/products/database/enterprise-edition/index.html. Access on December 20, 2010.

[Ozsoyoglu & Snodgrass, 1995] Ozsoyoglu, G., & Snodgrass, R. T. (1995). *Temporal and Real-Time Databases: A Survey*. IEEE Transactions on Knowledge and Data Engineering, 7(4), pp. 513-532.

[Park & Kim, 2003] Park, S.-M., & Kim, J.-H. (2003). *Chameleon: A Resource Scheduler in A Data Grid Environment*. CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid, pp. 258-265.

[Rahman, Hassan, & Buyya, 2010] Rahman, M., Hassan, R., & Buyya, R. (2010) *Jaccard Index based availability prediction in enterprise grids*. Procedia Computer Science, *1* (1), pp. 2707-2716,

[Ramamritham, 1993] Ramamritham, K. (1993). *Real-time databases*. Distributed and Parallel Databases, 1(2), pp.199-226.

[Ranganathan & Foster, 2001] Ranganathan, K. & Foster, I. T. (2001). *Identifying Dynamic Replication Strategies for a High-Performance Data Grid*. In Proceedings of the Second international Workshop on Grid Computing. Lecture Notes in Computer Science, 2242, pp. 75-86.

[Ranganathan & Foster, 2004] Ranganathan, K., & Foster, I. (2004). *Computation scheduling and data replication algorithms for data Grids*. In: Grid resource management: state of the art and future trends. pp. 359-373. Kluwer Academic Publishers.

[Röhm et al, 2000] Röhm, U., Böhm, K., and Schek, H. (2000). *OLAP Query Routing and Physical Design in a Database Cluster*. In Proceedings of the 7th international Conference on Extending Database Technology: Advances in Database Technology. Lecture Notes in Computer Science, *1777*, pp. 254-268.

[Rood & Lewis, 2008] Rood, B. & Lewis, M. (2008). *Resource Availability Prediction for Improved Grid Scheduling*. In Proceedings of the 2008 Fourth IEEE International Conference on eScience (ESCIENCE '08). pp. 711-718.

[Sathya, Kuppuswami & Ragupathi, 2006] Sathya, S. S., Kuppuswami, S., & Ragupathi, R. (2006). *Replication Strategies for Data Grids*. In International Conference on Advanced Computing and Communications (ADCOM 2006). pp. 123-128.

[Sanchez-Macian et al, 2006] Sanchez-Macian, A., López, D., Vergara, J. L., & Pastor, E. (2006). *A Framework for the Automatic Calculation of Quality of Experience in Telematic Services*. In Proceedings of the 13th HP-OVUA Workshop.

[Schroeder et al, 2006a] Schroeder, B., Harchol-Balter, M., Iyengar, A., Nahum, E. & Wierman, A. (2006). *How to Determine a Good Multi-Programming Level for External Scheduling*. In Proceedings of the 22nd International Conference on Data Engineering. pp.60.

[Schroeder et al, 2006b] Schroeder, B., Harchol-Balter, M., Iyengar, A. & Nahum, E. (2006). *Achieving Class-Based QoS for Transactional Workloads*. In Proceedings of the 22nd International Conference on Data Engineering. pp. 153.

[Sidell et al, 1996] Sidell, J., Aoki, P. M., Sah, A., Staelin, C., Stonebraker, M., & Yu, A. (1996). *Data Replication in Mariposa*. ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering, pp. 485-494.

[Silaghi, Arenas & Silva, 2007] Silaghi, G.C., Arenas, A.E., & Silva, L.M. (2007). A Utility-Based Reputation Model for Service-Oriented Computing. Proceedings of the CoreGRID Symposium, pp. 63-72.

[Singh & Liu, 2003] Singh, A. & Liu, L. (2003) *TrustMe: Anonymous Management of Trust Relationships in Decentralized P2P Systems*. Peer-to-Peer Computing, pp. 142-149.

[Smith et al, 2002] Smith, J., Gounaris, A., Watson, P., Paton, N., Fernandes, A. A., & Sakellariou, R. (2002). *Distributed Query Processing on the Grid*. In Third International Workshop on Grid Computing - GRID 2002, pp. 279-290.

[Sonnek et al, 2006] Sonnek, J., Nathan, M., Chandra, A., & Weissman, J. (2006). *Reputation-Based Scheduling on Unreliable Distributed Infrastructures*. Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, pp. 30.

[Spiliopoulou, Hatzopoulos & Costas, 1996] Spiliopoulou, M., Hatzopoulos, M. & Vassilakis, Costas. (1996). A Cost Model for the Estimation Query Execution Time in a Parallel Environment Supporting Pipeline. Computers and Artificial Intelligence, 15 (4).

[SQL Server, 2010] SQL Server 2008: Overview, http://www.microsoft.com/sqlserver/2008/en/us/overview.aspx. Access on December 20, 2010.

[Stankovic, Son & Hansson, 1999] Stankovic, J. A., Son, S. H., & Hansson, J. (1999). *Misconceptions About Real-Time Databases*. IEEE Computer, 32(6), pp. 29-36.

[Stöhr et al, 2000] Stöhr, T., Märtens, H., and Rahm, E. (2000). *Multi-Dimensional Database Allocation for Parallel Data Warehouses*. In Proceedings of the 26th international Conference on Very Large Data Bases, pp. 273-284.

[Stonebraker et al, 1996] Stonebraker, M., Aoki, P. M., Litwin, W., Pfeffer, A., Sah, A., Sidell, J., et al. (1996). *Mariposa: a wide-area distributed database system*. The VLDB Journal — The International Journal on Very Large Data Bases, 5(1), pp. 48-63.

[TPCH, 2010] TPC-H – Homepage, http://www.tpc.org/tpch/. Access on December 20, 2010.

[TPCW, 2010] TPC-W – Homepage, http://www.tpc.org/tpcw/. Access on December 20, 2010.

[Venugopal, Buyya & Ramamohanarao, 2006] Venugopal, S., Buyya, R., & Ramamohanarao, K. (2006). *A taxonomy of Data Grids for distributed data sharing, management, and processing*. ACM Computing Surveys, 38(1).

[Watson, 2001] Watson, P. (2001). *Databases and the grid*. UK e-Science Technical Report Series.

[Wehrle, Miquel & Tchounikine, 2007] Wehrle, P., Miquel, M., & Tchounikine, A. (2007). *A Grid Services-Oriented Architecture for Efficient Operation of Distributed Data Warehouses on Globus*. 21st International Conference on Advanced Networking and Applications (AINA '07), pp. 994-999.

[Wolfson & Jajodia, 1992] Wolfson, O., & Jajodia, S. (1992). *Distributed algorithms for dynamic replication of data*. PODS '92: Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 149-163.

[Zapater & Bressan, 2007] Zapater, M. N., & Bressan, G. (2007). *A Proposed Approach for Quality of Experience Assurance of IPTV*. ICDS '07: Proceedings of the First International Conference on the Digital Society, pp. 25.