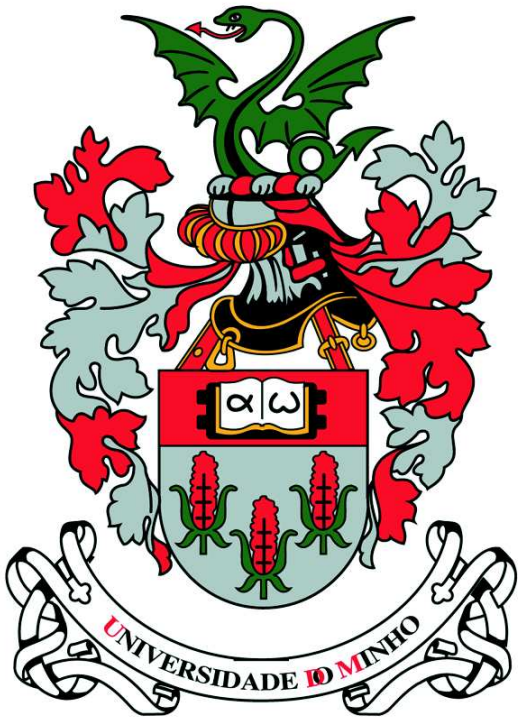
Distributed Transaction Processing in the **Escada Protocol**

Alfrânio Tavares Correia Júnior



Dissertação submetida à Universidade do Minho para obtenção do grau de Mestre em Informática, elaborada sob a orientação de Rui Carlos Mendes Oliveira

Departamento de Informática
Escola de Engenharia
Universidade do Minho
Braga, 2004

Resumo

Replicação é uma técnica essencial para a implementação de bases de dados tolerantes a faltas, sendo também frequentemente utilizada para melhorar o seu desempenho. Infelizmente, quando critérios de consistência forte e a capacidade de actualização a partir de qualquer réplica são consideradas, os protocolos de replicação actualmente disponíveis nos gestores de bases de dados comerciais não apresentam um bom desempenho. O problema está relacionado ao custo produzido pelas interacções entre as réplicas no intuito de garantir a consistência, e pelos protocolos de terminação que procuram assegurar que todas as réplicas concordam com o resultado da transacção. De uma maneira geral, o número de "aborts", "deadlocks" e mensagens trocadas cresce de maneira drástica, ao aumentar o número de réplicas. Em outros trabalhos, foi provado que a replicação de base de dados num cenário desses é impraticável.

No intuito de resolver esses problemas, diversos estudos tem sido desenvolvidos. Inicialmente, a maioria deles deixou de lado os requisitos de consistência forte ou a capacidade de actualização a partir de qualquer réplica para conseguir soluções viáveis. Recentemente, protocolos de replicação baseados em comunicação em grupo foram propostos, nos quais os requisitos de consistência forte e actualização a partir de qualquer réplica são preservados e os problemas contornados. Neste contexto encontra-se o projecto **Escada**. Sucintamente, ele tem como objectivo estudar, projectar e implementar mecanismos de replicação transaccionais adequados para sistemas distribuídos de larga escala. Em particular, o projecto explora as técnicas de replicação parcial para fornecer critérios de consistência forte sem introduzir pesos significantes de sincronização e sem prejudicar o desempenho.

Nesta dissertação, extendemos o projecto **Escada** com um modelo e um mecanismo de processamento de consultas distribuído, o que é um requisito inevitável num ambiente de replicação parcial. Além disso, explorando características dos protocolos, propomos um cache semântico para reduzir o peso gerado ao aceder a réplicas remotas. Também melhoramos o processo de certificação, ao procurar reduzir os "aborts", utilizando informação semântica presente nas transacções.

Finalmente, para avaliar os protocolos desenvolvidos pelo projecto **Escada**, o cache semântico e o processo de certificação utilizamos um modelo de simulação que combina código simulado e real, o que nos permite avaliar nossas propostas em diferentes cenários e configurações. Mais do que isso, ao invés de usar cargas fictícias, submetemos nossas propostas a cargas baseadas nos "benchmarks" TPC-W e TPC-C.

Abstract

Database replication is an invaluable technique to implement fault-tolerant databases, being also frequently used to improve database performance. Unfortunately, when strong consistency among the replicas and the ability to update the database at any of the replicas are considered, the replication protocols do not scale up. The problem is related to the number of interactions among the replicas in order to guarantee consistency and to the protocols used to ensure that all the replicas agree on transactions' result. Roughly, the number of aborts, deadlocks and messages exchanged among the replicas grows drastically, when the number of replicas increases. In related works, it has been proved that database replication in such a scenario is impractical.

In order to overcome these problems, several studies have been developed. Initially, most of them released the strong consistency and the update-anywhere requirements to achieve feasible solutions. Recently, replication protocols based on group communication were proposed, in which the strong consistency and update-anywhere requirements are preserved and the problems circumvented. This is the context of the **Escada** project. Briefly, it aims to study, design and implement transaction replication mechanisms suited to large scale distributed systems. In particular, the project exploits partial replication techniques to provide strong consistency criteria without introducing significant synchronization and performance overheads.

In this thesis, we augment the **Escada** with a distributed query processing model and mechanism, which is an inevitable requirement in a partially replicated environment. Moreover, exploiting characteristics of its protocols, we propose a semantic cache to reduce the overhead generated while accessing remote replicas. We also improve the certification process, while attempting to reduce aborts using the semantic information available in the transactions.

Finally, to evaluate the **Escada** protocols, the semantic caching and the certification process, we use a simulation model that combines simulated and real code, which allows to evaluate our proposals under distinct scenarios and configurations. Furthermore, instead of using unrealistic workloads, we test our proposals using workloads based on the TPC-W and TPC-C benchmarks.

Acknowledgements

First of all, I would like to thank my parents for all support that I received since I was born. In particular, I want to thank them for the innumerable advises and for teaching me living according to invaluable qualities, mainly honest and simplicity; and one magic word, work. Specially, I want to thank André Conceição Correia, my brother who is no longer alive and whose generosity and goodness will be always remembered. I am sure that if he was alive, his theoretical contributions and support will be essential to complete this work. I also would like to thank my adviser Rui Carlos Mendes Oliveira for accepting me in the Distributed Systems Group at University of Minho and for his support.

I am also thankful for the incredible discussions about the theoretical and practical aspects of this work by Antônio Luis Souza, José Orlando Pereira, Luís Soares and Luciano Miguel Rocha, members of the Distributed Systems Group at University of Minho. I would like also to thank the other members of the group and the Departamento de Informática for the cordiality.

I would like to thank Gustavo Vasconcelos Arnold for establishing the first contacts with my adviser, resulting in the opportunity to do this work. I either cannot forget to thank important friends that besides the distance support me. I would like to thank Alba Couto, Fabricio Pinto, Camilo Telles, Ivy Michelle and Weber Souza for the excellent discussions about computer science and other different subjects. I would like to thank Cirlã Brasil Lopes, Danilo Mota, Eduardo Argollo, Ronaldo Florence and Rui Burgos for their friendship. Specially, I would like to thank Eduardo Argollo for the great moments in La Coruña and Lisbon. I also would like to thank my new friends, brazilian friends that I met here in Portugal: Diógenes Rubert Librelotto, Fábio Cavalcanti, Giovani Rubert Librelotto, Marco Antonio Barbosa, Ricardo Alexandre Martins, Ronnie Cley Alves and Tiago Chaves. For the friends that I did not mention here, I would like to say that I am so sorry but the lack of time and space do not allow me to thank everybody. But remember, friends are forever.

I would like to thank the LaSiD (Laboratório de Sistemas Distribuídos da UFBa) for introducing me in this amazing area called Distributed Systems, in particular, professor Flávio Morais de Assis Silva. I want to thank the FCT (Fundação para Ciência e Tecnologia) for supporting this work through project StrongRep (FCT POSI/CHS/41285/2001).

Finally, I would like to thank all the readers whose suggestions and corrections contributed for the final version of this work. I also would like to apologize them for the English.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The world was the stage of some revolutions that changed the course of our lives forever. And probably, many others will come. Nowadays, we certainly passing through the ages of the information revolution in which information means power. The companies gather information in order to conduct their business and also transform it into knowledge about the market in an attempt to become more competitive. Regardless how the information is used and manipulated, we need a place to store it and mechanisms to easily access it. Ultimately, in a technological point of view, the database systems are conceived in order to provide these features.

The databases systems are at the core of our information society, supporting a wide range of economic, social and public administration activities. The loss of the information or even its unavailability can cause serious damages and in some cases may result in loss of lives. For that reason, we must rely on dependable systems and therefore, strong properties like reliability and availability must be part of our concerns.

Database replication is an invaluable technique to implement dependable databases, being also frequently used to improve database performance. Unfortunately, when strong consistency among the replicas and the ability to update the database at any of the replicas are considered, the replication protocols do not scale up. The problem is related to the number of interactions among the replicas in order to guarantee consistency and to the protocols used to ensure that all the replicas agree on transactions' result. Roughly, the number of aborts, deadlocks and messages exchanged among the replicas grows drastically, when the number of replicas increases. Gray et al. [42], point out that a replicated database with n copies stored over n sites can have a deadlock rate proportional to n^3, which is impractical. Generally speaking, these replication protocols attempt to guarantee that all the replicas have the same state at the end of the transaction (i.e., strong consistency) [9]. For each transaction's operation a lock (i.e., two-phase lock) is acquired at all the replicas for the respective items retrieved or updated. Upon receiving the commit request, signaling the end of the transaction, a termination protocol is started (e.g., two-phase commit, three-phase commit) to ensure that all the replicas achieve the same outcome, commit or abort. The termination protocol needs more than one phase since just one does not allow a database site to unilaterally decide to abort a transaction. In the case of the two-phase commit protocol: (*i*) a site which has the role of a coordinator verifies if all the sites have the intention to commit the transaction, requesting them to prepare to commit; (*ii*) if one site decides to abort the transaction, the coordinator request the abort for all the others, otherwise, it request the commit.

Several efforts have been made to circumvent these problems, most of them, weakening the consistency or withdrawing the ability to update the database at any of the replicas [42, 9]. Basi-

cally, these efforts attempt to establish a trade-off between performance and dependability. However, in some cases the first approach may be infeasible since some applications cannot tolerate weak consistency and the second approach may seriously affect the scalability of the system, introducing bottlenecks. For those reasons, most of the replication protocols available are not widely used in production environments. For instance, one common solution to avoid the overhead of the strong consistency is to return the transactions' outcome, either commit or abort, to the client before propagating the transactions' updates to the replicas. It is simple to see that this solution may generate inconsistency when the ability to update the database at any of the replicas is considered. Therefore, it is usually applied with the election of one replica to receive the updates, which means that the transactions are carried out by this primary replica and afterwards the updates are propagate to the other backup replicas.

Recently, in order to achieve reliability and availability without introducing these problems, a set of replication protocols that exploit group communication was proposed [90, 74, 60, 63, 87]. In particular, protocols such as those presented in [74, 60, 63, 87] allow a transaction to be executed at any site and postpone the interaction among concurrent transactions, which can be seen as an optimistic execution. Upon receiving the commit request, they propagate the set of information read and written by the transaction to all replicas. If conflicts arise among concurrent transactions, the order in which the transactions were delivered is used to decide which transactions must be committed and aborted. The group communication layer (i.e., total order broadcast primitive [49]) guarantees that the order of the transactions is the same at all replicas, allowing that all the sites achieve the same decisions. Database replication based on group communication appears as a promise to overcome the scalability and performance problems of the traditional strong consistency protocols, reducing the interactions among the replicas and eliminating the deadlocks.

However, when we consider large scale distributed systems and several replicas distributed in a wide area network, fully replicated databases may not be suitable. We need to resort to partially replicated databases. Basically, partial replication is done by splitting the database according to the application semantics and then by replicating each fragment at a subset of the available sites. It is invaluable for the scalability and performance of very large and geographically distributed databases. For instance, fragmentation allows less relevant data items to be replicated by fewer sites and access locality allows that data items are kept close to those sites that need them more often. Thus, if each transaction requires only a small subset of all sites to execute and commit, the processing and communication overhead associated with replication can be reduced.

The work presented in this thesis is on the design and implementation of a partial replicated database system in the context of the **Escada** project.[1] **Escada** is a project that aims to study, design and implement transaction replication mechanisms suited to large scale distributed systems. In particular, the **Escada** exploits partial replication techniques to provide strong consistency criteria without introducing significant synchronization and overheads. **Escada** extends the *Database State Machine* (i.e., a full replication approach), or simply DBSM [74], which is based on the group communication approach described before. The previous efforts of the **Escada** project were concerned about the group communication futures, such as optimistic total order in WANs [88], and specifically about the replication protocols for partially replicated databases, *Partial Database State Machine* or simply PDBSM [87].

[1] The project was developed by University of Minho (Departamento de Informática da Universidade do Minho) and funded by FCT POSI/CHS/33792/1999.

1.1 Problem Statement

Partial replication can be troublesome when the database does not transparently handle the fact that not all the fragments accessed by a transaction are located at the same site. In such a scenario, it is difficult to manage and maintain large and geographically distributed databases. For instance, changes in the location of a fragment may imply in changes in the applications. For those reasons, we aim at augmenting the **Escada** [87] with distributed query processing capabilities, which means that the database must transparently handle the transaction's request, accessing the necessary fragments locally or remotely available. Specifically, we concentrate our analyses and efforts on relational databases, regarding the requirements to augment a centralized database.

Unfortunately, the distributed execution may increase transaction's latency and reduce the overall performance. For that reason, it is important to evaluate the impact of the distributed execution on the overall system and to develop mechanisms to help reducing its impact. We evaluate its benefits using a simulation tool that combines real and simulated code. Such a model allows us to evaluate the impact of the design and the implementation decisions of the protocols on the overall performance. In contrast with real systems, this approach allows us to set up and run multiple tests with slight variation of configuration parameters, in scenarios with large number of replicas and wide-area networks. When compared with a fully simulated approach, it gives us the opportunity to estimate the resources required by the protocols, since their are real code that interfaces with the simulated environment. To reduce the impact of the distributed execution on the overall performance, we propose the use of a semantic caching [27, 50] approach, which minimizes the need to contact remote sites in order to answer transactions' requests. Basically, in this approach, the entries in the cache are identified using the predicates involved in the queries. Our approach also takes advantage of the group communication primitives to update or invalidate the entries in cache, eliminating the negative impact of a centralized management. It also avoids the management overhead of the tuples, which usually involves retrieval, update and replacement per tuple. In contrast to page caching, it reduces the management overhead and further overcomes the problem of space consumption, which is a consequence of the page fixed size, while disregarding the size of the result set and always allocating pages.

This replication approach can be infeasible when the amount of read information that needs to be transfered to all the replicas is high. The main reasons are bandwidth consumption and latency. In order to avoid this, it is possible to reference the relations instead of the read elements. However, this solution may increase the number of aborts as a consequence of the coarse grain. To attempt to reduce the number of possible aborts, we propose the use of a "smart certification", which means sending also the queries that reference the relation (i.e., our coarse grain) and in the procedure that verifies the occurrence of conflicts use the queries to see if the conflicts are real or just a consequence of the coarse grains. Similar to the semantic caching, the concepts of this approach are based on the satisfiability problem [47].

Furthermore, the evaluation of the proposals, including the current protocols developed under the **Escada** project, must be handled using realistic workloads. Without them, we cannot effectively evaluate the impact of our decisions on the overall performance since all the problems mentioned before are highly dependent on the application semantics.

1.2 Contributions

This thesis provides five main contributions:

Distributed Execution - It augments the **Escada** project with a distributed query processing protocol, releasing the assumption that all the fragments accessed by a transaction are located at a single site.

Semantic Caching - It proposes the use of a semantic caching to reduce the impact of the distributed execution on the overall performance, exploiting characteristics of the **Escada** protocols. It exploits the existing broadcast primitives to update and invalidate cache entries.

Smart Certification - It proposes a smart certification process, based on the same theories of the semantic caching, which attempts to reduce the number of aborts in consequence of conflicts that arise when relations are referenced.

Integration - It analyzes some issues that could arise when integrating the protocols designed for the **Escada** project into a real database system. It also suggest an integration using the PostgreSQL as the target database.

Evaluation of the PDBSM - It evaluates the **Escada** protocols using realistic workloads and a simulation tool that combines simulated and real code. Specifically, it uses workloads based on the TPC-W [101] and TPC-C [100] benchmarks.

1.3 Thesis Organization

This thesis is organized as follows. In Chapter 2, we present the computational model and definitions used throughout this thesis. In Chapter 3, we describe the **Escada** protocols. Namely, we explain in detail the Partial Database State Machine (PDBSM) which is a novel approach for the partial replication of databases. In Chapter 4, we present distributed query processing mechanisms and design an approach to be used in the PDBSM. In Chapter 5, we present the ideas behind semantic caching and discuss how to exploit it in the PDBSM. In this chapter, we also present the ”smart certification“. In Chapter 6, we design an extension to the PostgreSQL in order to provide a distributed transaction processing. This chapter corresponds to the integration of the protocols designed for the **Escada** project into a real database system. In Chapter 7, we evaluate the PDBSM using realistic workloads. In Chapter 8, we conclude the thesis, summarizing its contributions and outlining possible future work.

Chapter 2

Model and Definitions

In this chapter, we present concepts and definitions required by the chapters that follow. Section 2.1 presents the distributed system model to which the contributions developed in this thesis apply. Section 2.2 presents the database model adopted, considering the sites, the transactions executed on behalf of a client request and the language used to build the requests. In Section 2.3, we introduce the distributed database model. In Section 2.3, we present the communication primitives used, which are responsible for the communication established among the distinct database sites.

2.1 Distributed System

We consider a distributed system composed of a set of database sites $S = \{s_1, \ldots, s_n\}$ which are fully connected. The sites communicate through message passing. The system is asynchronous in that there is no bound on process relative speeds, clock drifts, or communication delays.

Sites can only fail by crashing and we do not rely on site recovery for correctness. However, we assume that when a site recovers, it does so with the state that it had before the failure.[1] Furthermore, we assume that our asynchronous model is augmented with a failure detector oracle so that, consensus is solvable [16].

Since we admit that sites may recover, we distinguish between *running* and *crashing* sites. A running site is a site that takes steps of its automaton; a crashing site takes null steps. A running site that crashes becomes a crashing site; a crashing site that recovers becomes a running site. In our model, a *correct* site is a site that eventually stops crashing [73]. An *incorrect* site is a site that is not correct.

2.2 Database

2.2.1 Relational Database

A relational database $DB = \{R_1, \ldots, R_p\}$ is a set of relations $R_i \subseteq A_1 \times \ldots \times A_q$ defined over data sets not necessarily distinct. Each element $(a_1, a_2, \ldots, a_q)$ of a relation R_i is called a tuple and each a_i is called an attribute. To uniquely identify each tuple of a relation, we assume

[1] This can be easily realized in practice having processes executing on non-volatile memory combined with some mechanism of state transfer [61, 73].

the existence of a minimum non empty set of attributes, called the *primary key*. To reference an attribute a_i of a relation R_i, we use the following expression: $R_i.a_i$.

2.2.2 Transaction

A transaction is a sequence of read and write operations over tuples and it is finished with a commit or abort operation. In other words, a transaction t is represented by a *read set*, a *write set* and *write values*. The read set of t is the set of primary keys identifying the tuples read by t. The write set of t is the set of primary keys identifying the tuples written by t. The write values of t is the set of tuples written by t.

2.2.3 Relational Algebra and Calculus

The transaction can also be represented as a set of read and write operations expressed using a language based on the relational algebra or the relational calculus [71, 98]. The relational algebra can be informally described as a procedural language that specifies how to build a relation from one or more relations in the database. The relational calculus can be described as a non-procedural language used to formulate a definition of a relation in terms of one or more database relations. It has been proved that these languages are equivalent, which means that the same expressions can be formulated using both languages.[2]

Some important operations from the relational algebra are presented bellow:

Selection [$\sigma_f(R)$] - It defines a relation that contains a horizontal subset of *R*, extracting tuples that satisfy the *predicate* f. See the relational calculus for a definition of *predicate*.

Projection [$\pi_{a_p,\ldots,a_q}(R)$] - It defines a relation that contains a vertical subset of *R*, extracting the attributes specified in $a_p, \ldots, a_q$.

Union [$R \cup S$] - It defines a relation that consists of the union of two relations *R* and *S*, which must be domain compatible. Two relations are said to be domain compatible if they are defined over the same sequence of data sets.

Cartesian Product [$R \times S$] - It defines a relation that consists of the cartesian product of two relations *R* and *S*.

Intersection [$R \cap S$] - It defines a relation that consists of the intersection of two relations *R* and *S*, which must be domain compatible.

θ-Join [$R \bowtie_f S$] - It can be seen as a cartesian operation followed by a selection, where the *predicate* f is restricted to $R.a_p\ \theta\ S.a_q$, and θ is a comparison operator $(\leq, \geq, \neq, =, <, >)$. It is one of the most important operations in relational algebra and the most difficult to implement efficiently in relational databases. For an in-depth discussion about join operations see [80].

The relational calculus has the following general form: $\{S_1.a_p, \ldots, S_k.a_q \mid F(S_1, \ldots, S_w)\}$, $k \leq w$, where $S_1, \ldots, S_k$ are tuple variables, each a_i is an attribute of the relation over which S_i

[2] The precise statement is that the set of queries expressible in the algebra is the same as the set of queries expressible in the calculus. For a detailed discussion of this subject see [65].

ranges, and F is a *formula*. A tuple variable is a variable whose allowed values are the tuples of a relation. The *formulas* are structured using the following components:[3]

Tuples [$R(S_i)$], where R is a relation and S_i is tuple variable. For instance $\{S_1 \mid R(S_1)\}$ returns all the tuples of a relation R.

Common tuples [$S_i.a_p\ \theta\ S_j.a_q$], where S_i and S_j are tuple variables, a_p is an attribute of the relation over which S_i ranges, a_q is an attribute of the relation over which S_j ranges, and θ is one of the comparison operators $(\leq, \geq, \neq, =, <, >)$. For instance, $\{S_1, S_2 \mid R(S_1) \wedge (\exists(\cdot\ S_2))(S(S_2) \wedge S_1.a_1 = S_2.a_1)\}$ returns tuples with the attributes of R and S, where $R.a_1 = T.a_1$.

Specific tuples [$S_i.a_p\ \theta\ c$], where S_i is a tuple variable, a_p is an attribute of the relation over which S_i ranges, θ is one of the comparison operators and c is a constant value which belongs to the same domain of $S_i.a_1$. For instance, $\{S_1 \mid R(S_1) \wedge S_1.a_1 = c\}$ returns the tuples from R, where $R.a_1 = c$.

Combined components We can put together one or more formulas using conjunction and disjunction. We also can negate a formula.

Quantifiers In a formula, we can use the existential and universal quantifiers.

The formal and standard language used to manipulate relational databases is the SQL (Structure Query Language), which is based on the relational calculus. The subset of the SQL considered in this thesis is presented as follows:

Select - "***select*** A ***from*** R ***where*** f_s" , where $A = S.a_1 \ldots, S.a_n, T.b_1 \ldots, T.b_m, Y.e_1 \ldots, Y.e_j$, ... defines the projection over the set of attributes of the relations. $R = S, T, Y, \ldots$ defines the set of relations. f_s is the predicate that defines the set of tuples to be selected. This SQL statement establishes a special pattern called SPJ, that is, set of (s)elections, (p)projections and (j)oins.

Insert - "***insert into*** $R(R.a_1 \ldots, R.a_n)$ ***values***$(c_1 \ldots, c_n)$", where $R = S$ is the set of relations,[4] $R.a_1 \ldots, R.a_n$ the attributes to be updated and $c_1 \ldots, c_n$ the respective values, which must belong to the same domain of the attributes.

Delete - "***delete from*** R ***where*** f_d", where $R = S$ is the set of relations and f_d is the predicate that defines which tuples from R must be deleted.

Update - "***update*** R ***set*** $R.a_1 = c_1 \ldots, R.a_n = c_n$ ***where*** f_u", where $R = S$ is the set of relations, $R.a_1 \ldots, R.a_n$ the attributes to be updated, $c_1 \ldots, c_n$ the respective values, which must belong to the same domain of the attributes, and f_u is the predicate that defines which tuples from R must be updated.

[3] This nomenclature based on components belongs to us. It is used to facilitate an informal correlation between the calculus and the relational algebra. Basically, the predicates or formulas from the relational algebra are the formulas from the relational calculus without the quantifi ers.

[4] For the update operations (i.e., delete, insert and update) the cardinality of the set of relations is expressed as $\mid R \mid = 1$.

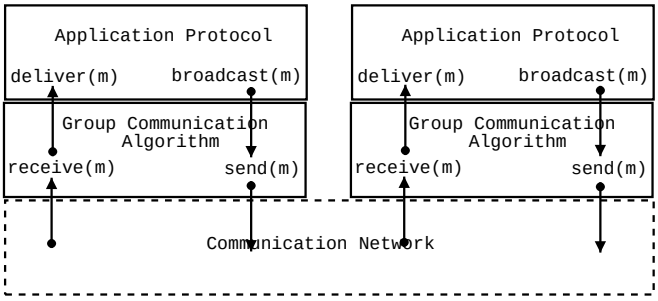


Figure 2.1: Application/Broadcast Layering

2.3 Distributed Database

We consider a distributed relational database as a relational database whose relations are distributed, i.e. fragmented, among the set S of database sites. This distributed database is given by $DDB \subseteq DB \times S$.

The relations of the database can be fragmented horizontally, using a selection operation from the relational algebra, or vertically, using a projection operation. To avoid semantic changes to the relational database as a consequence of the fragmentation process, the following properties must be enforced [71], where the function $frags(R)$ gives the fragments of a relation R:

Completeness The fragmentation cannot generate any loss of information: $R = \bigcup R_i, \forall\ R_i \in frags(R)$.

Reconstruction It must be possible using a relational algebra operation ∇ to rebuild the original relation R as follows: $R = \nabla\ R_i, \forall\ R_i \in frags(R)$. This operation is a *union* in case of horizontal fragmentation and a *join* in case of vertical fragmentation.

Disjointness If a relation *R* is horizontally fragmented, then every two distinct fragments cannot have a single common tuple, i.e. $\forall\ R_i, R_j \in frags(R),\ R_i \cap R_j = \emptyset,\ where\ i \neq j$. If a relation *R* is vertically fragmented, then every two fragments must have the same keys, i.e $\forall\ R_i, R_j \in frags(R),\ R_i \cap R_j =$ *{set of primary key attributes of R}*, $where\ i \neq j$.

An important assumption we make is that for every fragment of a relation there is a correct site that replicates it.

2.4 Group Communication

Group communication is a fundamental building block for developing fault-tolerant distributed applications. It offers strong properties on communication reliability despite failures. Besides reliability, it usually offers message ordering such as FIFO, causal or total order [49].

The database sites communicate exchanging messages by means of group communication primitives. Upon receiving the message, the group communication algorithms, process the message and deliver it to the application according to Figure 2.1. The stage before delivering the message imposes the desired requirements [49].

We assume the existence of a Uniform Reliable broadcast primitive satisfying the following properties [49]:

- **Validity**: If a correct process $broadcast$ a message m, then it eventually $delivers$ m.
- **Uniform Agreement**: If a process $delivers$ a message m, then all correct processes eventually $deliver$ m.
- **Integrity**: For any message m, every correct process delivers m at most once, and only if m was previously broadcast by some process.

Informally, it guarantees that all the correct processes eventually agree on the set of messages to deliver despite failures.

Moreover, we assume the existence of a total order primitive satisfying the following property [49]:

- **Total Order**: If correct processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

Chapter 3

Escada Project

Escada is a project that aims to study, design and implement transaction replication mechanisms suited to large scale distributed systems. In particular, the project exploits partial replication techniques to provide strong consistency criteria without introducing significant synchronization and performance overheads.

Escada extends the *Database State Machine* (i.e., a full replication approach), or simply DBSM [74]. Briefly, the DBSM allows an optimistic local execution, postponing the interaction with remote concurrent transactions. That is, each transaction request is optimistically executed by a single site and interaction with other sites is only initiated after the commit request. Upon receiving the commit, the outcome of the transaction is propagated to all replicas using atomic multicast. A certification procedure is run upon delivery by all sites to determine conflicts with other concurrently executed transactions, and thus whether the transaction should be committed or aborted. Atomic multicast guarantees that all sites deliver the outcome of the transaction in the same order. In the case of a conflict, the certification uses this order to decide which transaction commits or aborts. The determinism of the certification ensures a strong consistency and as the deterministic execution is confined to the certification, no restrictions impairing performance are imposed on scheduling during the execution stage.

The goal of this chapter is twofold. First, we present the motivation to use group communication for replicating databases. Second, we present the protocols developed under the **Escada** project and propose extensions, specifically, to the termination protocol.

The remaining sections of this chapter are organized as follows. In Section 3.1, we present some concepts about database replication and describe possible classifications for the techniques used to replicate data. In Section 3.2, we present in detail the Database State Machine approach for database replication. In Section 3.3, we extend the DBSM to a partial replication environment, presenting the Partial DBSM. We describe the PDBSM's execution model and possible termination protocols. Finally, in Section 3.4, we discuss important issues involved in the implementation of the PDBSM approach.

3.1 Database Replication

Database replication is an invaluable technique to implement fault-tolerant databases, being also frequently used to improve database performance. Unfortunately, when strong consistency among the replicas and the ability to update the database at any of the replicas are considered, the repli-

cation protocols do not scale up. The problem is related to the number of interactions among the replicas in order to guarantee consistency and to the protocols used to ensure that all the replicas agree on transactions' result. Roughly, the number of aborts, deadlocks and messages exchanged among the replicas grows drastically, when the number of replicas increases. Furthermore, Gray et al. [42], point out that a replicated database with n copies stored over n sites can have a dead lock rate proportional to n^3, which is impractical.

Several efforts have been made to circumvent these problems, most of them, weakening the consistency or withdrawing the ability to update the database at any of the replicas. Basically, these efforts attempt to establish a trade-off between performance and dependability. However, in some cases the first approach may be infeasible since some applications cannot tolerate weak consistency and the second approach may seriously affect the scalability of the system, introducing bottlenecks. For those reasons, most of the replication protocols available are not widely used in production environments.

Recently, in order to achieve reliability and availability without introducing these problems, a set of replication protocols that exploit group communication was proposed [90, 74, 60, 63, 87].

According to [42], database replication protocols can be classified based on two parameters: (*i*) when the updates are carried through and (*ii*) who carries them on.

Considering the first parameter, the replication can be classified as synchronous (*eager replication*) or asynchronous (*lazy replication*). In the former, the client has a guarantee that the updates were sent to all the replicas and applied, upon receiving the outcome of the transaction (in this case commit). In order to do that, the synchronization among the replicas occurs inside the transaction, that is, before the confirmation that is sent to the client. On the other hand, in the later approach, the client does not have the guarantee that the replicas are up to date since replication occurs after the confirmation.

Considering the second parameter, the replication can be classified as primary copy or update-anywhere. In the former case, all the updates are carried on by a single site or replica, which incurs in scalability problems. In the later case, the updates can be sent to any replica, but it requires complex protocols to coordinate the consistency among diverse replicas.

Specifically for eager replication protocols, which represent the basis of our study in this thesis, another classification was proposed in [104]. It organizes the eager replications according to three parameters, which are: (*i*) the site architecture, (*ii*) how changes are propagated across sites and (*iii*) the transaction termination protocol.

The first parameter defines primary copy or update-anywhere replication protocols and it is based on the ideas from [42].

The second parameter considers the number of messages exchanged to handle the operations of a transaction, except its termination, as follows:

Constant Interaction characterizes the protocols that, independently of the operations in the transactions, have a constant number of messages. Basically, protocols that correspond to this classification group all the operations before sending. It is important to remember that the operations can be represented using SQL, relational algebra, relational calculus or as a set of write values, read and write sets.

Linear Interaction characterizes the protocols that generate a message on a per operations basis.

The last parameter determines how atomicity is guaranteed, which means how transactions are terminated, and it is presented as follows:

Voting Termination requires an extra set of messages to coordinate the commitment of the transaction, which can be as complex as the atomic commitment protocol (e.g., two-phase commit or three-phase commit [9]) or a simple confirmation message.

Non-Voting Termination uses a deterministic process to decide the outcome of the transaction, allowing that each site achieves the same decision without a coordination.

The replication can also be classified as full or partial. In the full replication scenario, all sites have copies of all the relations in a database. In contrast, partial replication has the database split according to application semantics and each fragment replicated at a subset of the available sites. It exploits access locality allowing each transaction to require only a small subset of all sites to execute and commit, thus reducing processing and communication overhead associated with replication. Partial replication is an alluring technique to ensure the reliability of very large and geographically distributed databases while, at the same time, offering good performance. The advantages of partial replication have however to be weighted against the added complexity that is required to manage it. In fact, if the chosen configuration cannot make transactions execute locally or if the overhead of consistency protocols offsets the savings of locality, potential gains cannot be realized.

Finally, it is important to understand the issues related to data distribution and allocation for a broad knowledge about replication. The resource allocation problem in a network computing environment is a well known problem [71], which can be classified as static or dynamic. In static allocation, considering the fragments $F = \{f_1, f_2, ..., f_n\}$, the sites $S = \{s_1, s_2, ..., s_i\}$ and the set of transactions $T = \{t_1, t_2, ..., t_x\}$ that accesses the fragments F. The allocation problem consists on finding a distribution of the fragments F over the sites S, according to a pattern access, i.e. the application semantics, to minimize an objective function such as resource consumption or response time. It is well known that this problem is NP-Hard in the number of fragments and sites [30]. In addition, the access pattern is difficult to gather and can change over time, invalidating the established distribution [21].

For those reasons, dynamic allocation techniques have been developed. It can be classified according to the objective function as follows: (*i*) minimize communication cost in WANs, **migrating** data to servers near to the places with the highest access or (*ii*) load balancing in LANs, **replicating** the data recently accessed. For instance, in the first scenario, there is an interesting solution: Mariposa [93] that uses an economic model to migrate its fragment.

Recently, replication protocols based on group communication are receiving a lot of attention. The reason is that it appears as a promise to overcome the scalability and performance problems of the traditional strong consistency protocols. Using the classification presented in this section, the replication can be characterized as (*i*) eager and (*ii*) update-anywhere. With respect to the number of messages we are interested in protocols with a (*iii*) constant interaction among the replicas [74, 60, 63, 87] and with respect to the termination protocol, some variations can arise according to the requirements adopted. See Sections 3.2 and 3.3 for a detailed description.

3.2 The Database State Machine

The Database State Machine [74], depicted in Figure 3.1, is based on the deferred update replication technique [9] which reduces the need for distributed coordination among concurrent transactions during their execution. Using this technique, a transaction is locally synchronized during its execution at the database where it initiated according to some concurrency control mechanism [9]

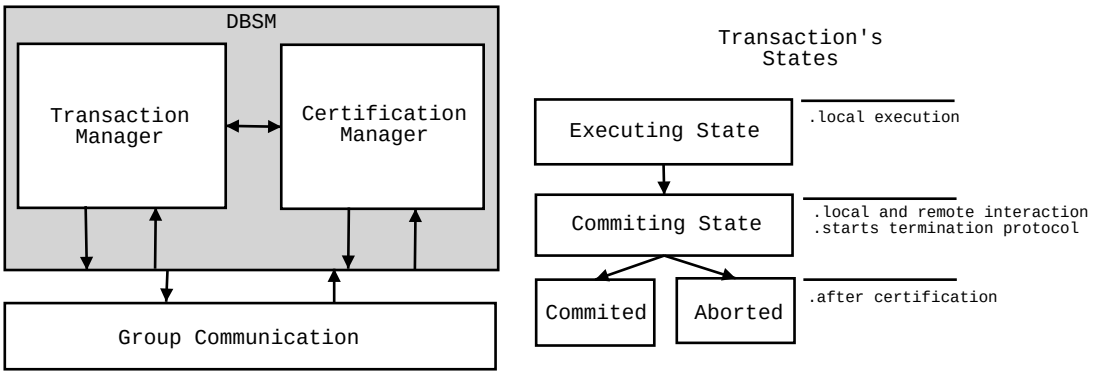


Figure 3.1: DBSM Architecture and Transaction's States

(e.g., two-phase locking, multiversion). From a global point of view, the transaction execution is optimistic since there is no coordination with any other database site possibly executing some concurrent transaction. Interaction with other database sites on behalf of the transaction only occurs when the commit is requested. At this point, a termination protocol[1] is started: i) the transaction write values, read and write sets are atomically propagated to all database sites, and ii) each database site certifies the transactions determining its fate: commit or abort. Summarizing, from the time it starts until it finishes, a transaction passes through some well-defined states. A transaction is considered to be in the *executing state* as soon as the request is received by an *initiator site* and until a commit operation is issued. The transaction then enters the *committing state* and the distributed termination protocol is started.

In order for a database site to certify a committing transaction t, the site must be able to determine which transactions conflict with t. A transaction t' *conflicts with* t if: i) t and t' have conflicting operations and ii) t' does not *precede* t.

Two operations conflict when they are issued by different transactions, access the same data item (e.g., a tuple) and at least one of them is a write operation. The precedence relation between transactions t and t' is denoted $t' \rightarrow t$ (i.e., t' precedes t) and defined as: i) if t and t' execute at the same database site, t' precedes t if t' enters the committing state before t; or ii) if t and t' execute at different sites, for example s_i and s_j, respectively, t' precedes t if t' commits at s_i before t enters the committing state at s_i.

3.3 Partially Replicated Database State Machine

The DBSM is based on a full replication scenario. Releasing the assumption that each database site contains a full copy of the database, directly impacts both the execution and the certification of transactions. In this section, we address the issues raised by partial replication in the Partial Database State Machine (PDBSM). In detail, we address the execution model and two possible termination protocols that deal with partial replication, with either independent or coordinated certification.

[1] The DBSM claims to provide 1-copy-serializability [9] as its consistency criterion [74]. See Chapter 6 for a detailed discussion of this subject.

3.3.1 Transaction Execution

Unlike the DBSM, the initiator site in a partial replication setting may not be able to locally complete the execution of a transaction. In fact, it is possible that no single site can, if the required fragments are nowhere held together. Therefore, the execution of a transaction t in the PDBSM requires that the initiator site s_i coordinates the distributed processing of t among a set of sites that together contain all the fragments accessed by t. Roughly, the initiator site s_i goes through the following steps [66]: i) it parses each request and rewrites the operations mapping the original database relations into the actual fragments, ii) selects the appropriate sites for each fragment accessed and iv) starts the distributed execution. Concurrency control is performed locally by each site when executing operations on behalf of the initiator site.

The proposed distributed transaction execution behaves as a nested transaction [6], in the sense that the initiator can spawn one different subtransaction per site in order to process the statements. Our distributed transaction execution has the following properties:

1. The initiator can spawn one subtransaction per site and each site performs its own concurrency control mechanism.

2. Only the initiator can spawn subtransactions, which avoids the possibilities of deadlocks inside the same transaction. Considering that we are interested in avoiding deadlocks, others sites rather than the initiator could start subtransactions if we disable the possibilities of a child transaction to access the same sites of its parent and vice-versa. Nevertheless, we consider that more than one level of subtransaction is not an important feature since we are not concerned in modeling protocols to deal with long-lived transactions [6]. Besides this, the protocol presented in this chapter could be easily extend to support this future.

3. Deadlocks among concurrent transactions can arise when the sites resort on locks as a concurrency control mechanism. In order to avoid this, we assume that the subtransactions execute optimistically at remote sites and we only rely on the concurrency control mechanisms of the initiator site to control its own transactions. For a detailed discussion about deadlock see [67].

4. Upon the initiator abort, all the subtransactions are also aborted.

5. Upon a child abort, all the subtransactions are also aborted. It is important to notice that one of the objectives of the nested transactions is to provide unit of recovery, which can be easily done extending the grammar of the database without change our model. However, since we propose to augment the PostgreSQL with a distributed execution model (see Chapter 6) and it does not allow subtransactions, we suggest this default behavior in case of aborts.

6. We do not allow intra-transaction parallelism. To do that would be required deep modifications in the processing query mechanisms of the PostgreSQL and for that reason we do not model this future.

3.3.2 Termination Protocol

An issue of major impact for the PDBSM is how the results of the distributed transaction processing are handled. While in the DBSM, the whole set of write values, read and write sets were relevant to all database sites, in a fragmented database this is no longer true. On the contrary,

the fragmentation of the database is meant to exploit data and operation locality and therefore the propagation of write values should be restricted to the sites replicating the involved fragments.

With respect to the read and write sets, however, it is not obvious whether they should be propagated to all database sites or just to those containing the relevant fragments. Indeed, this directly influences the certification phase and establishes a trade-off between network usage and protocol latency. If the whole read and write sets of the transaction are fully propagated, then it will enable each site to independently certificate the transaction. Otherwise, if each site is provided with only the parts of the read and write sets regarding the site's fragments, then it can only make a partial judgment and the transaction certification requires a final coordination among all sites.

Independent Certification

With the propagation of the whole read and write sets to all database sites, we adopt a termination protocol similar to that of the DBSM, in which each site can independently certify the transactions. As soon as a transaction t enters the committing state, all sites containing database fragments involved in the transaction are requested to *stabilize* the transaction write values. The stabilization of a fragment ensures that all sites are able to participate on the termination protocol and to eventually commit the transaction despite the failure of the sites involved in the transaction execution. By request of the initiator site s_i, each database site s_j involved on the execution reliably multicasts the transaction write values of the fragments it is responsible for to all sites replicating these fragments. The initiator s_i receives a stabilization acknowledgment, in the form of read and write sets, from s_j. Should s_j fail, s_i may unilaterally decide to abort the transaction.

Once the initiator site gathers all the acknowledgments, it atomically multicasts the transaction read and write sets to all database sites. This message totally orders the certification of t, and thus upon delivery of the message, along with the write set of previously certified transactions, each site has the necessary knowledge to certify t. If t passes the certification test, all write values of t previously obtained during t's stabilization phase are applied to the database and t passes to the *committed state*. Otherwise, t passes to the *aborted state*.

The cost of independent certification is given by the cost in network bandwidth of propagating the whole read and write sets to all sites, plus the cost, at each site, of keeping this write set while required for the certification of pending transactions, and finally the cost of certifying the whole transaction at each site. From these, our main concern is actually on the network usage. The write set of a transaction t can be discarded as soon as t is known to precede every pending transaction, that is, any transaction in the executing or committing state. The difference in the cost of doing total or partial certification is almost negligible.

Coordinated Certification

On the other hand, to fully exploit data locality we restrict the propagation of the transaction read and write sets to the database sites replicating the corresponding fragments. The knowledge required to certify a transaction becomes itself fragmented and each site may only be able to certify part of the transaction. Therefore, a final coordination protocol is required.

Once the transaction reaches the committing state the initiator site requests all sites containing database fragments involved in the transaction execution to stabilize. Each one of these sites reliably multicasts the write values and read and write sets of the transaction's fragments it is responsible for to all sites replicating these fragments, and acknowledges the end of the stabilization.

When the initiator site gathers all the acknowledgments it atomically multicasts a message to all sites to totally order the certification of t. Upon the delivery of the message, each database site s_j certifies t against the fragments it replicates and votes on a Resilient Atomic Commitment (RAC) [87] protocol to decide the final state of t. This protocol allows participants to decide *commit* even if some of the replicas of a fragment read or written by the transaction are suspected to have failed, as a single representative from each fragment suffices. Resilient Atomic Commit satisfies the same agreement and termination properties of Weak Non-Blocking Atomic Commit [45] and is defined as follows:

- **Agreement:** No two participants decide differently.
- **Termination:** Every correct participant eventually decides.
- **Validity:** If a site decides *commit* for t, then for each fragment accessed by t there is at least a site s_i replicating it that voted *yes* for t.
- **Non-triviality:** If for each fragment accessed by t there is at least a site s_i replicating it that votes *yes* for t and is not suspected to have failed, then every correct site eventually decides *commit* for t.

If the outcome of the RAC is commit, then all write values of t previously obtained during t stabilization phase are applied to the database and t passes to the *committed state*. Otherwise, t passes to the *aborted state*.

Under the assumption that each fragment is replicated by a correct site that does not fail, the RAC protocol is trivially implemented by having each site multicasting its vote [82]. A site decides upon receiving a vote from at least a representative of each database fragment.

3.4 Implementation Issues

In this section, we discuss in detail possible algorithms to implement the protocols introduced before. Furthermore, we point out an important optimization to the PDBSM, called *FastAtomic Delivery*. This optimization was included in our PDBSM prototype and contribute to the experimental results presented in Chapter 7. We chose to present it separately to avoid cluttering the description of the protocols with performance oriented concerns.

3.4.1 Algorithms

In this section, we present algorithms, using a VDM-SL notation [32], to materialize the distributed execution and termination protocols, discussing possible optimizations according to issues that affect each algorithm. However, the Certification [74] and the Resilient Atomic Commitment (RAC) [87] protocols were studied before and for that reason, we do no present them here.

Transaction Execution

Following, we outline an algorithm that captures the ideas of the distributed transaction execution. Details about the termination protocol are presented in Section 3.3.2 and the topics about SQL processing are discussed in Chapter 4.

In our algorithm the structure *nodeOperation* defines the operation[2] to be processed. The operations are based on SQL and can be a "select", "insert", "update" or a "delete" statement, a "begin" to delineate the start of a transaction or a "commit" or "abort" to establish its end. The structure *replicas* defines a map between the relations (i.e., fragments) and the sites. The structure *transControl* traces distributed execution recording which sites were accessed during the transaction's execution. Specifically, it does that recording which relations were accessed and for each relation which site was used to process the statement. Furthermore, it stores the write values, read and write sets and registers which site is the initiator. The mechanism used to extract the write values, read and write sets are explained in detail in Section 6.1. Finally, in order to organize these information, we group them in a structure called *DDb*, which stands for Distributed Data$b$ase.

The functions are organized in three main categories:

- Interfaces that must be provided by the database implementations in order to use our protocols. These interfaces can be identified by "Db" at the end of their names like *initProtocolDb*.
- Interfaces responsible to establish a remote communication with other sites.
- Generic functions that do not belong to the other categories and actually outline our concerns.

The function *processOperation* is the main point of our algorithm and does the following: (*i*) verifies the operation's class (i.e., *enumOperations*) and (*ii*) then calls the appropriate function. In case of the "begin", it calls the function *initProtocol* that is responsible to start a subtransaction. It is important to observe that a request arriving at a initiator on behalf of client must be globally identified and mapped to each subtransaction started, including the subtransaction created at the initiator site. For a globally identification, it could be used the "site id" combined with a local "transaction id". The other operations trigger the function *executeCommand* which defines the possible sites that can handle the operation and contact one of them. The set of possible sites is chosen based on the assumption that, during execution of transaction t, whenever a site s_i is used to process a statement in which a relation R appears on, the site s_i will be chosen for future requests that reference R during the activities of the transaction t. This behavior, for instance, avoids to update a relation in one replica and read stale information from another. It also guarantees that it will not have cycles inside a transaction, avoiding possible deadlocks that can arise as a consequence of this situation. In particular, the function *possibleSite* is responsible for that. In case that a set of sites can be used, the function *chooseSiteDb* tries to find the best one according to cost policies, which are described in detail in Chapter 4. The first time that a site is contacted, which is observed using the structure *DDb.transDb.access*, a new local transaction is started.

```
1.0  nodeOperation :: classOperation : enumOperations
 .1                    relationId : relation

 .2  inv opr ≙
 .3      (opr.classOperation ∈ {SELECT, INSERT, UPDATE, DELETE} ∧ opr.relationId ≠
[]) ∨
 .4      (opr.classOperation ∈ {COMMIT, ABORT} ∧ opr.relationId = []);

2.0  enumOperations = SELECT | INSERT | UPDATE | DELETE |
 .1                    BEGIN | COMMIT | ABORT;
```

[2]For a discussion about how to transform SQL in relational algebra operations see Section 4.1.

3.0 $transControl = transId \xrightarrow{m} controlInformation;$

4.0 $controlInformation :: access : relation \xrightarrow{m} site$
.1 $dataSet : storedSets$
.2 $siteOrig : site$

.3 inv $ctr \triangleq (\forall\, r \in \mathsf{dom}\ ctr.access \cdot r \neq []) \wedge$
.4 $(\forall\, s \in \mathsf{rng}\ ctr.access \cdot s \neq []);$

5.0 $transOrig :: idOrig : transId$
.1 $siteOrig : site$
.2 $siteExecution : site$

.3 inv $tx \triangleq (tx.siteOrig \neq []) \wedge (tx.idOrig \geq 0);$

6.0 $storedSets :: rs : relation \xrightarrow{m} pk\text{-set}$
.1 $ws : relation \xrightarrow{m} pk\text{-set}$
.2 $wv : relation \xrightarrow{m} tuple\text{-set}$

.3 inv $sset \triangleq$
.4 $(\forall\, idrs \in \mathsf{rng}\ sset.rs \cdot idrs \neq \{\}) \wedge$
.5 $(\forall\, idws \in \mathsf{rng}\ sset.ws \cdot idws \neq \{\}) \wedge$
.6 $(\forall\, idwv \in \mathsf{rng}\ sset.wv \cdot idwv \neq \{\});$

7.0 $replicas = relation \xrightarrow{m} site\text{-set}$

.1 inv $rep \triangleq$
.2 $(\forall\, s \in \mathsf{rng}\ rep \cdot s \neq \{\}) \wedge$
.3 $(\forall\, sl \in \bigcup \mathsf{rng}\ rep \cdot sl \neq []) \wedge$
.4 $(\forall\, r \in \mathsf{dom}\ rep \cdot r \neq []);$

8.0 state DDb of
.1 $transDb : transControl$
.2 $repDb : replicas$
.3 $localDb : site$
.4 $transMapDb : transId \xrightarrow{m} transId$
.5 inv $ddb \triangleq$
.6 $(\forall\, tid \in \mathsf{dom}\ ddb.transDb \cdot tid \in \mathsf{dom}\ ddb.transMapDb) \wedge$
.7 (let $tid \in \mathsf{dom}\ ddb.transDb$ in
.8 $(\forall\, rra \in \mathsf{dom}\ ddb.transDb\,(tid).access \cdot rra \in \mathsf{dom}\ ddb.repDb) \wedge$
.9 $(\forall\, rrb \in \mathsf{dom}\ ddb.transDb\,(tid).dataSet.rs \cdot rrb \in \mathsf{dom}\ ddb.repDb) \wedge$
.10 $(\forall\, rrc \in \mathsf{dom}\ ddb.transDb\,(tid).dataSet.ws \cdot rrc \in \mathsf{dom}\ ddb.repDb) \wedge$
.11 $(\forall\, rrd \in \mathsf{dom}\ ddb.transDb\,(tid).dataSet.wv \cdot rrd \in \mathsf{dom}\ ddb.repDb) \wedge$
.12 $(\forall\, ssa \in \mathsf{rng}\ ddb.transDb\,(tid).access \cdot ssa \in \bigcup \mathsf{rng}\ ddb.repDb) \wedge$
.13 $(ddb.transDb\,(tid).siteOrig \in \bigcup \mathsf{rng}\ ddb.repDb))$
.14 end

9.0 $regSite : transOrig \times site \to transOrig$

.1 $regSite\,(tx, s) \triangleq$
.2 mk-$transOrig\,(tx.idOrig, tx.siteOrig, s);$

```
10.0  regControl : transOrig × transControl → transControl
 .1   regControl (tx, txControl) ≙
 .2     txControl†{tx.idOrig ↦ mk-controlInformation ({↦}, mk-storedSets ({↦}, {↦
}, {↦}), tx.siteOrig)}
 .3   pre  tx.idOrig ∉ dom txControl ;

11.0  regContact : relation × site × relation -m-> site → relation -m-> site
 .1   regContact (rel, s, access) ≙
 .2     access † {rel ↦ s};

12.0  processOperation : transOrig × nodeOperation -o-> 𝔹
 .1   processOperation (tx, opr) ≙
 .2     ( cases  opr.classOperation:
 .3         BEGIN → return (initProtocol (tx)),
 .4         COMMIT, ABORT → return (terminateProtocol (tx, opr.classOperation)),
 .5         others → return (executeCommand (tx, opr))
 .6       end)
 .7   pre  tx.siteOrig ∈ ⋃ rng DDb.repDb ;

13.0  initProtocol : transOrig -o-> 𝔹
 .1   initProtocol (tx) ≙
 .2     (dcl r : 𝔹 := false;
 .3      r := initProtocolDb (tx, DDb.transMapDb);
 .4      if (r = true)
 .5      then DDb.transDb := regControl (tx, DDb.transDb);
 .6      return (r) )
 .7   pre  tx.idOrig ∉ dom DDb.transDb ∧ tx.idOrig ∉ dom DDb.transMapDb
 .8   post tx.idOrig ∈ dom DDb.transDb ∧ tx.idOrig ∈ dom DDb.transMapDb ;

14.0  initProtocolDb (tx : transOrig, mapLocal : (transId -m-> transId)) r : 𝔹
 .1   pre  tx.idOrig ∉ dom mapLocal
 .2   post tx.idOrig ∈ dom mapLocal ;

15.0  executeCommand : transOrig × nodeOperation -o-> 𝔹
 .1   executeCommand (tx, opr) ≙
 .2     (dcl c : site,
 .3          r : 𝔹 := false,
 .4          s : site-set := {};
 .5      s := possibleSites (tx, opr.relationId);
 .6      c := chooseSiteDb (tx, opr, DDb.repDb, s);
 .7      if (tx.siteExecution = c)
 .8      then (dcl rmaps : relation -m-> site := DDb.transDb (tx.idOrig).access;
```

```
.9          r := executeCommandDb (tx, opr, DDb.transMapDb);
.10         if (r = true)
.11         then (rmaps := regContact (opr.relationId, c, rmaps);
.12               DDb.transDb(tx.idOrig).access := rmaps))
.13    else (dcl settx : transOrig := regSite (tx, c),
.14              rmaps : relation -m-> site := DDb.transDb (settx.idOrig).access;
.15         if ({c} \ rng DDb.transDb (settx.idOrig).access ≠ {})
.16         then (r := processOperationRemoteSite (settx, mk-nodeOperation (BEGIN, []), c);
.17               if (r = true)
.18               then (rmaps := regContact (opr.relationId, c, rmaps);
.19                     DDb.transDb(settx.idOrig).access := rmaps;
.20                     r := processOperationRemoteSite (settx, opr, c)))
.21         else (r := processOperationRemoteSite (settx, opr, c);
.22               if (r = true)
.23               then (rmaps := regContact (opr.relationId, c, rmaps);
.24                     DDb.transDb(settx.idOrig).access := rmaps)));
.25    return (r) )
.26 pre  opr.relationId ∈ dom DDb.repDb ∧ tx.idOrig ∈ dom DDb.transDb
.27 post opr.relationId ∈ dom DDb.transDb (tx.idOrig).access ;

16.0 possibleSites : transOrig × relation -o-> site-set
.1   possibleSites (tx, rel) ≜
.2     (if (rel ∈ dom DDb.transDb (tx.idOrig).access)
.3      then return ({DDb.transDb (tx.idOrig).access (rel)})
.4      else if (tx.siteExecution ∈ DDb.repDb (rel))
.5           then return ({tx.siteExecution})
.6           else return ({r | r ∈ DDb.repDb (rel)}) )
.7   pre  tx.idOrig ∈ dom DDb.transDb ∧
.8        rel ∈ dom DDb.repDb∧rng DDb.transDb (tx.idOrig).access ⊆ ⋃ rng DDb.repDb
;

17.0 chooseSiteDb (tx : transOrig, opr : nodeOperation, rep : replicas, s : site-set) r : site
.1   pre  opr.relationId ∈ dom rep ∧ s ≠ {} ∧ s ⊆ ⋃ rng rep ∧ tx.siteOrig ∈ ⋃ rng rep
.2   post r ∈ s ;

18.0 processOperationRemoteSite (tx : transOrig, opr : nodeOperation, s : site) r : 𝔹
.1   post true ;

19.0 executeCommandDb (tx : transOrig, opr : nodeOperation, mapLocal : (transId -m-> transId)) r : 𝔹
.1   pre  tx.idOrig ∈ dom mapLocal
.2   post true ;

20.0 terminateProtocol (tx : transOrig, opr : enumOperations) r : 𝔹
.1   pre  tx.idOrig ∈ dom DDb.transDb
.2   post true
```

Independent Certification and Coordinated Certification

We present an algorithm to capture the termination protocol called independent certification. The structure *transaction* gathers the write values, read and write sets produced during the transaction's execution. Unfortunately, this information is distributed among the database sites that handled the transaction and thus must be gathered to initiate the certification.

The termination protocol is triggered upon receiving the "commit" or the "abort" operation. In the case of an "abort", the procedure is simple. The initiator site contacts the sites involved in the distributed execution requesting to roll back the transaction, which is done calling the function *requestRollBack*. In case of "commit", it stabilizes the write values and then atomically multicasts the transaction and waits for the outcome of the certification, calling the function *startCertification*. In order to stabilize, it calls the function *startStabilization*, which finds the sites that handled the distributed execution and proceeds with the function *computeStabilization*. In it, for each site involved is requested the stabilization calling *requestStabilize*. Upon receiving the answers, it gathers the results using the function *rebuildTransaction*. Basically, this function accumulates the answers *collectedSets* in the structure *transaction* that will be used in certification. However, there is a subtle detail in it. It assumes that just one site s_i will contribute with information about the read and write sets of a relation R (see its pre-condition). It is important to remember that this is a consequence of the distributed execution presented in Section 3.3.1.

```
21.0  collectedSets :: rs : relation -m-> pk-set
.1                     ws : relation -m-> pk-set
.2                     wv : relation -m-> tuple-set

.3  inv cset ≙
.4      (∀ idrs ∈ rng cset.rs · idrs ≠ {}) ∧
.5      (∀ idws ∈ rng cset.ws · idws ≠ {}) ∧
.6      (∀ idwv ∈ rng cset.wv · idwv ≠ {});

22.0  disregControl : transOrig × transControl → transControl
.1  disregControl (tx, txControl) ≙
.2    {tx.idOrig} ⩤ txControl

23.0  processOperationRemoteSite (tx : transOrig, opr : nodeOperation, s : site) r : 𝔹
.1  post true ;

24.0  terminateProtocol : transOrig × enumOperations -o-> 𝔹
.1  terminateProtocol (tx, opr) ≙
.2    (dcl r : 𝔹 := false;
.3     cases opr:
.4       ABORT → return (executeRollBack (tx, rng DDb.transDb (tx.idOrig).access)),
.5       COMMIT →
.6            (r := executeStabilization (tx);
.7             if (r = true)
.8             then r := processCertificationRemoteSite (tx)
.9             else executeRollBack(tx, rng DDb.transDb (tx.idOrig).access) )
.10    end ;
.11    return (r) )
```

```
.12  pre  tx.idOrig ∈ dom DDb.transDb
.13  post tx.idOrig ∉ dom DDb.transDb ;

25.0 executeRollBack : transOrig × site-set →^o 𝔹
.1   executeRollBack (tx, setSites) ≙
.2     (if (setSites ≠ {})
.3      then (let s ∈ setSites in
.4              if (s ≠ DDb.localDb)
.5              then processOperationRemoteSite( tx, mk-nodeOperation (ABORT, []), s)
.6                   executeRollBack(tx, setSites \ {s}) )
.7      else (DDb.transDb := disregControl (tx, DDb.transDb);
.8              executeRollBackDb(tx, DDb.transMapDb) );
.9      return (true) )
.10  pre  tx.idOrig ∈ dom DDb.transDb ∧ tx.idOrig ∈ dom DDb.transMapDb
.11  post tx.idOrig ∉ dom DDb.transDb ∧ tx.idOrig ∉ dom DDb.transMapDb ;

26.0 executeRollBackDb (tx : transOrig, mapLocal : transId →^m transId) r : 𝔹
.1   post tx.idOrig ∉ dom mapLocal ;

27.0 executeStabilization : transOrig →^o 𝔹
.1   executeStabilization (tx) ≙
.2     (dcl stabilizeSites : site-set := rng DDb.transDb (tx.idOrig).access,
.3          r : 𝔹 := false;
.4      if (DDb.certMode = NOVOTE)
.5      then r := computeStabilizationNoVote (tx, stabilizeSites)
.6      else r := computeStabilizationVote (tx, stabilizeSites);
.7      return (r) )
.8   pre  tx.idOrig ∈ dom DDb.transDb
.9   post let setrs = dom DDb.transDb (tx.idOrig).dataSet.rs,
.10           setws = dom DDb.transDb (tx.idOrig).dataSet.ws in
.11       ∀ rrs ∈ setrs, rws ∈ setws ·
.12            rrs ∈ setws ∧ rws ∈ setrs ;

28.0 computeStabilizationNoVote : transOrig × site-set →^o 𝔹
.1   computeStabilizationNoVote (tx, sites) ≙
.2     (if (sites = {})
.3      then return (true)
.4      else let site ∈ sites in
.5           let collectedSet = processStabilizationRemoteSite (tx, site) in
.6           if (collectedSet ≠ nil )
.7           then (rebuildTransaction(tx, collectedSet) ;
.8                 computeStabilizationNoVote(tx, sites \ {site}) )
.9           else return (false) );
```

```
29.0  computeStabilizationVote : transOrig × site-set →o 𝔹
 .1   computeStabilizationVote (tx, sites) ≙
 .2     if (sites = {})
 .3     then return (true)
 .4     else let site ∈ sites in
 .5          let collectedSet = processStabilizationRemoteSite (tx, site) in
 .6          if (collectedSet ≠ nil )
 .7          then computeStabilizationVote(tx, sites \ {site})
 .8          else return (false) ;

30.0  rebuildTransaction : transOrig × collectedSets →o 𝔹
 .1   rebuildTransaction (tx, collectedSet) ≙
 .2     (dcl r : 𝔹 := true,
 .3          setrs : relation →m pk-set := DDb.transDb (tx.idOrig).dataSet.rs,
 .4          setws : relation →m pk-set := DDb.transDb (tx.idOrig).dataSet.ws;
 .5      DDb.transDb(tx.idOrig).dataSet.rs := setrs ⌊m⌋ collectedSet.rs;
 .6      DDb.transDb(tx.idOrig).dataSet.ws := setws ⌊m⌋ collectedSet.ws;
 .7      return (r) )
 .8   pre let setrs = dom DDb.transDb (tx.idOrig).dataSet.rs,
 .9          setws = dom DDb.transDb (tx.idOrig).dataSet.ws in
 .10     tx.idOrig ∈ dom DDb.transDb ∧
 .11     ¬ (dom collectedSet.rs ⊆ setrs) ∧ ¬ (dom collectedSet.ws ⊆ setws) ;

31.0  processStabilizationRemoteSite (tx : transOrig, cst : site) r : [collectedSets]
 .1   post true ;

32.0  processCertificationRemoteSite (tx : transOrig) r : 𝔹
 .1   post true
```

Implicit Functions

There are some implicit functions presented before which deserve more explanation:

Requesting stabilization - Ultimately the stabilization is done calling the function *processStabilizationRemoteSite* for each site involved in the distributed transaction. However, it may fail while contacting the remote site(s). For that reason, its return *collectedSets* can be *nil*, which is indicated by using the brackets around *collectedSets*. Each site communicates with its replicas using a uniform reliable multicast. Furthermore, to inhibit the situation in which a site s_j fails after accomplishing the stabilization and before answering the request to the initiator, we can devise a protocol in which the initiator is also involved in the uniform reliable multicast.

Starting certification - The initiator atomically multicasts the transaction and waits for the outcome of the certification calling *processCertificationRemoteSite*. It also should fail and in this case, a false value is returned to indicate failure and a true value is used to indicate success. No matter what happens, the end of the transaction must be reported and the information recorded in *transControl* eliminated.

Processing a remote command - Should the contact with a remote site fail and in this case, a false value is returned to indicate failure and a true value is used to indicate success.

Failures

Failure of the initiator Considering that the clients access the sites using unmodified mechanisms such as JDBC [96], which usually uses unicast communication, the failures of the initiator site must roll back the transaction, since the connection with the clients will be interrupted. In this case, the other sites detected the failure of the initiator based on changes in the membership and undo the effects of the subtransactions.

Failure of other sites The failure of other sites can also be detected with the changes in the membership, immediately after the failure or postponed to the moment that the initiator needs to contact these sites.

3.4.2 FastAtomic Delivery

The idea behind this protocol is simple and consists on the early delivery of transactions with a tentative order, allowing an optimistic certification to run concurrently to the total order protocol. This allows to overlap the final delivery of the transaction with its certification, whenever the tentative order matches the final delivery order.

The Atomic Broadcast and Fast Atomic Broadcast are the communication abstractions used by database sites to communicate. The Atomic Broadcast is defined by the primitives *broadcast*(m) and *deliver*(m), and satisfies the following properties [49]: [3]

- **Validity**: If a correct process $broadcast$ a message m, then it eventually $delivers$ m.
- **Agreement**: If a correct process $delivers$ a message m, then all correct processes eventually $deliver$ m.
- **Integrity**: For any message m, every correct process delivers m at most once, and only if m was previously broadcast by the senders of m.

When using an atomic broadcast primitive, all sites must wait until they agree on message order before atomically delivering it. In the following, we present the Fast Atomic Broadcast, which allows sites to deliver messages tentatively, that is, before the order has been agreed.

The Fast Atomic Broadcast protocol is similar to the Atomic Broadcast with Optimistic Delivery, introduced in [62]. It is defined by the primitives *broadcast(m)*, *FST-deliver(m)*, and *FNL-deliver(m)* and satisfies the following properties:

Validity. If a correct site broadcasts a message m, then it eventually FNL-delivers m.

FST-Agreement. If a correct site FST-delivers a message m, then every correct site also FST-delivers m.

FNL-Agreement. If a correct site FNL-delivers a message m, then every correct site eventually FNL-delivers m.

[3] These properties were also defined in Chapter 2 but are presented here again for simplicity and to allow a easy comparison with the properties defined by the Fast Atomic Broadcast.

Integrity. For every message m, every site FST-delivers m only if m was previously broadcast; and every site FNL-delivers m only once, and only if m was previously broadcast.

Local Order. No site FST-delivers a message m after having FNL-delivered m.

Final Order. If two sites FNL-deliver two messages m and m', then they do so in the same order.

Notice that if a site FST-delivers a message and then changes its initial guess, i.e. the final order do not matches the fast order, it is the responsibility of the application to cope with messages FST-delivered in the wrong order.

Chapter 4

Distributed Query Processing

In this chapter, we present the distributed query processing issues and analyze the most important mechanisms developed to accomplish its goals. Generally speaking, the distributed query processing aims at executing queries considering resource consumption or response time. It attempts to execute the queries in a manner that consumes minimum resources or that takes less time to return an answer. We introduce and discuss this subject in order to understand the ideas behind the solutions developed to overcome the different issues and hence exploit them to materialize the distributed execution model presented in the previous chapter.

The rest of this chapter is organized as follows. In Section 4.1, we introduce the distributed query processing issues. In Section 4.2, we adopt a well known criteria to analyze this subject. For each criterion proposed, we devoted a particular section to explain it. In Section 4.3, we consider the optimization mechanisms. In Section 4.4, we present information about plan cost and catalog. In Section 4.5, we present query processing mechanisms that exploit the network organization and infrastructure. In Section 4.6, we describe caching mechanisms, specifically one called materialized view, and compare it with the replication approach. In Section 4.7, we describe some operators designed exclusively for distributed databases. In Section 4.8, we classify some important distributed database prototypes and particular solutions according to the adopted criteria. Finally, in Section 4.9, we outline our contributions to the **Escada** Project.

4.1 Query Processing Issues

The centralized or distributed SQL processing mechanism can be designed according to the architecture depicted in Figure 4.1 which is described as follows [71, 98, 66]:

Parser In this stage occurs a decomposition of a high-level query language (for instance, SQL). It passes through a lexical, syntactic and semantic analysis. On completion of the process, the high-level language is transformed into some internal representation based on a query graph [77], which is more suitable for the next processing step. In this stage also, the predicate is transformed into a normal form, that is, a conjunctive normal form or a disjunctive normal form. In the former, a sequence of conjuncts are connected with the $\wedge$ operator, where each conjunct contains one or more terms connected with the $\vee$ operator. In the later, a sequence of disjuncts are connected with the $\vee$ operator, where each disjunct contains one or more terms connected with the $\wedge$ operator.

Query Rewriter It aims at optimizing the queries, using strategies that does not take into account

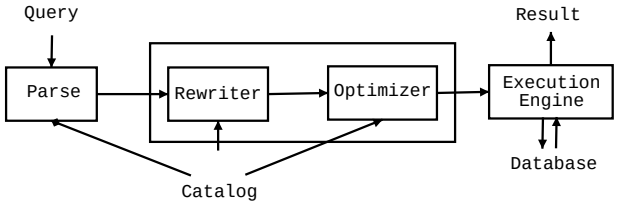


Figure 4.1: Architecture of a Query Processor

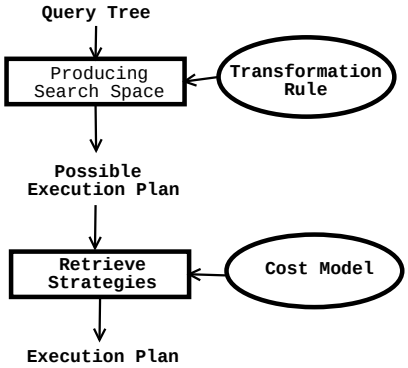


Figure 4.2: Query Optimizer's Ste

where data is stored or located (i.e., it does not consider replicas, cache, resource usage such as processor and network) and physical state of the system (i.e., it does not consider size of tuples, relations and indices). Basically, the optimization applied consists on the simplification of expressions [41], rewrite of subqueries and views [98]. In a distributed system, it also considers the fragments to answer the queries. On completion of the process, the graph is transformed into some internal representation based on the relational algebra tree, which is more suitable for the next processing step. The relational algebra tree (query tree) is produced as follows:

- A leaf node is constructed to each base relation in the query.
- A non-leaf node is created for each intermediate relation produced by the relational algebra operation over a base relation(s) or other intermediate result(s).
- The root of the tree represents the result of the query.
- The sequence of operations is directed from the leaves to the root.

Query Optimizer In this stage, the relational algebra tree is optimized regarding the data localization and physical state of the system. On completion of the optimization process an execution plan is produced (Figure 4.2). The optimizer applies a set of rules to the relational algebra tree and produces possible execution plans. Basically, these rules consist on changing the order of the operation(s) and relation(s) or substituting group of operation(s) for equivalent one(s). To evaluate the cost of a possible execution plan, the optimizer considers a cost model, defining an objective function (e.g., minimize resource consumption or response time) and weights associate to each possible operation.

Code Generation In this stage, the chosen plan is transformed into a low level representation, allowing an efficient evaluation of the expressions and predicates.

Query Execution This stage is responsible for the execution according to the operations defined in the plan.

4.2 Criteria for analysis

We adopt the criteria proposed in [71, 66]. In this thesis though, we group the topics in a different manner and also attempt to evaluate particular solutions such as Microsoft SQL Server, Oracle, DB2 and PostgreSQL. Regardless of the chosen criteria, it is difficult to evaluate and compare query processing solutions. There are many implementations, each with its own subtleties that makes this evaluation a complex process. The approach presented here allows us to visualize the differences and similarities among the centralized and distributed solutions, and offers possibilities to distinguish the critical factors for performance. The organization proposed is the following:

Optimization The optimization techniques search for the execution plan that has an optimal cost in the space of possible execution strategies. The optimizer's job represents the most important and complex issue in centralized or distributed query processing. In fact, the mechanisms presented in the other topics aims at helping the optimizer to produce optimal plans.

Catalog and Cost Model The catalog stores metadata about the database, which describe, for instance, the frequency distribution of the attributes and the existence of indexes. The cost model considers these factors and others (such as network bandwidth consumption, CPU usage) to evaluate possible plans. For that reason, it is important an accurate maintenance of the metadata information. In this topic we present concepts involved in the catalog maintenance and possible cost models.

Network Technologies and Distributed Computing In a distributed environment, the query processing mechanism can exploit the network infrastructure to execute the queries. For instance, the optimizers can exploit broadcast facilities and client's machines in order to distribute processing or to cache information for future use.

Caching and Replication The use of caching mechanisms are common to avoid the access to slow resources. In this topic, we describe the approaches called caching and replication, presenting their differences and similarities.

Operators and Operations In this topic, we present how to extend the operators and operations designed for a centralized environment to be used in a distributed environment. We also present some operators and operations, such as semi-join, designed specifically to exploit the distributed environment's characteristics.

4.3 Optimization

The actual goal of an optimizer is to select an execution plan that is close to the optimal and, perhaps more important, to avoid bad plans. However, it is known that this problem is NP-Hard in the number of relations [52]. For complex queries, searching for the best execution plan can incur in a prohibitive processing time. For that reason, the studies in this area attempt to find techniques that choose with low cost the most efficient plan. In the following, we divide this problem considering the issues involved in how to optimize and when to optimize.

4.3.1 How to optimize

In this section, the intuition behind the optimization techniques is presented, describing the deterministic and randomized classes of algorithms [68]. In spite of this classification, both classes attempt to reduce the costs involved in the search process, considering indeed different strategies. Basically, the difference is a trade-off between processing time and accuracy of the plan.

In the deterministic class, the algorithms based on dynamic programming are the most popular and can be found in various commercial databases [66, 22]. Generally speaking, the algorithms based on dynamic programming solve problems by combining the solutions to subproblems. In our particular case, it evaluates almost all possible strategies in order to find an optimal execution plan, which incurs in a significant processing cost, unfortunately, since the numbers of possible strategies to evaluate can be prohibitive. To reduce the cost when a number of relations is greater than five or six, some heuristics are applied. For instance, it is possible to prune the cartesian operation as a possible step if not actually specified in the SQL statement; or prune initial strategies that are a priori worse than other possibilities [71]. The use of this last approach avoids to evaluate the combination of bad strategies with others, which would be worthless since the produced results would not be optimal.

Other deterministic solution is based on greedy programming [71, 22]. In contrast to the dynamic programming approach, this algorithm works in two steps to reduce the processing cost. First, it selects an initial feasible execution plan based on a simple objective such as “transfer all relations to a single site minimizing the communication cost and process the query”. In this case, the algorithm needs to determine the site that has the maximum amount of information. Second, it attempts to reduce the cost of the initial plan searching for an equivalent solution. Unfortunately, this algorithm can produce a non-optimal initial plan and whatever the effort to improve it will be worthless.

The randomized strategies, such as *Iterative Improvement* and *Simulated Annealing* [66, 97, 53], attempt to search for a good solution close to the optimal, but do not guarantee the best solution, which avoids the high inherent costs of the deterministic approaches. It is characterized by two steps similar to the greedy programming. In fact, the first step defines one or more initial plans using a greedy approach. The second step attempts to improve the initial plans applying random transformations. For instance, a random transformation could be the exchanging of two randomly chosen relations.

4.3.2 When to optimize

Other factor that categorizes the optimization process is the moment when it is applied. It is possible to optimize before the execution, which leads to static optimization [85, 15], or during execution, which leads to dynamic optimization [40, 20]. In the static approach, the main goal is to avoid the optimization cost each time a query is processed. For that reason, the optimization is applied before execution and the chosen plan is cached for future use. It is a suitable approach for the deterministic algorithms, since an optimal plan can be selected at once. The approach relies on statistic information to choose the optimal plan and can select sub-optimal strategies when the statistics are inaccurate [15]. For a discussion about how statistics are used, collected and maintained see Section 4.4. With respect to the dynamic optimization, the statistics are considered to define the first operations and the following decisions are based on the results being obtained. The dynamic approach is a more expensive solution due to the need of constantly process the same queries. It is better suited for processing ad-hoc queries.

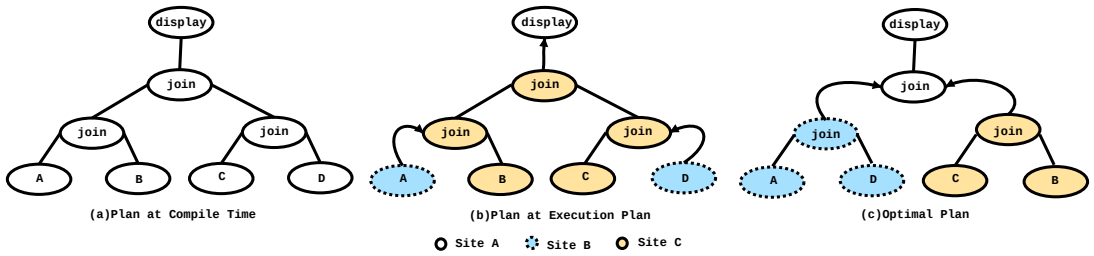


Figure 4.3: Two-Step Optimization and the Communication Problem

Hybrid approaches attempt to exploit the advantages of both worlds. For instance, there are strategies that optimize the query before its execution and compare the estimated costs of each operation with the real ones after the execution. In case of a large difference, the old plan is discarded and the optimizer selects a new plan from the point where the problem was detected. In this way, it adjusts the plan to the new environment's conditions and constraints [103].

In distributed environments, to avoid the high cost associated with the dynamic optimization strategies, the "two-step optimization" [14] approach represents an alternative. In the first step, it selects an execution plan for a given query and stores its choice in cache for future use. In the second step, whenever the query is sent to execution, it accesses the cache to retrieve the plan and carries out certain decisions before execution. Several variations of this solution can be found in the literature [14, 93], but a simple algorithm that captures its ideas is presented in [66] as follows: (*i*) at compile time, the optimization is accomplished selecting a plan suitable to be executed in a centralized environment; (*ii*) on every execution, a selection of sites is performed considering information such as network bandwidth, processor load average and available memory. Each step can further be optimized using any suitable technique.

Nevertheless, this approach also has its weakness and problems. It ignores the communication issues when selecting its plan in the step (*i*), which can lead to use sub-optimal plans with high communication cost, as it is possible to see in Figure 4.3 [66]. The plan (*a*) represents the optimization at compile time; the plan (*b*) represents the optimization regarding the site(s) selection; the plan (*c*) is the optimal solution. Basically, it is not taking into account that relations A and D are collocated at the same site and relations B and C are collocated at another site. The high communication cost is related to the join ordering specified by the first step, which leads to transfer the relations A and D to other site in order to process the plan. In contrast, if the optimal plan could be chosen this expensive communication would be avoided, incurring in a less expensive final transmission of the result of join A and D.

4.4 Catalog and Cost Model

4.4.1 Catalog

The catalog is essential in all phases of the query processing. For instance, the parser identifies the relations, attributes referenced in SQL statements and their properties, accessing the catalog. The rewriter applies some transformations to the SQL statements[1] and needs to access the catalog to determine which transformations must be applied and how. In the optimization phase, information about replicas and statistics about the database are retrieved from the catalog and used

[1]It is important to notice that the optimizer in this phase manipulates an internal representation of the SQL statement issued. However, this distinction is not relevant here and the term SQL statement is referenced instead, for simplicity.

to compute the costs involved in each operation. Hence, observing these aspects, we have sufficient and important reasons to consider its role essential and fundamental to the query processing mechanisms.

In a first look, the technologies used to maintain the catalog and its associated information appear to be simple. Unfortunately, even in a centralized environment, there are challenges, such as statistics maintenance that is quite important to produce optimal plans and whose goal must balance precision and resources consumption. We discuss this issue in detail in the next section. In a distributed environment, there are additional challenges, such as catalog organization and replica management [28], that need to be taken into account.

The catalog organization in a distributed environment can be implemented in a central site using an approach similar to a centralized environment. However, this solution introduces a single point of failure and does not scale up. To circumvent these problems, a replication mechanism could be used, but a full replication could incur in prohibitive overhead to guarantee consistency among the replicas. According to [28], an intermediate solution seems to be the best proposal and to locate the fragments and replicas it discuss the use of location dependent or independent identifiers. In the first solution, there are server addresses associated to the identifiers expressing where the information is located. It is a simple solution that unfortunately cannot efficiently exploit migration and replication without a high cost to manage the server addresses. In contrast to it, location-independent identifiers can be used to circumvent this problem. Generally speaking, this solution implements distributed indexes which map each identifier to a server list where the information is located. To reduce the communication cost, the approach tries to put the indexes near to the information.

4.4.2 Plan Cost

In order to correctly evaluate the execution strategies, the cost model must be accurate, and therefore, must consider the environment's current state to accomplish its goal. Basically, the cost model can be categorized according to its objective function as total time [72] or response time [35]. The total time model, also know as classical model, tries to minimize the total resource consumption increasing the overall throughput. Inside the total time classification, however, it is also possible to see some specializations. For example, some solutions aim at minimizing the total communication cost. The response time model tries to minimize the execution time exploiting parallelism, which can lead to a higher resource consumption when compared to the former approach.

In a centralized system, CPU, I/O and memory consumption are possible factors that are used as cost components. However, in a distributed database system, it is important to consider the communication factor. In some approaches, the communication factor is responsible for the biggest impact on the cost, and in others, for the simplification of the cost model, just the communication factor is considered [71].

The cardinality of base and intermediate relations are essential to compute the cost related to each factor considered. The statistics stored in the catalog are used to retrieve or estimate each cost. The following data is available into the catalog: number of tuples in each relation, attribute size, tuple size and number of unique values per attribute. Usually, it is stored as histograms on the frequency distribution of attributes. To avoid the selection of bad strategies, each cost must be obtained with a high level of precision relying on accurate and updated statistics to accomplish this. On the other hand, to maintain the statistics there is a management overhead, which leads to an establishment of a trade-off between acceptable precision and overhead [71, 37].

Cost models based on economic paradigms are frequent in several area of research. The use of economic paradigms to model problems in distributed computing has been studied since the mid-1980s (e.g, economic models for resource allocation, load balancing, flow control and quality of service) [31]. The motivation to use an economic model is that distributed systems are too complex to be controlled by a single centralized component with a universal cost model [66]. In the distributed databases, the first to be based on an economic model was the Mariposa [93].

4.5 Network Technologies and Distributed Computing

The network organization and its associated technologies have a direct impact on the optimization approaches. The first distributed database systems considered the communication factor as preponderant to define cost models [71]. Even though this assumption absolutely simplified the optimization problems, currently it has to be re-evaluated on the light of existing network technologies. Nowadays, communication time in LAN environments (e.g., ATM and GigabitEthernet) is comparable to I/O time [38, 84], which increases the weight of other factors such as CPU, memory and I/O to compute the objective functions. In WAN environments, the assumption is still valid and it is possible to consider the communication factor as preponderant.

There are optimization approaches that exploit specifics characteristics of the network [71]. It is possible to find studies on the use of broadcast primitives to increase the number of parallel operations, or algorithms designed specifically to star [64] and satellite [51] networks.

There are studies that attempt to use the client resources, based on the fact that the number of clients is larger than the number of servers. They propose to transfer some activities to be executed at the clients. For example, some approaches execute the initial steps of the query processing at the clients: parse, rewrite and optimization steps. Other approaches go beyond these simple activities, sending the data to be processed at the client (data shipping), instead of using the traditional paradigm where the queries are sent to the server to be processed (query shipping). Some important information about the data shipping effort [66] is outlined as follows:[2]

- This approach can scale better than other solutions, since it exploits the client resources.
- Sending the data to be processed at the client might increase the communication cost. To reduce it, it is possible to cache[3] the shipped data at the client, but this solution brings another problem related to coherence management of the information stored at the client's cache.
- It increases the complexity of the optimizers, since there are new ways to process the queries.
- The use of hybrid solutions can take advantage of the benefits of both approaches. For example, it is possible to execute transactions that do not update large amount of data at clients, propagating the updates in batch operations to the server. This solution reduces the communication cost, and in case of rollback, it discards the transaction without affecting the server activities [13].

[2] In [34, 33] a detailed analysis about this subject can be found. Some important performance considerations are discussed and is recommended the use of hybrid solutions to exploit the advantages of both approaches.

[3] The caching mechanism is presented in detail in Section 4.6.

4.6 Caching and Replication

4.6.1 Materialized views

Caching mechanisms exploit the principle of locality, minimizing access to slow resources (e.g., use the disk to avoid network access, or memory to avoid disk access). Earlier optimization techniques cached information such as base relations, indexes or part of them [66]. In addition, some studies suggest that caching of intermediate results can also dramatically increase the performance [27, 48]. This approach is called materialized view and some commercial products such as Oracle [7, 25] and Microsoft SQL Server [39, 79] use it.

Materialized views can be used to avoid the overhead of computing complex queries. Instead of processing the query when the view is referenced, the results are automatically retrieved from the cache, using disk to avoid network communication or main memory to avoid I/O access, according to the amount of resources necessary to store the information. Besides, it is also possible to substitute portions of a query using the view, whenever the optimizer identifies a chance to reduce computation cost. On the other hand, this improvement in performance comes with the cost of keeping cached information up to date. See Chapter 5 for a detailed discussion about materialized views and their issues.

4.6.2 Replication

Replication also exploits locality and, at the same time, increases resiliency. The differences between replication and caching (not only materialized views, but cache in a broader sense) can be presented as follows [66]:

	replication	caching
placement	server	client, server or intermediate layer
granularity	coarse	fine
storage device	usually disk	usually main memory
impact on catalog	yes	no
update protocol	propagation	invalidation
remove copy	explicit	implicit
mechanism	separate fetch	keep copy after use

Table 4.1: Differences between Replication and Caching

placement The replication technique deals with relations that need to be stored on the server. On other hand, the caching technique manipulates subsets of relations or result sets of queries that can be stored anywhere.

granularity The relation is the grain of the replication whereas caching deals with sets of tuples.

storage device In consequence of the coarse grain used in replication, information is stored on disk. In contrast, caching usually stores information in memory. Nevertheless, this decision is highly influenced by the amount of memory necessary to store the information, and even caching can use the disk as a storage area (e.g., materialized views).

impact on catalog When a replica is created or reallocated, this information must be updated in the Catalog to allow the clients to find the replicas. Caching manipulates information that is not shared. It is something particular to the place where it is stored.

update protocol When information is updated, this new data must be propagated to all replicas. In a caching mechanism, it is usually invalidated. The materialized view represents an exception because its state is kept up to date, which is a consequence of its principle: “avoid recomputation”.

remove copy The “copied information” is removed from the cache implicitly whereas, using replication, the remotion depends on the protocol designed.

mechanism To access the replicated information, it is necessary to refer to the replica explicitly or implicitly. The use of the information in cache is consequence of successively operations instead.

Basically, these differences between replication and caching are related to a simple fact: “replication deals with long duration information, large number of clients and a pattern access that do not change frequently [66]. Observing such an information, it is simple to realize the reason for what materialized views do not perfectly fit in the caching classification.

For more information about database replication see Chapter 3.

4.7 Operators and Operations

Several techniques available in a centralized database to implement relational algebra operators [80, 41] can be used in distributed query processing, being enough to augment them with basic communication primitives. However, there are special techniques developed to circumvent specific problems (e.g., communication cost) of a distributed environment [66, 91].

Semi-join is the name of an important technique developed for a distributed environment [10]. It proposes to implement the join operation among relations in different sites. In a simple manner, the traditional join applied to a distributed environment would lead to transferring one of the relations from one site to the other. This transfer unfortunately could generate unnecessary communication costs, since some tuples could be discarded in the join. The semi-join tries to avoid this unnecessary network bandwidth consumption. Basically, it sends only the attributes that are necessary and sufficient to compute the join and after that it retrieves only the tuples that match. On the other hand, it is important to notice that this approach uses two communication steps and two CPU steps against one communication step and one CPU step in the traditional join. In situations that the relation to be transfered is large and the selectivity of the join is good, this solution is better.

Experimental research shows that semi-joins are not suited to centralized environments [70, 72]. However, recently, studies show it can be appropriate for join involving large relations in which the tuples are also large [91].

Other technique, called **Pipeline** [55], allows that the first results are delivered as soon as possible, exploiting the parallelism and reducing the response time. The pipelining mechanisms are sometimes also know as on-the-fly processing. Roughly speaking, instead of waiting to process an operation whose results are stored on an intermediate place, it starts in parallel the next step of a relation algebra tree. The drawback with pipelining is that the inputs to operations are not necessarily available all at once for processing.

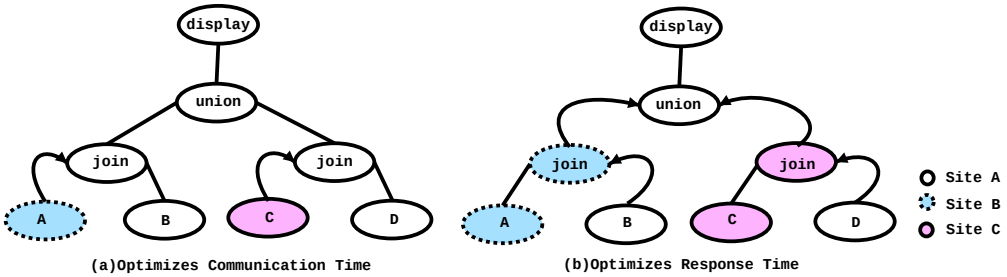


Figure 4.4: Ingres Objective Function

4.8 Classification of Database Systems

The interest in distributed query processing is evident and has been increased over the years. Observing the number of studies that discuss its different aspects, such as optimization, replication and materialized views, it is simple to realize its importance. Other evidence is the increasing number of prototypes designed to test the ideas and concepts developed.

It would be almost impractical to introduce an uniform classification of all available systems according to the criteria defined here. Therefore, we restrict our presentation to relational database systems. Specifically, we show and classify the first efforts in building distributed relational database systems: INGRES, System R* and ADD-1. After that, we also show the Mariposa system that was responsible for providing an economic paradigm for query processing and data migration. Finally, some commercial products, such as Oracle, Microsoft SQL Server, DB2 and PostgreSQL are presented. We propose here to present the classification of some components presented in these products that are important for the query processing. However, it is important to notice that there are no commercial distributed relational database products available.[4] Our classification is presented as follows.

- **In the Distributed INGRES [94, 71]:**

  Optimization It uses a deterministic dynamic optimization characterized by a limited search of the solution space, where a decision is evaluate for each step without considering its impact and consequences on the global optimization. In contrast to other approaches presented here such as dynamic programming, it does not evaluate all the possibilities to find the best one. It attempts to reduce the natural overhead imposed by the dynamic optimizations making it a feasible solution.

  Catalog and Cost Model Its objective function attempts to minimize a combination of both communication time and response time. However, these objectives can be conflicting and the optimization algorithm favors one of them. For example, in Figure 4.4, the first execution plan is adjusted to reduce the communication time, since all the operations are executed in site A. In the second execution plan, the response time is favored and a parallel execution plan (e.g., independent or pipeline parallelism) is chosen. Each join is processed in parallel in a different site and the result is gathered in site A. It is simple to see, that this last approach increases the communication time and reduces the response time. Furthermore, statistics provide information about the cardinality of the base relations which are used in the initial steps of the optimiza-

[4] From the best of our knowledge, there was an attempt by Choera, according to [66], to distribute a Mariposa commercial version. Nevertheless, during the writing process, there was no information about the product in [19].

tion process. Its "on-line" approach allows it to use real information to evaluate the intermediate steps.

Network Technologies and Distributed Computing It exploits broadcast to replicate information and to maximize the degree of parallelism.

Caching and Replication It implements replication and horizontal fragmentation.

Operators and Operations There is no semi-joins. The relations are transfered among sites, whenever an execution plan must join relations that are in different sites.

- **In the System R* [72]:**

Optimization It uses a deterministic static optimization characterized by an exhaustive search among all the alternatives. To reduce the number of possibilities in the search space, making a feasible solution, it applies dynamic programming and heuristics.

Catalog and Cost Model Its objective function attempts to minimize a total cost improving the overall throughput and performance. It considers local processing time (I/O and CPU) and communication time (number of messages and message size) to accomplish this. Its static approach implies estimation of the cardinality of the intermediate results to produce near optimal plans. For that reason, it keeps statistic information about the cardinality of base relations and the number of unique values per attribute.

Network Technologies and Distributed Computing It does not exploit aspects of the network technologies and distributed computing.

Caching and Replication It does not implement replication nor fragmentation. Although the algorithm described in [85] deals with fragmentation.

Operators and Operations There is no semi-joins or pipelined hash joins. Hence, the algorithm extends traditional join operators using send and receive. It decides based on estimations which result to transfer to each site.

- **In SDD-1 [10]:**

Optimization It implements a gready approach (deterministic and static) in which the initial plan is defined based on a global optimization that it is iteratively improved with an algorithm called "hill-climbing". Roughly speaking, its first plan is established computing the intersite communication cost to schedule all operations on a single site. It is important to notice, however, that the cost of transferring the result to the final site is ignored. Hence, after that, it attempts to improve the initial plan allocating operations to be executed in others sites and gathering the results in the chosen one.

Catalog and Cost Model Its objective function attempts to minimize the total communication time. It does not take into account local processing time (I/O and CPU) and response time to accomplish this. It keeps statistic information about the base relations such as cardinality, the number of unique values per attribute, join selectivity factor, size of projection on each join attribute, attribute size and tuple size.

Network Technologies and Distributed Computing It does not exploit aspects of the network technologies and distributed computing.

Caching and Replication It does not implement replication nor fragmentation.

Operators and Operations It does use semi-joins in order to reduce the total communication time.

- **In Mariposa [93]:**

Optimization It implements an approach based on the two-step optimization. In the first step, it uses the local PostgreSQL[5] optimizer, which produces a plan without considering the fragments. Then a module called fragmenter is responsible to change the plan to reflect the data fragmentation. In the second step, the microeconomic solution takes place deciding where each piece of the fragmented plan will execute.

Catalog and Cost Model It argues that global optimization strategies using cost-based approaches cannot perform well in WAN. For example, they do not scale up to a large number of possible processing sites nor adapt well to different access constraints (e.g., workload limits and patterns). For those reasons, it proposes a microeconomic paradigm for query and storage optimization. Furthermore, instead of using a centralized catalog to gather statistics about the sites, it implements a distributed advertising service to announce sites that can be used to evaluate queries. It also maintains the traditional statics in each site in order to evaluate its local execution cost.

Network Technologies and Distributed Computing It does not exploit aspects of the network organization, but its algorithm is designed to work in WAN environments.

Caching and Replication It allows to create horizontal fragments using four partition modes: random, round-robing, key-based and hash-based. Every new relation can be split into two fragments, which can be split into two and so one. In the random mode, the tuples are placed in one fragment or the other at random. In the round-robin, it provides an equal allocation of tuples among the relations. In the key-based, the tuples are split based on attribute value. In the hash-based, a function over an attribute and a value define the fragment to place the tuples. Instead of providing a static allocation, the fragments can be bought, sold, split and coalesced according to the access pattern. It implements an asynchronous replication where the replicas periodically receive updates. There are two types of Mariposa replicas: (*i*) a read-only that receives its update from its parent site; (*ii*) a read-write that allows the children to receives updates and propagate it to the parent.

Operators and Operations It does use semi-joins in order to reduce the total communication time, but can parallelize activities to improve performance.

- **Microsoft SQL Server [78]:**

Optimization It implements a static optimization approach based on dynamic programing. It works in multiple phases. First, it produces a simple reasonable plan that satisfies the query. If the plan takes less time than a specified threshold, it stops its work. Otherwise, it continues to search for a plan that takes less time than the specified threshold. In this way, its mechanism avoids unnecessary work to produce optimal plans for queries whose performance is not critical.

Catalog and Cost Model Its cost model is computed based on statistics managed in a central catalog. It builds and maintains statistics on relations in order to estimate distribution of the attributes and its selectivity [37]. To accomplish this, it uses histograms on the frequency distribution of the attributes. Periodically, it updates the histograms using a complex sampling method which provides the necessary accuracy and minimizes impact on transaction throughput [24]. Basically, this method samples random pages where the number of pages is defined according to a minimum number of tuples. The frequency of the updates is determined by the volume of data in the relation and the amount of changing data.

[5]The single database engine distributed with Mariposa was POSTGRES: a pre-alpha release of POSTGRES95 which has the basis of the current development.

Network Technologies and Distributed Computing It does not exploit aspects of the network technologies and distributed computing.

Caching and Replication It implements materialized views know as index views in SQL Server 2000 [39, 79]. In order to transform a simple view in a materialized view, some constraints must be followed. First, the view must be defined by a single-level SQL statement containing selections, inner joins and aggregations. Second, the "from" must reference just base relations. Third, if a "group by" clause is used, all attributes that appear on the aggregation must be referenced in the "select" clause and the aggregation functions are limited to "sum" and "count". The objective of all these restrictions is to allow that the view can be updated incrementally. Using this approach, it can increase the performance of queries that reference the view, since its result set do not need to be computed in runtime. Furthermore, it can use the view to substitute parts of queries if possible. See Chapter 5 for a detailed discussion about this subject. In regarding the replication, it implements the traditional approaches used in commercial solutions. See Chapter 3 for more information.

Operators and Operations According to [78], it can use semi-joins to compute join operations that deals with huge amount of data. It also provides intra-query parallelism allowing to schedule portions of queries to be executed on multiple processors in SMP computers.

- **Oracle [25]:**

Optimization It implements a static optimization approach based on dynamic programing. Its optimizer estimates the cost for various alternative strategies and chooses the one with the lowest cost.

Catalog and Cost Model Its cost model is defined according to the database administrator's preference. It is possible to establish a model whose goal is to minimize the time to return the first row or first set of N rows. Or to establish a model whose goal is to return the result set in the least amount of time. To accomplish its objectives, it maintains a set of statistics that can be classified as object, system and user-defined statistics. Its object statistics are similar to the others commercial databases as Microsoft SQL Server and DB2. Their principles are the same, regardless of the subtle differences. Otherwise, the system statistics allow to consider the CPU cost and I/O cost based on the machine's performance, instead of relying upon a fixed formula to combine them. The user-defined statistics allow customer to extend the functions and information used to compute the costs.

Network Technologies and Distributed Computing It does not exploit aspects of the network technologies and distributed computing.

Caching and Replication It implements materialized views [7, 25]. It can increase the performance of queries that reference the view, since the result set do not need to be computed in runtime. Furthermore, it can use the views to substitute parts of queries if possible, having a low cost. It implements the traditional replication approaches used in commercial solutions.

Operators and Operations According to [25], it uses semi-joins to compute join operations that deals with huge amount of data. It also provides intra-query parallelism allowing to schedule portions of queries to be executed on multiple processors or different nodes. It is also important to notice that the optimizer actually takes into account the impact of parallel execution when choosing the best plan.

- **DB2 [23]:**

  Optimization It is simple to see that it implements a static optimization approach based on dynamic programing. Its optimizer estimates the cost for various alternative strategies and chooses the one with the lowest cost.

  Catalog and Cost Model Similar to Oracle's approach, its cost model can be defined based on the application's characteristics. It is possible to establish a model whose goal is to minimize the time to return the first set of N rows. Or to establish a model whose goal is to return the result set in the least amount of time. To accomplish its objectives, it maintains a set of statistics that are similar to the others.

  Network Technologies and Distributed Computing It does not exploit aspects of the network technologies and distributed computing.

  Caching and Replication It also has a materialized view implementation that is known as summary table. The optimizer can use the information when requested or when it detects that it would profit from the use of a precomputed value. It also implements the traditional replication approaches used in commercial solutions.

  Operators and Operations It does not use semi-joins. It provides intra-query parallelism allowing to schedule portions of queries to be executed on multiple processors or different nodes.

- **PostgreSQL [44]:**

  Optimization It implements a static optimization based on dynamic programming. Its optimizer estimates the cost for various alternatives strategies and chooses the one with the lowest cost. However, in order to avoid the inherited overhead related to the exhaustive searches, it has an option that enables a genetic query optimizer (GEQO). Basically, it is a "random" solution adapted from D. Whitley's Genitor algorithm [36], that supports large join queries effectively through non-exhaustive search.

  Catalog and Cost Model Despite the difference regarding the GEQO, it applies a central catalog and histograms to support the optimization. It is also possible to configure CPU and disk performance according to the machine used, in manner similar to the oracle approach. However, instead of being an automatic process, we need to do it manually.

  Network Technologies and Distributed Computing It does not exploit aspects of the network technologies and distributed computing.

  Caching and Replication Its current release 7.4.x does not implement materialized views. Support for single master/multi slave asynchronous replication was recently developed. Regardlessly of the official development, there is also a replication solution based on group communication (Postgres-R) developed by [60] in release 6.4.2. Further, nowadays there are efforts to extend the traditional replication solutions and to incorporate Postgres-R in the current release.

  Operators and Operations It does not use semi-joins nor intra-query parallelism.

4.9 Distributed Query Processing in the Escada

In this section, we present the distributed query processing mechanisms designed for the **Escada**. The approaches and decisions were somehow constrained by the work's timeframe. We also took

into account the possibilities of easily integrating the mechanisms in any database system. Generally speaking, we propose a two-step optimization, a fully replicated catalog and simple extensions to the SQL grammar which, for instance, provides the necessary commands to configure the replication. We also exploit the atomic multicast to invalidate or update a distributed semantic caching.

Our approaches are roughly presented as follows (see Chapter 5 and Chapter 6 for a detailed discussion):

Optimization We use a two-step optimization to process the queries. This mechanism is a good choice when we intend to augment an available centralized database in order to provide a distributed query processing without incurring in much effort or lots of changes. Thus, we can preserve the normal optimization mechanisms available in the database system. In the first step, everything is done like in a centralized execution until the queries arrive at the execution engine. In the second step, before the execution, the execution engine must decide in which site each node of the query tree will be executed. Initially, we propose to choose the first possible and available site, which means any correct site that has the referenced fragments in the node. Regardless of this decision, it is possible to construct the second step based on any optimization policies and mechanisms without much impact on the current implementation of the database.

Catalog and Cost Model We preserve the available mechanisms used for the cost model in the case of local fragments and augment it to deal with remote fragments. Each site is responsible for gathering its statistics and optimize the queries without even known about the distributed execution. In the case of queries accessing fragments that are not locally available, we must resort to extensions of the catalog which gathers information from all the replicas. This is done as follows. Each site is responsible for computing its statistic for the local fragments. Periodically, the result of the computation is broadcast to the other sites. The catalog is fully replicated and to reduce the overhead of the full replication, we rely on lazy replication for the majority of the updates. However, some operations such as the creation, deletion or modification of the object structures are applied synchronously which avoids inconsistency among the replicas. Finally, this process can take advantage of the atomic multicast used to serialize the transactions, avoiding the overhead of another communication step.

Network Technologies and Distributed Computing We do not directly exploit the network technologies to process the queries. However, all the extensions proposed here rely on the multicast primitives used in the PDBSM. Furthermore, regarding that each site computes its statistic and afterwards exchanges the result of the computation and that we also propose a distributed cache management, we can consider that our approach is based on distributed computing mechanisms.

Caching and Replication We exploit the multicast primitives in order to build a distributed cache management. Basically, we take advantage of the atomic multicast that is used to serialize the transactions to update or invalidate distributed cache entries. The entries in the cache are result of the distributed execution and are used to reduce the overhead involved in the remote communication. It is important to notice that is not build to reduce the local access to storage resources, rather than that it is used to circumvent communication problems such as latency and bandwidth consumption. However, instead of using tuples or pages as cache entries, our approach is based on semantic. In other words, the entries in the cache are identified using the predicates available in the queries.

Operators and Operations We do not use semi-joins nor intra-query parallelism. The implementation of the these functionalities would require several modifications to the target database. For that reason, we do not provide them.

Chapter 5

Semantic Caching

The optimistic execution and the atomic broadcast can be seen as the fundamental characteristics of the replication approach of the **Escada**. Our proposal is to exploit these properties in order to build a semantic caching and improve the certification process.

Specifically, the first mechanism intends to develop a distributed cache management, taking advantage of the broadcast of the transactions to update or invalidate the entries in cache. In this case, the **Escada** permits the development of this distributed cache without much effort and without the negative impact of a centralized management. However, instead of managing the cache using tuples or pages, we propose a semantic approach [27, 50]. Basically, in this approach, the entries in the cache are identified using the predicates involved in the queries. Doing this, we avoid the management overhead of the tuples, which usually involves retrieval, update and replacement per tuple. In contrast to page caching, we also reduce the management overhead and further overcome the problem of space consumption, which is a consequence of the page fixed size, while disregarding the size of the result set and always allocating pages. Furthermore, using the semantic approach, it is possible to build better replacement algorithms that assign values in fair and lightweight models; and parallelize the retrieval of local information available in the cache with the request of missing tuples from remote sites.

The second mechanism intends to reduce the number of aborts in the certification process. In order to do that, it tries to avoid false conflicts as a result of coarse grains, in this case relations, which arise when instead of using the accessed tuples to compose the transaction's outcome, we resort to the relation(s) evicting hence to flood the network and making the **Escada** protocols feasible.

The rest o this chapter is organized as follows. In Section 5.1, we present the satisfiability and implication problems in database systems. In Section 5.2, we present the ideas behind semantic cache and establish a comparison among other approaches of caching. We describe how to detect that previous queries can be used to improve performance. We also define how the updates invalidate the entries in cache and how to recompute some class of queries using just information available in cache and in the updated tuples. Finally, in Section 5.3, we present a discussion about our approach comparing it with related works and outline a possible extension to the **Escada** which is currently available as a prototype [95].

5.1 Satisfiability and Implication Problems

The theory behind problems involving satisfiability and implication provides us background and basis to important issues in database research. For instance, it is possible to determine the distributed horizontal fragments that must be returned to answer a request; or to find materialized views that can improve the overall performance. Formally, these problems can be defined as follows [47]:

Expressions Let *S* and *T* be expressions based on the relational calculus except that both do not have quantifiers.

Satisfiability/Contradiction S is satisfiable if and only if there exists at least one assignment for S that satisfies S. Otherwise, there is a contradiction in S.

Implication - *S implies T*, denoted as $S \rightarrow T$, if an only if every assignment that satisfies S also satisfies T.

Equivalence - S and T are equivalent if and only if S implies T and T implies S, denoted as $S \leftrightarrow T$.

In spite of being important for database researches and even commercial products [7, 25, 39, 79], generic algorithms to solve these problems are known to be NP-Hard [47]. For that reason, we restrict the class of problems to achieve solutions in acceptable time. We consider formulas connected through conjunction, variables and constants defined over either the integer or real domain, and the following comparison operators $(\leq, \geq, =, <, >)$.

We present an algorithm to detected satisfiability, [1] based on [47]:

- **Step 1:** Perform a transformation on each $(S_i.a_1\ \theta\ S_j.a_2)$ and $(S_i.a_1\ \theta\ C)$ in S such that only $(S_i.a_1\ [<\ |\ \leq]\ S_j.a_2)$ and $(S_i.a_1\ [<\ |\ >\ |\ \leq\ |\ \geq]\ C)$ remain. If $(S_i.a_1 \neq C)$ or $(S_i.a_1 \neq S_j.a_2)$ is found, report NP-Hard problem and exit. At the same time, eliminate the trivial formulas, namely, $(S_i.a_1\ \theta\ S_i.a_1)$ and $(C_1\ \theta\ C_2)$ in S. If $(S_i.a_1\ [<|>\ |\ \neq\ ]\ S_i.a_1)$, or $(C\ [<|>\ |\ \neq\ ]\ C)$, or $(C_1\ [<|\leq\ |\ =\ ]\ C_2)$ with $(C_1 > C_2)$, or $(C_1\ [>|\geq\ |\ =\ ]\ C_2)$ with $(C_1 < C_2)$, then report that S is unsatisfiable, and exit.
- **Step 2:** Construct the minimum range $[C_{low}^{S_i.a_1}, C_{up}^{S_i.a_1}]$ for each $S_i.a_1$ by scanning all $(S_i.a_1\ [<\ |\ >\ |\ \leq\ |\ \geq]\ C)$.
- **Step 3:** Construct the labeled direct graph G_S; detect all $SCCs$. If any "<" is found in any SCC, then S is unsatisfiable; exit. Otherwise, collapse $SCCs$ and obtain an acyclic graph $G_{S_{collapsed}}$.
- **Step 4:** Topologically sort all nodes of the graph to compute the "real" minimum ranges $[A_{low}^{S_i.a_1}, A_{up}^{S_i.a_1}]$.
- **Step 5:** If any $C_{low}^{S_i.a_1} > C_{up}^{S_i.a_1}$ or $C_{low}^{S_i.a_1} = C_{up}^{S_i.a_1}$ with $C_{low}^{S_i.a_1}$ or $C_{up}^{S_i.a_1}$ open, S is unsatisfiable. Otherwise, report that S is satisfiable.

In **Step 1**, we are considering that the expressions are in a normal form, which means that there is neither parenthesis nor negations. This assumption is valid since we are planning to use

[1] For a detailed discussion about graphs structures, topological sort and strongly connected components see [22].

this algorithm after the optimizer which normalizes the predicate. The transformations for $(S_i.a_1 \ \theta \ S_j.a_2)$ exploit the fact that $(S_i.a_1 = S_j.a_2) \equiv (S_i.a_1 \leq S_j.a_2) \wedge (S_i.a_1 \geq S_j.a_2)$ and the possibility of reordering expressions putting $S_i.a_1$ before $S_j.a_2$ and vice-versa. The transformations for $(S_i.a_1 \ \theta \ S_j.a_2)$ exploit the fact that $(S_i.a_1 = C) \equiv (S_i.a_1 \leq C) \wedge (S_i.a_1 \geq C)$. In case of C belonging to the real domain the expressions are reduced to $(S_i.a_1 \ [< \mid > \mid \leq \mid \geq] \ C)$. In case of the integer domain, we can further reduce the operators to $(S_i.a_1 \ [\leq \mid \leq] \ C)$ because $(S_i.a_1 < C)$ is equivalent to $(S_i.a_1 \leq C\text{-}1)$ and $(S_i.a_1 > C)$ is equivalent to $(S_i.a_1 \geq C + 1)$. Finally, the step tries early to decide that S is unsatisfiable based on trivial approaches outlined before.

In **Step 2**, we associate a minimum range for each $S_i.a_1$ by scanning all $(S_i.a_1 \ [< \mid > \mid \leq \mid \geq] \ C)$. When the comparison operator is either $<$ or $>$ the correspondent bound is classified as open, otherwise it is closed.[2] It is important to notice that when C belongs to the integer domain, it is always classified as closed because $(S_i.a_1 < C)$ is equivalent to $(S_i.a_1 \leq C\text{-}1)$ and $(S_i.a_1 > C)$ is equivalent to $(S_i.a_1 \geq C + 1)$.

In **Step 3**, we build a labeled directed graph $G_S = (V_S, E_S)$ for S. Each node has a direct correspondence to a distinct variable $S_i.a_1$ in S and each edge is labeled with $<$ or $\leq$. Still in this step, all SCCs (Strongly Connected Components), which means nodes that are reachable via paths from each other, are detected and that implies $S_i.a_1 = S_j.a_2$. However, if during this process an edge labeled with "<" is found, S is unsatisfiable because $S_i.a_1$ will not be equal to $S_j.a_2$. Otherwise, the SCCs must be collapsed to obtain an acyclic graph that will be used in the next phase.

In **Step 4**, using the acyclic graph, we topologically sort all nodes to compute the "real" minimum ranges $[A_{low}^{S_i.a_1}, A_{up}^{S_i.a_1}]$, which is defined as follows. For all $S_i.a_1 \ [< \mid \leq] \ C_i$ and $S_i.a_1 \ [> \mid \geq] \ C'_i$, we define $[C_{low}^{S_i.a_1}, C_{up}^{S_i.a_1}]$, where $C_{up}^{S_i.a_1} = min(C_i)$, $C_{low}^{S_i.a_1} = max(C'_i)$ and define if the brackets are closed or open according to the comparison operators. Furthermore, we must consider the relations established among different variables. For such, we need to traverse the topologically sorted graph to redefine $[C_{low}^{S_i.a_1}, C_{up}^{S_i.a_1}]$, producing what is called real minimum ranges.

In **Step 5**, we decide whether S is unsatisfiable or not. Basically, S is unsatisfiable if any $C_{low}^{S_i.a_1} > C_{up}^{S_i.a_1}$ or $C_{low}^{S_i.a_1} = C_{up}^{S_i.a_1}$ with $C_{low}^{S_i.a_1}$ or $C_{up}^{S_i.a_1}$ are open.

We just presented the satisfiability algorithm, since the implication problem can be evaluated as follows: $(S \rightarrow T) \leftrightarrow (S \wedge \neg T)$. In [47] is presented a detailed analysis of the algorithm outlined here and specific algorithms to solve implication.

5.2 Semantic Cache

The semantic cache is designed to avoid recomputation of queries based on previous requests. However, instead of organizing the information using pages or tuples, it uses predicates available in SQL statements to do that.

Shaul Dar et al. in [27] establish a comparison among semantic cache, page and tuple approaches using three factors: data granularity, request and cache replacement policies. In page caching architectures, the unit of transfer between the servers and clients is the page. The ability to cluster tuples in pages simulating a database organization can be a good idea when compared to approaches that transfer one tuple at time. For instance, it has a positive impact on the bandwidth consumption, reducing the communication steps. Unfortunately, this leads to a false sharing prob-

[2] This information is usually indicated changing the brackets from [to] for $C_{low}^{S_i.a_1}$ and from] to [for $C_{up}^{S_i.a_1}$. However, to a better presentation of the algorithm we omit this fact considering always closed brackets.

lem, which has noticeable impact on situations that a single tuple is requested and a whole page is returned. In tuple caching architectures, the overhead to transfer one tuple at time is inconceivable. To avoid this, the architecture whenever possible groups the tuples to transfer. Regardless of the effort to reduce its high cost, it implies a management overhead that is proportional to the number of tuples, which is reflected on entries in cache and its associated information. In contrast, the semantic approach allows to group the tuples according to the issued queries. For that reason, the impact on bandwidth and also on cache management is reduced. It also provides the alternative to group entries in cache, further reducing the overhead involved in management.

The cache must somehow identify which information is locally available and which is not. Page and tuple caching use the same idea, accomplished with an index structure that reflects the information available in the server. When a cache miss occurs, a fault requirement based on the page or tuple identification is sent to the server. Using predicates, the semantic caching provides a completely different approach. Just analyzing metadata available about the cached queries, without navigating through indexed tuples, is possible to determine which information is locally available and which must be requested from remote servers. Furthermore, the local retrieve can be done in parallel with the remote request, improving the overall performance.

Finally, the cache replacement policies associate a value function to each cached item and choose as victims those items with lowest values. These functions typically exploit temporal locality or spatial locality. In the first case, the function is based on the assumption that items referenced recently have a high possibility of being referenced again. LRU and MRU algorithms are good examples of those approach. In the second case, the function is based on the assumption that items physically close to that recently referenced have a high possibility of being referenced again. The page caching tries to use this approach when accessing fixed size pages instead of tuples. The semantic caching extends this approach allowing the spatial locality to be adapted to the pattern of the queries.

In [27], it is also presented an evaluation of a semantic caching implementation against page caching and tuple caching. It concludes that the semantic performance and overhead are better under different scenarios of selection (e.g., indexed and no-indexed), altering different parameters (disk cache, memory cache, query size and replacement policies). In the worst case, both have the same performance or overhead, except when a semantic caching that does not coalesce entries is used with indexed selections. In this case, the semantic caching, for queries whose size is lower than six percent of relation, presents a worse overhead when compared with the page caching using LRU.

Besides the comparison established among the semantic cache, page and tuple approaches, the former can also be analyzed according to the following topics:

- (*i*) Definition and selection of queries suitable to be materialized. In our case, we are considering just queries constructed from an arbitrary number of (s)elect, (p)roject and (j)oin operations SPJ. This restriction is explained in the next section. Besides, along with this restriction the workload has a direct influence in the selection of the queries and this issue is presented in Chapter 7. For a formal approach to select queries to be materialized see [18] and for a commercial tool see [3].

- (*ii*) Storage and index creation. This topic is out of the scope of this thesis. The ideas are borrowed from optimization strategies well known in database systems. Please refer to [2, 81, 3, 18] for more information about this subject.

- (*iii*) Navigation structures to search for materialized views created. We presented in Section 5.2.2, an organization approach based on [39].

- (*iv*) Integration of materialized views with the optimizer. This regards production and evaluation of plans using materialized views, as well as detection of common expressions to replace parts of the queries with cached information. This topic too is out of scope of this thesis. Please refer to [17, 39] for a comprehensive treatment of this subject.

- (*v*) Maintenance algorithms which regard incremental update, avoiding the overhead of recomputing the materialized views. In Section 5.2.2, we analyze this subject in detail.

5.2.1 Query Matching

Paraphrasing [59], examination and maintenance of cached tuples via predicate descriptions entails determining satisfiability of predicates. The examination process, called query matching, searches for cached results to compute queries and can produce four distinct outcomes:

- None of the materialized views available in the cache can be used, which implies that a counterpart to the issued predicate was not found.

- One or more materialized views covers the issued predicate, being the optimizer responsible to decide which one is the best according to the cost model designed. Furthermore, a restricted predicate must be produced to eliminate possible exceeding tuples available in the chosen materialized view.

- One or more materialized views overlap with the issued predicate, but none has the complete information requested. Hence, the result must be the union of the partial overlaps and the additional information retrieved from elsewhere to complement the request.

- The issued predicate covers one or more materialized views, meaning that each one is contained within the predicate. Hence, the result must be the union of the materialized views which were covered and the additional information retrieved from elsewhere to complement the request.

In our approach, we consider a restricted but important class of materialized views based on SPJ queries. According to [12], this class of queries allows the development of feasible algorithms that can be used to detect irrelevant and autonomously computable updates regardless of some database semantic knowledge or database instance. [3] These properties are invaluable to our approach. For instance, the possibility to detect if an update (insert, delete and update statement) is irrelevant to the materialized views, meaning that the content of the materialized views can be preserved without further processing, is extremely important to improve the overall performance of the system. The possibility to refresh or recompute the materialized views using the updated information and the view itself avoids an access to base relations located on remote site(s) or the invalidation incurring in subsequent remote access(es). It is also important to notice that this last behavior is essential to optimize the certification process reducing aborts. The refresh problem is treated in detail in Section 5.2.3.

We also restrict the entries that are used to answer a request. Even though it is possible to group different entries providing partial answers to a request, our approach disregards this behavior considering that is the optimizer's job to produce subqueries that can take advantage of partial cached information. For that reason, we just consider entries in cache which cover the requested

[3] The term database instance is used in this case to indicate the actual state of a database system when an algorithm is applied.

queries, being again the responsibility of the optimizer to choose the most suitable according to a plan cost.

Formally, according to [17], we can state that a SPJ query q can be computed from a materialized view v, or the predicate of v satisfies q, if there is an attribute set A and a formula F such that, for any database instance d,

$$q(d) \leftrightarrow \pi_A(\sigma_F(v(d))). \tag{5.1}$$

Before continuing we need to introduce some additional notation. Consider v a materialized view defined as $v = (\mathbf{A_v}, \mathbf{R_v}, \mathbf{f_v})$, where $\mathbf{A_v}$ is the set of attributes projected, $\mathbf{R_v}$ is the set of relations referenced and $\mathbf{f_v}$ is the predicate. The query is similar defined as $q = (\mathbf{A_q}, \mathbf{R_q}, \mathbf{f_q})$. The function $\alpha(x)$ defines a set of attributes, where x can be a relation, a set of relations, a predicate or even a statement (i.e., a query or a materialized view). For instance,

- $\alpha(\mathbf{f_v})$ the set of all attributes referenced in the predicate $\mathbf{f_v}$.
- $\alpha(\mathbf{R_v})$ the set of all attributes of the set of relations in $\mathbf{R_v}$.
- $\alpha(v)$ the set of all attributes exposed by the view v being equal to $\mathbf{A_v}$.

The Equation 5.1 can be explained as follows. For SPJ queries the requirements presented bellow must be satisfied in order to be possible to use a cached result:

- **The base relations referenced in the cached statement must be equal to the base relations referenced in the query:** $R_q = \mathbf{R_v}$. Despite the restriction presented here, the integration with an optimizer can allow the production of subqueries when the queries have more relations than the actual materialized views. Further, in [39] is presented an approach which permits to use a materialized view with more relations whenever the cardinality of the materialized view is not changed with the extra relations.
- **The materialized relation must contain all rows needed to compute the requested expression:** $f_q \rightarrow \mathbf{f_v}$ and there exists a restricting formula f^r on v such that $f_q \leftrightarrow \mathbf{f_v} \wedge f^r$. This formula allows to eliminate the extra tuples belonging to v which are not of interesting for q.
- **All the required rows must be selected from the materialized view:** $\alpha(v) \supseteq (\alpha(q) \cup \alpha(f^r) \cup \alpha(f_q))$. This guarantees that all the attributes in the projection of the queries and the attributes required to evaluate the f_r are available.

5.2.2 Managing Materialized Views

To manage the materialized views, three aspects must be considered: (*i*) structure to index the cached results, allowing fast localization; (*ii*) storage mechanism to accommodate the results; (*iii*) replacement policies when there is not enough space available. The storage mechanism and the replacement policies are beyond the scope of this thesis. For a discussion about these topics see [27, 2, 81, 3, 18].

Basically, we designed a mechanism to index the cached results speeding up the matching process presented in the previous section. To accomplish this, we adapted the approach presented in [39], which is based on lattice [1].

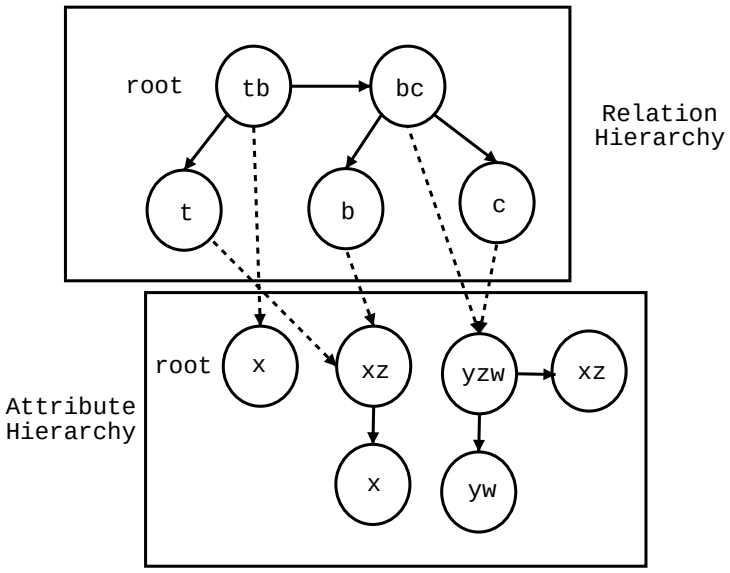


Figure 5.1: Hierarchical Filter Graph

We use a hierarchical filter graph depicted in Figure 5.1, where each node contains a key establishing a searching condition, a set of pointers and a set of neighbors. The keys on the first group of the hierarchical filter are associated with the set of relations referenced in the materialized views. The keys on the second group are associated with the set of attributes exposed by the materialized views, that is $\alpha(v)$. The set of pointers establishes a cover relation among the nodes. The nodes which are not covered by any other are called root nodes. If it is not possible to establish a cover relation among distinct nodes, a neighbor is created. In case of a node in the first hierarchical group, there is also another set which links the first and the second groups.

The hierarchical filter is constructed connecting one or more filter graphs, where each one represents a partitioning condition. In order to search for a materialized view, we start looking for relations in the set of root nodes, until a *match* or a *cover* (i.e., $R_v = R_q$ or $R_v \supseteq R_q$) is found. For each *cover* we proceed to the next level in the same hierarchical group (i.e., relations or attributes). For each *match* we access the correspondent node in the attribute hierarchy. Otherwise, we must proceed to the neighbor nodes. The same steps are applied to the attribute hierarchy. Finally, it is important to notice that the search algorithm must be recursively enforced until all the elements of the root of the relation hierarchy are evaluated.

5.2.3 Maintenance of Materialized Views

The maintenance problem consists of guaranteeing that the materialized views are consistent with the information from which they are derived. Thus, when the base relations are updated the following steps can be used to accomplish this: (*i*) compute the changes to the materialized views from the changes to the base relations and (*ii*) refresh the materialized views using the computed changes.

The last step leads us to reason about when a materialized view must be refreshed defining possible maintenance policies as follows:

Immediate Materialized Views: The views are refreshed immediately upon an update to a base relation. This approach can slow down update intensive transactions and increase the speed

of read intensive transactions.

Deferred Materialized Views: In this case, the updates to the materialized views are serialized after transactions which produced the changes. In order to use this mechanism, it is necessary to have a log which traces and stores the postponed updates. Different deferred maintenance policies can be defined:

Lazy Deferred: The views do not need to be immediately consistent with the base relations. Instead of updating the views during the transaction, the refresh is postponed until the view is required or accessed again.

Periodic Deferred (Snapshot): The views are periodically updated at pre-established times.

Forced Delay: The views are updated after a pre-established number of changes.

In our case, we cannot tolerate access to stale information since our effort is to build efficient protocols using strong consistency criteria. We could however allow to have a lazy deferred maintenance postponing the refresh without compromising consistency. However, we prefer to use an immediate maintenance approach avoiding an extra effort to trace changes.

The maintenance problem can also be observed along four dimensions [48]:

Information Dimension: The amount of information used or available to update the materialized views. Do we have access to all base relations? Do we know about foreign keys, primary keys, nulls? For instance, it is possible in some cases as stated before to refresh the materialized views using just the update statements (i.e., insert, update or delete) and the materialized views which is called autonomously computable updates.

Modification Dimension: The restrictions imposed to the update, delete and insert statements to base relations which the materialized views can handle.

Language Dimension: The restrictions imposed to the subset of the relational algebra, for instance used to define the materialized views. In our case, we consider SPJ queries.

Instance Dimension: Can the maintenance mechanisms be used in all databases or there are restrictions? We propose to use generic mechanisms that can operate regardless the database.

We consider the problem of updating cached information reasoning about the overhead involved to recompute it using the base relations. This perspective establishes three possibilities: (*i*) detect irrelevant updates, which avoids to recompute materialized views when their results are not changed; (*ii*) incremental update based on the materialized view and the changed tuples; (*iii*) discard the content of the cached information since its invalid.

Irrelevant updates

In this section, we present the basic ideas behind the algorithms used to detect irrelevant updates and to compute autonomous updates. The term update is used in a broader sense to designate insert, delete and update operations.

One naive approach to detect irrelevant updates, could be to test each tuple individually against the predicates defining the materialized views. Unfortunately, this solution would not scale up well. However this alternative can be the only one available. In this case, a trade-off between

detecting irrelevance and invalidating entries in cache must be established. For that reason, our effort consists in providing mechanisms that detect irrelevant updates but without incurring in unacceptable overheads. In order to do that, we use the ideas from [12] as a start point.

First we introduce some additional notation. The insert operation is defined as $\{INSERT(R_i, T_i)\}$, where R_i is the relation to be updated and T_i is the set of tuples to be inserted. The delete operation is defined as $\{DELETE(R_d, f_d)\}$, where R_d is the relation to be updated and f_d is the predicate to be applied. The update operation is defined as $\{UPDATE(R_u, f_u, U_u)\}$, where U_u is the set of updated attributes $U_u = \{u_1 = c_1, ..., u_n = c_n\}$, $u_1, ..., u_n \in R_u$ and c_n belonging to either the integer or real domain.[4]

Informally, we can define that an insert operation into a base relation is irrelevant to a materialized view if it causes no tuple to be inserted into the materialized view. In order words, this can be expressed as follows: an operation $\{INSERT(R_i, T_i)\}$ is irrelevant to the materialized view defined by $v = (\mathbf{A_v}, \mathbf{R_v}, \mathbf{f_v})$, $R_i \in \mathbf{R_v}$, if and only if $\mathbf{f_v}(t)$ is unsatisfiable to every tuple $t \in T_i$. It is worth noticing that this approach does not consider insert operations in which there is a sub-select statement, which means that the irrelevance is evaluated using the tuples produced, regardless of the predicate.

Informally, we can also define that a delete operation is irrelevant to a materialized view if it causes no tuple to be deleted from the materialized view. In other words, the operation $\{DELETE(R_d, f_d)\}$ is irrelevant to the materialized view defined by $v = (\mathbf{A_v}, \mathbf{R_v}, \mathbf{f_v})$, $R_d \in \mathbf{R_v}$, if and only if the condition $f_d \wedge \mathbf{f_v}$ is unsatisfiable.

It is simple to see that the definition for deletion it is quite similar to the one provided for insertion, being enough to use a similar algorithm. However, in contrast to the insert, the delete operation allows a predicate whose terms are attributes of R_d. Thus, we can use it to improve the tests applied, avoiding to test each tuple. Nevertheless, whenever a sub-select, join or something similar is allowed, we can turn out to the generic solution which verifies each tuple.

The update operation has an algorithm more complicated than insert and delete. Informally, an update is irrelevant when the following situations happen:

- The updated tuples must be irrelevant before the statement is issued, and after its execution. This reasoning allows to detect updated tuples that already belonged to the materialized views, and also tuples that did not belong to them, but beginning to do, after the update.
- Even the updated tuples which are not irrelevant according to the previous definition can still be considered irrelevant if the changed attributes exposed in the materialized views remain with the same values.

Considering just the first situation, the operation $\{UPDATE(R_u, f_u, U_u)\}$ is irrelevant to the materialized view defined by $v = (\mathbf{A_v}, \mathbf{R_v}, \mathbf{f_v})$, $R_u \in \mathbf{R_v}$, when the following expression evaluates to false:

$$(f_u \wedge \mathbf{f_v}) \vee (f'_u \wedge \mathbf{f_v}(U_u))$$

where f'_u is the original predicate after elimination of the attributes that appear in $\alpha(\mathbf{f_v}(U_u))$. It avoids contradictions arising upon evaluation of updates similar to ***update*** R_u ***set*** $u_1 = c_1$ ***where***

[4] In [12] is presented a different definition of the update operation. Instead of using U_u, it defines F_M in its place as the set of update expressions. Unfortunately, it does not restrict the type of expressions, allowing expressions which do not belong to the class of boolean expressions. Hence, it prevents the use of our algorithms and even the algorithms that it proposes. In Section 5.3, we show that this distinction is irrelevant to our approach.

u_1 = c_n. Furthermore, observing the update operation as a delete and after an insert, the first part of the disjunction deals with the delete operation, regarding the views which are affected before the changes. The second part of the disjunction deals with the insert operation, regarding the views that are affected after the changes.

Unfortunately, our approach does not consider the second situation in which the attributes exposed in the derived relations remain the same. Even though it classifies some updates as relevant when an approach which verifies the second situation does not, it never classifies an update as irrelevant when it is relevant. To develop such an algorithm, we must evaluate the tuples which cannot be always efficiently although the implementation is quite simple.

Autonomously Computable Updates

Using the processes previously described, we are able to detect the irrelevance of an update. However, if the irrelevance is not verified, a sequence of steps is still required to attempt to recompute the derived relations using just the updated tuples and the materialized views.

Informally, we define that the effect of an insert operation can be autonomously computable if the following conditions are verified:

- For each inserted tuple, it must be possible to decide whether it satisfies the materialized view's predicate or not.
- The values for all visible attributes in the materialized view must be obtained from the updated tuples.

Intuitively, it is simple to realize that the first requirement is accomplished according to the theoretical background provided to detect irrelevant inserts. However, the second requirement needs more attention. In cases where $\mathbf{R_v} \supset \{R_i\}$, the situation in which is possible to achieve the requirement is when $(\alpha(v) \cup \alpha(\mathbf{f_v})) \subseteq \alpha(R_i)$. Unfortunately, regardless of this restriction, it is not always possible to autonomously compute the inserts for all database instances when $\mathbf{R_v} \supset \{R_i\}$. For demonstration, suppose a database instance d defined over a set of relations R, where $\mathbf{R_v} \subseteq R$ and $R_z, R_i \in \mathbf{R_v}$. Furthermore, $R_z = \emptyset$ and $z \neq i$. In this case, the view v is an empty set, regardless of the inserts into R_i. Nevertheless, if we just have had followed the previous requirements it would be necessary to insert the new tuples into R_v. As a consequence to handle inserts autonomously the materialized view must be built using just one base relation.

Formally, the effect of an insert operation $\{INSERT(R_i, T_i)\}$ on a materialized view, defined by $v = (\mathbf{A_v}, \mathbf{R_v}, \mathbf{f_v})$, is autonomously computable if and only if $\mathbf{R_v} = \{R_u\}$.

In order to autonomously compute delete operations, the attributes appearing on the materialized view must be sufficient to evaluate the delete, which means that all attributes referenced in f_d appear on v. Hence the effect of the operation $\{DELETE(R_d, f_d)\}$ on the materialized view $v = (\mathbf{A_v}, \mathbf{R_v}, \mathbf{f_v})$ is autonomously computable if and only if $\alpha(v) \supseteq \alpha(f_d)$.

In [12] is presented a weakest requirement. However, it requires a solution to an implication problem which can contribute to reduce the performance of our approach. For that reason, we prefer to adopt our strong proposal without incurring in this additional overhead. In Section 5.3, we will come back to this subject giving details about the integration in **Escada** and demonstrating that this choice is worthwhile.

In case of the update, we can choose to handle the problem either as a combination of a delete and after an insert or as a different operation. In the first choice, there is a laborious work in order

to remove the tuples from the materialized view and afterwards to insert some tuples which were removed. It is also important to notice that this approach combines restrictions from the delete operation and from the insert. Even though the second choice avoids this behavior, it requires some steps that use satisfiability algorithms. For that reason, we adopt the first approach. For a detailed explanation on how to handle the update operation without combining delete and insert see [12].

5.3 Contributions

5.3.1 Algorithm

We present an algorithm based on [39] to determine whether some derived relations can be used to compute the queries or not. Furthermore, upon receiving an update statement, the algorithm also identifies whether it is possible to autonomously computing the changes or not. In order to do that, we make an assumption that the derived relations are organized according to the schema outlined in Section 5.2.2. The algorithm is presented as follows:

- **Identification** - The first step consists on recognizing the statement as either a select, insert, update or delete.

  Queries The queries lead to a process of query matching followed by either an answer using a materialized view or an insertion in cache.

  Updates The update statements lead to the detection of irrelevance. In case of the update being irrelevant to the derived relations, nothing is done. Otherwise, the algorithm tries to autonomously compute the updates for each materialized view for which the update is relevant. When it is possible to refresh the materialized view, it applies the correct procedures according to the statement recognized, and when it is not possible, it removes the materialized view from the cache.

- **Conjunctive Normal Form** - The statements to be evaluated must have their predicates organized in a conjunctive normal form. Usually, the query processor is responsible for that and the algorithm does not need to consider this step. However, it is presented here for completeness covering the case of using the algorithm outside the database mechanism.

- **Extracting Information** - The relations, attributes and predicates that appear on the statements must be identified as follows:

  Queries The queries to be evaluated must be SPJ queries as defined before. Otherwise, the cache cannot be used. The expressions belonging to the predicate must be grouped as either *equi-join expressions*, *range expressions* or *residual expressions*. The relations appearing on the queries must be extracted and also the projected attributes.

  Updates The updates, inserts and deletes must reference at most one relation. The same classification of the predicate is valid for the update statements. If the updates do not match the specified pattern, we use the referenced relations to invalidate all the cache entries who have one of the relations, that is, $R_i \subseteq \mathbf{R_v}$ or $R_d \subseteq \mathbf{R_v}$ or $R_u \subseteq \mathbf{R_v}$.

- **Classes of Equivalence** - The next step consists on organizing the attributes in classes of equivalence. Basically, the class identifies which attributes are interchangeable. For instance, this knowledge can be used to decide to autonomously compute a delete when all

the attributes appearing on its predicate are not projected by a materialized view. We start defining a class for each referenced attribute. Recursively, for each element $R_i.a_1 = R_j.a_2$ (i.e., *equi-join expressions* set), we find the set containing $R_i.a_1$ and the set containing $R_j.a_2$. If they are in different sets, we merge the two sets, otherwise we do nothing.

- **Comparing Relations** - The information extracted from the previous steps is used to search the hierarchical filter outlined in Section 5.2.2 and to store the derived relations in cache.

  Queries It is important to remember that the set of relations from the query must be equal to the set of relations from the materialized view as defined in Section 5.2.1. This requirement is just a simplification rather than a technical or theoretical limitation. In [39] is presented an approach that considers supersets of the requested relations. Furthermore, this restriction can be also circumvented changing the optimizer to produce sub-plans using subsets of the requested relations.

  Updates In case of updates, it is sufficient that the relation appearing on the statement belongs to the set of relations of each materialized view.

- **Main Operations** - In this phase of the algorithm the queries will continue the matching process and the updates will be classified as irrelevant, autonomously computed or used to invalidate cache entries. Hence, for the materialized views retrieved from the previous step the following procedure is applied:

  Queries Using the classes of equivalence defined before, we can determine if all required attributes are available. For each equivalence class of the query, at least one of its columns must be available in the exposed attributes of the materialized view. In fact, this is still done searching in the hierarchical filter but now using the attribute level. Finally, to answer the queries the algorithm chooses the first view whose predicate satisfies the requirement $f_q \rightarrow \mathbf{f_v}$[5]. If this materialized view does not exist, the queries are inserted in the cache. Otherwise, the tuples in the materialized view that satisfy the materialized view's predicate are used to build the answer.

  Updates We present a distinct algorithm for each update statement.

  Insert In order to see if an insert is irrelevant, the algorithm proceeds as follows. For each tuple to be inserted, we substitute the values of the attributes appearing on the materialized view's predicate and test the satisfiability. In the case of the predicate being satisfied, the insert is relevant, otherwise it is not. In the second case, the algorithm must test if the insert is autonomously computable. In order to that the set of relations referenced in the materialized view must be equal to the set of relations referenced in the insert. If this situation happens, the algorithm inserts the tuples that satisfy the materialized view's predicate, otherwise it invalidates the materialized view.

  Delete A delete is irrelevant if the expression $f_d \wedge \mathbf{f_v}$ is unsatisfiable. In case of being relevant, the algorithm must test if the delete is autonomously computable. It does that testing if $\alpha(v) \supseteq \alpha(f_d)$. When the test evaluates to false, it invalidates the materialized view. Otherwise, for each tuple in the materialized view it tests the tuple against the delete's predicate and removes the tuple when the test evaluates to true.

  Update The update is treated as a sequence of delete and insert operations. First of all, regarding the delete operation, the algorithm tests if the materialized view is

[5] It is important to notice that this restriction is not a limitation of our algorithm, but just a simplification since our current prototype does not have the semantic caching integrated with the optimizer.

irrelevant using the following expression $(f_u \wedge \mathbf{f_v})$. If it is relevant, the algorithm applies the tests to see if it is possible to autonomously compute the deletes. If it is not possible it invalidates the materialized view and goes to next materialized view. Considering now the insert operation, the algorithm proceeds with tests of relevance using the following expression $(f'_u \wedge \mathbf{f_v}(U_u))$. If it is relevant, the algorithm applies the procedures to see if it is possible to autonomously compute the inserts. If it is not possible it invalidates the materialized view. Otherwise, for each tuple in the materialized view, it tests the tuple against the delete's predicate. If the test evaluates to true the tuple is removed. For each tuple to be inserted, it tests the tuple against the materialized view's predicate. If the test evaluates to true the tuple is inserted.

- **Satisfiability** - The tests of satisfiability are applied according to the algorithms presented in Section 5.1. This verification is made using just the *equi-join expressions* and *range expressions*. In order to test the validity of the *residual expressions*, consider each one as a sequence of characters without spaces and compare the string with the *residual expressions* from the materialized views manipulated in the same way. It is important to notice that we cannot remove spaces from constants similar to "string". It is also important to remember that we must use the classes of equivalences when comparing the strings. Finally, if these expressions are equal, we can proceed. Otherwise, we can state that the whole expression is unsatisfiable no matter what the other expressions are.

5.3.2 Extending the Escada

In this section, we present the modifications required to the **Escada** in order to exploit and support the semantic approach described. We outline the new distributed execution considering the existence of the cache and discuss possible design issues in relation to the consistency criteria. Following, we present the problem of refreshing the cache regarding the termination protocol adopted.

Exploiting the semantic cache, we can reduce the communication among distributed sites. Thus, before contacting a remote site, the database must evaluate the requested statement against the cache. First of all, this process consists on determining the class of the operation: select or update. In case of a select, the database site verifies if there is a materialized view that could be used to compute the request. If so, it returns the result without incurring in a communication overhead. Otherwise, it contacts the remote site and upon receiving the results generate a private cache entry which is kept invisible to the other transactions until the end of the certification.

In case of an update, the process is more complicated because we need to guarantee *one-copy serializability* (1SR). Generally speaking, we consider a shared cache per database site that is autonomously updated or invalidated upon transaction certification. During transaction execution, the possible changes are visible per transaction.

In other words, when an update is received, the database site verifies if the update is irrelevant to all derived relations. If so, it simply sends the request to the remote database site. Otherwise, it must annotate that some derived relations are not valid to a specific transaction. For each materialized view affected, it must either (*i*) indicate that the changed tuples are not valid and point to the new values; or (*ii*) indicate that the entire materialized view is not valid. In the first case, the update is autonomously computable. In the second case, it is not.

Upon certification, the shared cached entries must be brought up to date according to the

transaction outcome. In case of abort, nothing needs to be done. In case of commit, the changes to the cache that were only valid to a specific transaction must be shared. It is worth noticing that this process must be executed atomically according to the commit being applied.

Nevertheless, the reasoning described in the previous paragraphs considers the changes made on a semantic cache in a local database site. We must regard also the impact of the transaction on remote semantic caches. To do that, the remote site relies on the information propagated in the broadcast in order to certify the transaction. Unfortunately, the information propagated depends on the termination protocol implemented. For that reason, we analyze in the next paragraphs the possibilities to manage the cache according to the termination protocols designed to **Escada**.

Cache and PDBSM - In the termination protocol proposed for the PDBSM, the transaction carries read and write sets. The write values are reliably propagated just to the sites that have a replica of the updated relations. Using this information, we can start to the delineate a protocol to refresh the cache.

Since we just have the write sets, which means relations and primary keys of the changed tuples, the best we can do is presented as follows. Using the identification of the relations according to the write set, find for each relation the materialized views that are based on it. Following, for each materialized view found compare the primary keys of its tuples with the primary keys of the write sets. In case of matching, invalidate the materialized view. It is worth noticing that this process must be executed atomically according to the commit being applied.

Cache and PDBSM with RAC - In the termination protocol proposed for the PDBSM with RAC, the transaction carries only its identification. The write values, read and write sets are reliably propagated just to the sites that have a replica of the accessed relations. Using this information, we can start to the delineate a protocol to refresh the cache. In this case, there is nothing that we can do. We must augment the transaction with additional information. For example, the identification of the relations updated or the write sets. For the remaining portion of this section, we assume that the write sets are propagated until otherwise stated.

In both protocols, we are supposing that the materialized views have the attributes used to build the primary keys of each base relation referenced in the views. This assumption is completely fair since we need to augment the query processing mechanism as explained in Chapter 3 to retrieve the read sets. Using this assumption, we can improve our algorithms. For instance, we can always autonomously compute the delete operation appealing to a comparison per tuple. Of course, we must establish a trade-off between the laborious work of a comparison per tuple and an invalidation of the materialized view.

Finally, we can use the ideas behind the materialized views to attempt to reduce the number of aborts in consequence of coarse grains (see Chapter 3). Generally speaking, whenever the number of tuples is above a established threshold we reference that the entire relation is accessed. Even though this approach can reduce the required bandwidth to propagate the transaction, it can increase the number of aborts. For that reason, we propose to augment the information carried on by the atomic multicast in order to allow a smart certification without overloading the network. We propose to transport also the queries that reference these relations which triggered this contention mechanism. Doing that, we can start a detection of irrelevance whenever a conflict arises in consequence of these coarse grains.

5.3.3 Related Work

The idea of exploiting the semantic (i.e., predicates) involved in SQL processing is almost as old as the relational database research. In the seventies, it was suggested the use of predicate locks for concurrency control in order to avoid *phantoms* [29]. However, the implementations have not been successful because of its pessimistic assumptions and its execution cost. For instance, a transaction could be blocked waiting for another to release locks even when regarding relations without data. Our proposal improves transaction execution reducing remote communication. It is not designed to perform concurrency control.

According to [50], the approaches to the problem differ mainly depending whether they are concerned with query optimization and database design or with data integration. In the case of query optimization and database design, the main idea is related to computing queries based on previous executions, called materialized views or derived relations. The efficient optimization of queries using materialized views, even not being a recent concern, has been the subject of much research [39, 50, 7]. Especially as a consequence of the increasing interest in data warehouse [50] and data stream processing [5]. In a broad manner, this problem is concerned with: (*i*) the definition of queries suitable to be materialized; (*ii*) storage and automatically index creation; (*iii*) establishment of navigation structures to search for materialized views created; (*iv*) detection of common expressions to replace parts of the queries with the cached information; (*v*) integration of materialized views with the optimizer; (*vi*) incremental update of materialized views.

Data-shipping architectures [27] popularized by early generations of Object-Oriented Database Management Systems, represent other good examples where optimization and database design are concerns. Instead of processing queries sending the request to the server, the access occurs in a fault basis whenever the information is not available in the cache. In contrast to our approach, it requires that the server manages a central index structure such that when an information is updated, the server is responsible to inform the pertinent clients. In our case, the existing broadcast is used to accomplish this.

In the case of data integration [66, 86], the main focus has been on translating queries formulated in terms of an integration middleware into queries formulated in terms of a specific data source.

In a matter of fact, all the researches mentioned here are somehow related, regardless the subtle details, since the concepts and the theoretical background involved are the same. For that reason, we can generically define this research as studies on rewriting queries using queries. In other words, whether some queries are capable of being satisfied using other queries or not. Nevertheless, from the best of our knowledge, our approach is the first suited to replication environments based on group communication. To support our approach, we submitted our protocols to a set of simulations described in detail in Chapter 7. Using these simulations, we aim at testing our development efforts and demonstrating our ideas. Not less important, we also integrate our efforts into the PostgreSQL [43] in order to make available a distributed execution mechanism augmented with semantic cache and smart certification.

Chapter 6

PDBSM and PostgreSQL

In this chapter, we present how to materialize the approaches proposed for the PDBSM into a real database system. Generally speaking, we choose the PostgreSQL [43] as our target database. We discuss critical decisions that must be taken into account to integrate the PDBSM into the PostgreSQL and analyze the consistency criteria according to the abstract definition of the DBSM provided by [74]. We also design extensions to the PostgreSQL's grammar, in order to provide the commands to create, delete and change the fragments of a relation. Finally, we present how the PostgreSQL may be extended in order to provide a simple distributed execution mechanism.

The rest of this chapter is organized as follows. In Section 6.1, we present a definition of read and write sets based on relational algebra and describe a possible mechanism to extract them and discuss the consistency criteria provided by the DBSM. In Section 6.2, we present how to extend the PostgreSQL in order to materialize the distributed execution.

6.1 Read and Write Set

In this section, we present a definition of read and write sets based on the relational algebra and an extraction mechanism. We also discuss consistency problems that can arise when this mechanism which simply implements the definitions from [74] is used.

6.1.1 Definition and Extraction

To execute an operation a database can read an entire relation or just a subset of it, a decision that certainly depends on the data and ultimately on the statistics available for the optimizer [56, 80, 66]. For instance, the statistics can lead the optimizer to decide for an execution using a specific index among others, or to read an entire relation when it has a small size. In this last case, the scan is chosen because the optimizer based on statistics decides that this is the approach that uses minimum resources or takes less time. In order to avoid read sets with unnecessary data, which has a negative impact on network bandwidth and on transaction's latency, a deterministic approach must be used.

We consider that a read set is defined as the set of tuples read during a transaction's execution and projected over the primary key attribute(s), such that no proper subset exists. In other words, for each read operation it does not exist a proper subset of the primary keys produced which can be joined with the cartesian product of the relations referenced in the read operation to get the same

results.

The write set can be defined as the updated tuples (i.e., inserted, updated and deleted) projected over the primary key attribute(s). The write value can be defined as the updated tuples themselves.

The extraction of the write set or the write value is quite simple. On the other hand, the extraction of the read set is a complex process based on the sequence of steps of the query trees defined during a transaction's execution, where each transaction's operation leads to a different query tree. Generally speaking, the process consists on gathering intermediate results from the highest steps, considering a bottom-up processing, that still conserve the primary key attribute(s) of each base relation referenced in the tree. It can be defined as follows:

- The extraction must contemplate all the relations involved in the query processing.
- The primary key attribute(s) of each base relation must be returned and if necessary artificially introduced in it projection steps.
- Can exist more than one node in the query tree from which the primary key attribute(s) must be gathered to build the final result set.
- Once gathered the result set from a node, nothing prevent this result to be used as an input to another set of the relational step to produce more points of extraction.
- Update and delete operations are composed by a read followed by a write.
- Aggregation functions are only considered for relations that can have its primary key attribute(s) introduced in the result set without changing the meaning or the characteristics of the result set.
- When the base relation cannot have its primary key attribute(s) as part of the result set, in consequence of an aggregation function, as stated in the previous statement, the base relation must be considered as a whole or just filtered according to the where clause.

6.1.2 Phantom Anomaly

It is important to notice that the read and write sets defined above and combined with the certification process introduced in Section 3.2 do not prevent the phantom anomaly [8]. For instance, the phantom anomalies can arise when a transaction t selects a set of tuples based on a predicate such as $R.a_1 >= 10$ and a concurrent transaction t' inserts a tuple that does not exist before t', where $R.a_1 = 10$. Even though the transactions are concurrent, there is not a conflict according to the concepts presented in Chapter 3, since the read set of t does not intercept with the write set of t' and vice-versa. However, this scenario can lead to serious problems. Suppose that t is selecting the sales above a threshold to produce an extremely important report but does not see the insertion generated by t'. In this case, the report will not reflect the actual state of the database.

To avoid these anomalies, the proposed definitions must also take into account relations and indexes as read and updated information.[1] Unfortunately, this solution has some drawbacks. In the first case, upon receiving a transaction similar to t the database must indicate that the relation R was read and upon receiving a transaction similar to t' that the relation R was updated. Hence while evaluating the conflicts exists an overlap between the read and write sets of the transactions, i.e. the relation R. Of course that such an approach may increase the number of aborts. It is

[1] This solution is similar to hierarchy of locks in database systems. See [102] for additional information.

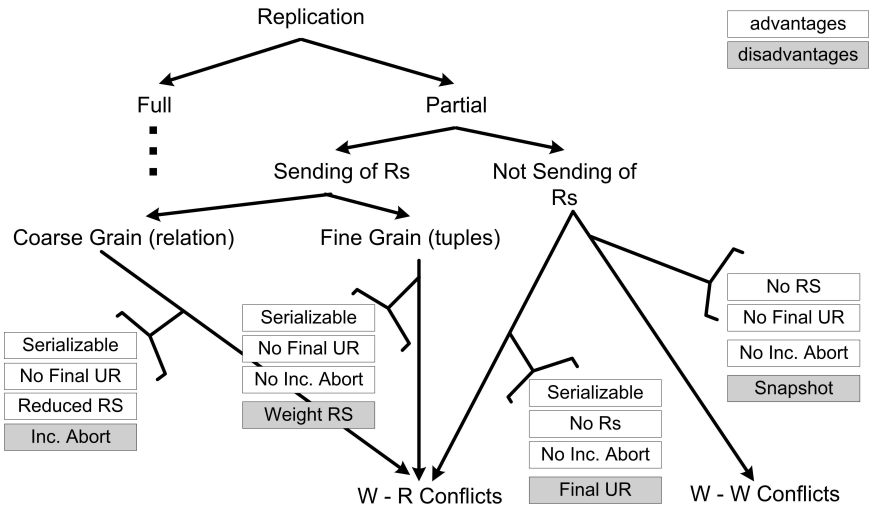


Figure 6.1: Design Issues: Read Set Decision Flow

enough having a transaction that reads a tuple of a relation and another concurrent transaction that updates a completely different tuple at the same relation to arise a conflict and later an abort. In order to attempt to reduce this problem, we can create indexes and indicate read operations also on indexes. However, this approach is possible if and only if there are indexes with the same attributes of the predicates, which is not always true. Thus, whenever an index is available we indicate that the index was read or updated. Otherwise, we resort to the relation.

6.1.3 Read Set and the Consistency Criteria

Unfortunately, despite our concerns and optimizations, sometimes it is prohibitive to send the read sets as a consequence of the amount of information to transfer.

In order to help us to decide about possible changes in the protocols involving the read set issue, consider the Figure 6.1. One of the first choices is on the definition of a threshold per relation from which the read set is sent by comprehension.[2] This solution can reduce the usage of network bandwidth and transaction's latency, although it makes difficult the detection of conflicts since the "accessed tuples" are not sent. A simple possibility is to detect conflict with a coarser grain based on the relation that was sent by comprehension. Unfortunately, this solution topically implies the growth of the number of aborts as a consequence of the "false read tuples". It is a situation similar to the phantom anomaly.

We can attempt to reduce the usage of network bandwidth and also control the growth of the number of aborts, sending the relation by comprehension as stated before and also the queries involving the relation. In case of a conflict, we evaluate the updated tuples against the queries in a similar process to that used to manage semantic caching. See Chapter 5 for a detailed explanation of this subject.

A different alternative to avoid the problems of sending the read set by comprehension or by extension has been presented in [63]. In this approach, just the write set is sent to certification. However, the initiator site is the only one that can decide about the outcome of the transaction. This happens since the read set used to detect conflicts is local to it, which in turn requires a final

[2]The term comprehension is used to indicate the read and write sets represented in a high level format (e.g., SQL statement) or even in a coarse grain (e.g., relation).

round in the protocol to decide about the abort or commit.

The last alternative available is also presented in [63]. It releases the strong consistency criterion resorting to a snapshot isolation level where only write/write conflicts are detected. Unfortunately, this approach brings the phantom and write skew anomalies. Finally, it is worth noticing that this consistency criterion is implemented by Oracle and PostgreSQL.

6.2 Extending PostgreSQL

Our first step in the direction of the distributed execution consists on extending the PostgreSQL with the notion of logical and physical objects. PostgreSQL considers as an object any element created in the database, such as relations, triggers, constraints and rules. However, in order to restrict the impact of the changes on PostgreSQL, specifically on the catalog and the semantic data control[3] (i.e., permissions, rules, triggers), we propose to augment just the relations. The logical relations correspond to the original relations prior the fragmentation and the physical relations correspond to horizontal fragments. We again restrict our proposal, in this case to horizontal fragmentation, to easily integrate with future releases. The main ideas are:

- The physical relations are created using the normal mechanisms available in the PostgreSQL.
- Except rules, no object can be create based on a logical relation.
- The logical relations act like a view, almost having the same properties.
- In contrast to a view, it is not possible to create a logical relation based on another logical relation.
- The rules are used to define the fragmentation, establishing a connection between the logical relation and the physical relation(s).
- The user can manipulate logical or physical relations. When a logical relation is referenced, the system must transparently accesses the physical relation(s).
- The fragments can be located in any database site.

In order to materialize these ideas, we can exploit the fact that PostgreSQL has a centralized query processing mechanism with the same components or modules of Figure 4.2. Thus, we augmented the parse module, changing the SQL Grammar of the PostgreSQL to allow the manipulation of logical relations. See “Extensions to PostgreSQL’s Grammar” at the end of this chapter.

The first step consists on allowing the creation of logical relations. It is important to notice that the logical relation and the fragments must have the same structure, although each one is created in different steps. The user must guarantee this. Internally, as stated before, the logical relations must be treated like a view and physical relations are created using separate commands available in PostgreSQL.

The next step consists on allowing the mapping between the logical and physical relations. Once more, we resort to modifications on the SQL Grammar and exploit the rule system (i.e.,

[3] See [71] for a detailed discussion about semantic data control in distributed databases.

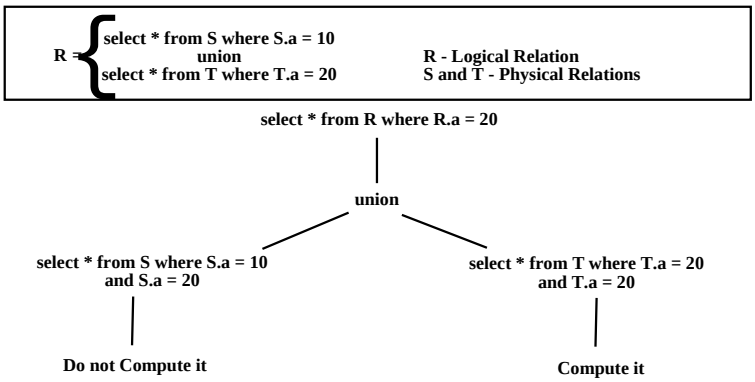


Figure 6.2: Pruning Irrelevant Expressions

rewriter) in PostgreSQL [43, 92]. It gives users the capability of defining rules as well as data. Basically, it specifies actions that must be done instead or with the original request, according to the operation (i.e., insert, delete or update) and when a pre-defined condition (i.e., predicate) evaluates to true. Using it, we want that whenever a logical relation is referenced in a select, insert, delete and update statement, it is unconditionally substituted by physical relation(s) according to the mapping established.

Select Operation

When a select is requested the rewriter must produce the following transformation: $\sigma_{f_s}(R) = \bigcup \sigma_{f_s}(R_i), \forall\ R_i \in \mathit{frags}(R)$, where R is the logical relation, each R_i is a physical relation and f_s is the predicate that appears on the statement. Unfortunately, the current parser available in PostgreSQL does not allow to create selection rules with *union*. We needed to extend it. Further, we needed to stop the progress of irrelevant selections. In this case, these selections are produced while combining the predicate of the statement with the predicate of the fragment, resulting in a worthless predicate. In other words, there is no tuple that satisfies this predicate. For instance, see Figure 6.2, where the select referencing the relation S does not need to be executed. Its predicate is a contradiction, which means that it always evaluates to false without bringing additional tuples to the final result. Instead, its execution can contribute to reduce the overall performance, being worse when a remote site must be contacted. See Chapter 5 for a detailed explanation about predicate contradiction.[4]

Update Operation

In case of the update statements, the reasoning is not so simple. The horizontal fragmentation is built creating selections over the set of attributes of a relation. For that reason, when an update is requested, the rewriter must redirect it to the correct fragment, which means that it must redirect the update according to the predicates available in the statement combining it with the fragments's definition. For instance, suppose that we have two distinct fragments[5], R_1 and R_2, defined respectively as $R_1 = \sigma_{f_1}(R)$ and $R_2 = \sigma_{f_2}(R)$, where $f_1 \neq f_2$. The update statement is ***delete from*** R ***where*** f_u, where $f_u = f_1$. Thus the update must be applied to the fragment R_1. This

[4]This idea can also be used to guarantee completeness and disjointness. See Section 2.3.

[5]For our proposal in this section, let us assume that the predicates can be equal or different, rather than using implication to establish that a predicate implies another.

situation is similar to the execution of the select statement presented before and can be handled using the same set of tools. Nevertheless, if not handled the damage can be worse, generating updates against improper relations.

In order to provide the essential information for the rewriter to do its job correctly, we must define for each update operation (i.e., insert, delete and update) and for each fragment a rule based on the common logical relation, using the commands in listing "Extensions to PostgreSQL's Grammar". Unfortunately, we cannot simply rely on the default behavior of the rewriter in PostgreSQL, since it acts as follows:

- For the insert operation:
  - It applies the original insert into the "logical relation", and after that evaluates the "inserted" tuples according to the predicates (i.e., conditions of the rules) defined to characterize the fragments. In fact, in our proposal, no operation is performed against the logical relation.
- For the delete operation:
  - It selects the tuple(s) from the "logical relation" according to the predicate specified in the original statement, and after that evaluates the "selected" tuples according to the predicates (i.e., conditions of the rules) defined to characterize the fragments. In fact, in our proposal, no operation is performed against the logical relation.
- For the update operation:
  - The same behavior specified for the delete operation.

For each update operation we must guarantee the following requirements:

- For the insert operation:
  - Each tuple to be inserted into a relation must be evaluated against the predicates that define the fragments. The outcome of the evaluation is used to decide in which fragment the tuple must be inserted. In situations that none of the fragments is chosen, we can report an error or proceed.
- For the delete operation:
  - For each fragment of the relation referenced in the statement, combine the predicate of the statement with the predicate of the fragment. If there is not a contradiction, process the delete against the fragment. Otherwise, do nothing.
- For the update operation:
  - For each fragment of the relation referenced in the statement, combine the predicate of the statement with the predicate of the fragment. If not exist a contradiction, process the update against the fragment. Otherwise, do nothing. The update must be processed taking into account if the new value(s) are different from the value(s) that are used to define the fragment. For example, suppose that a fragment R_i is defined as $R_i = \sigma f_i(R)$, where $f_i = \{t \mid R_i(t) \wedge t.a_1 = c\}$. The update statement is ***update*** R ***set*** $R.a_1$ **= *b where*** $R.a_1$ **= *c***, where $b \neq c$. In this case, the update must be treated as a delete followed by a set of inserts. Finally, we must consider the situation that

the update changes the values of the attributes and the new tuple satisfies none of the available fragments. We can choose to delete the tuple or we can choose to abort the operation and then the transaction. It seems quite reasonable to choose the later approach because it avoids the deletion of tuples when the operation requested was an update.

After all these steps, the logical and its physical relations are created. The augmented PostgreSQL can transparently execute the commands. However, the definition process is laborious and can lead to several inconsistencies, while conducted by the user. For that reason, we propose to create a high level command to implement this definition, although internally, it resorts to the set of commands and procedures presented here (See "Extensions to PostgreSQL's Grammar" at the end of this chapter and the Appendix B for a brief example of the set of commands).

Catalog, Optimization and Execution

Except for the information related to logical relations, the metadata is locally stored in the catalog of the database site where it was created. Whenever the database receives a request to manipulate a logical relation the process is synchronously applied at all the replicas, before returning the outcome to the user.

We use a two-step optimization to process the queries and preserve the normal optimization mechanisms available in PostgreSQL. Everything is done like in a centralized execution until the queries arrive at the executor module. Before execution, the PostgreSQL's executor module decides in which site each node of the query tree will be executed. Initially, we propose to choose the first possible and available site, avoiding problems to integrate with future releases.

However, to accomplish this goal, it is necessary to augment the PostgreSQL. In this case, we must provide means by which the replicas are identified and the fragments are placed at specific replicas. Basically, we add the name of the replica to the definition of the fragment (i.e., physical relations) and provide the set of the replicas where a fragment is placed. See "Extensions to PostgreSQL's Grammar" at the end of this chapter for a definition of the commands.

Extensions to PostgreSQL's Grammar

```
CREATE LOGICAL TABLE ltable
(
 columname columndatatype [,...]
)

DROP LOGICAL TABLE ltable [,...] [CASCADE | RESTRICT]

CREATE LOGICAL RULE rulename AS
 ON event
 TO ltable [WHERE qualification]
 DO INSTEAD [action | NOTHING]

DROP LOGICAL RULE rulename [,...] ON ltable

CREATE ESCADA TABLE ltable
(
```

```
 (
  columnname datatype [DEFAULT default_expr] [columnconstraint [,...]]
  |
  tableconstraint
  |
  LIKE parenttable [( INCLUDING | EXCLUDING ) DEFAULTS]
 ) [,...]
) [FRAGMENTS (clause,host) [,...] ]

DROP ESCADA TABLE ltable
```

Chapter 7

Results and Performance Analysis

In this chapter, we evaluate the database replication based on group communication and its cost using workloads widely adopted to measure performance of commercial database servers. Specifically, we propose to evaluate the **Escada** protocols using the TPC-W and TPC-C benchmarks [101, 100]. This effort is extremely important since a truly evaluation is highly dependent on the access patterns provided by the workloads. Unrealistic workloads are in this case worthless, precisely, because they may not be able to represent the appropriate concurrency and hot spots according to the database size and number of clients. The evaluation of the DBSM, PDBSM and the semantic caching directly depends on these characteristics.

In this section, we present our experimental results performing the evaluations as follows. First, we analyze the semantic caching in a single database machine in order to figure out its behavior (i.e., hit ratio and overhead) while changing the number of entries in cache, the number of clients and the workload. Second, we proceed to the evaluation of the PDBSM and the PDBSM with RAC. Specifically, we evaluate the distributed execution mechanisms and the advantages of the semantic caching, considering the results established with the previous experiments. In order to set a baseline, we run some experiments with the assumption that a site is locally able to complete the execution of a transaction, which avoids possible overheads introduced with the distributed execution mechanisms. Furthermore, as our proposal in this thesis is to assess the partial replication in large-scale systems, we must also establish another baseline using the DBSM. Unfortunately, from the best of our knowledge, there are not comparisons between the DBSM and the traditional replication protocols using real workloads. For that reason, we believe being invaluable a comparison that shows the advantages of the DBSM against the traditional replication protocols, and afterwards, we proceed to the PDBSM evaluation with the guarantee that if it leverages the DBSM, it will certainly be better than the traditional replication protocols.

These experiments are conducted using a simulation tool that combines real and simulated code. The certification, communication and semantic caching protocols are real implementations. Using such a model allows us to evaluate the impact of the design and the implementation decisions of these protocols on the overall performance. In contrast with real systems, this approach allows to set up and run multiple tests with slight variation of configuration parameters, in scenarios with large number of replicas and wide-area networks. When compared with a fully simulated approach, it gives us the opportunity to estimate the resources required by the protocols, since their are real code that interfaces with the simulated environment. Moreover, this model provides us the opportunity to subject real components to fault scenarios which would be difficult to test and replicate in real systems.

Relations	Cardinality	Tuple Length
Customer	= 2880 * c	760 bytes
Country	= 92	70 bytes
Address	= 5760 * c	154 bytes
Orders	= 51840 * c	220 bytes
Order Line	= 155520 * c	132 bytes
Author	= 25 * i	630 bytes
CC XActs	= 51840 * c	80 bytes
Item	= 1K, 10K, 100K...	860 bytes

Table 7.1: TPC-W Relations (*K* is 1000)

The rest of this chapter is organized as follows. Section 7.1 presents the workloads used in the evaluation process. Section 7.2 describes the simulation tool. Section 7.3 describes the integration of the protocol prototypes and the simulation tool. Section 7.4 presents the instantiation and validation of the model. Section 7.5 presents the results obtained.

7.1 Workload Pattern

A key issue in the evaluation of the performance of a database system is the traffic pattern used in benchmarking. In fact, the performance of the **Escada** approach to replication is tightly related with conflicts arising when transactions are concurrently executed in different sites. Therefore, we resort to traffic generated according to industry standard benchmarks, namely, the TPC-C [100] and TPC-W [101]. Both workloads are used to evaluate the semantic caching. For the other experiments, we use the TPC-C because of its OLTP characteristics which have been adopted as our basis for the evaluation of the **Escada** protocols [89, 57].

7.1.1 TPC-W Traffic Characterization

TPC-W mimics an Internet commerce application environment in which a retail store is defined. The customers can visit a web site to look at products, find information, place an order or request the status of an existing order. The traffic is related to a set of operations that simulate the search for products based on author's name, title, subject, date of publishing (i.e., new products) and sale (i.e., best sellers). Or related to a set of operations which involve write activities such as customer registration, order request and administrative tasks. The first set of activities are called *browse* and the second set *order*. The benchmark proposes three distinct mixes of interactions changing the degree of the *browse* and *order* sets as follows:

1. ***Browsing Mix*** The *browse* set represents 95% of the mix and the *order* set represents 5%;
2. ***Shopping Mix*** The *browse* set represents 80% of the mix and the *order* set represents 20%;
3. ***Ordering Mix*** The *browse* set represents 50% of the mix and the *order* set represents 50%;

The database relations and associated information are presented in Table 7.1. Notice that the database must be populated according to the number of clients (*c*) and items available (*i*). The following equation is used to calculate the time the user takes to enter and analyze information:

$$T_t = \text{-}ln(r) * (m) \tag{7.1}$$

Relations	Cardinality	Tuple Length
Warehouse	= w	89 bytes
District	= w * 10	95 bytes
Customer	= w * 30 *K*	655 bytes
History	> w * 30 *K*	46 bytes
Order	> w * 30 *K*	24 bytes
New Order	> w * 9 *K*	8 bytes
Order Line	> w * 300 *K*	54 bytes
Stock	= w * 100 *K*	306 bytes
Item	= 100 *K*	82 bytes

Table 7.2: TPC-C Relations (*K* is 1000)

where r is a random number uniformly distributed between 0 and 1, m is the mean time specified between 7 and 8, inclusive, and T_t is what is called *think-time*.

We use an augmented version of the TPC-W, as explained in Section 7.4, based on the implementation from University of Wisconsin [99]. Notice also that TPC-W is being used only as the basis for a realistic application scenario in order to evaluate **Escada** design decisions and not as a benchmark. The constraints required for throughput, performance, wait time, response time and screen load are not considered here and thus the results are not comparable with other system results obtained with TPC-W.

7.1.2 TPC-C Traffic Characterization

The TPC-C is the industry standard on-line transaction processing (OLTP) benchmark, which mimics a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. The traffic is a mixture of read-only and update intensive transactions. The database relations and associated information are presented in Table 7.2. Notice that, according to TPC-C, an additional warehouse should be configured for each additional 10 clients. The initial sizes of tables are also dependent on the number of configured clients.

A simulated client can request five different transactions types as follows:

1. ***New Order***, adding a new order into the system (with 44% probability of occurrence);
2. ***Payment***, updating the customer's balance, district and warehouse statistics (44%);
3. ***Order Status***, returning a given customer latest order (4%);
4. ***Delivery***, recording the delivery of products (4%);
5. ***Stock Level***, determining the number of recently sold items that have a stock level below a specified threshold (4%).

The equation 7.1 is used to calculate the time the user takes to enter and analyze information according to the means provided by Table 7.3. In this case, unless otherwise stated, we consider the term *think-time* as the sum of the keying time and the value provided by equation 7.1.

This application scenario can easily be extended to consider partial replication. Specifically, we consider horizontal fragmentation of relations according to the warehouse. The rationale for this is that these need to be replicated only locally within the warehouse itself and not globally. Unfortunately, when the system is not prepared to support distributed execution in order to access

Transaction	Keying Time	Mean Think Time
New Order	18 sec.	12 sec
Payment	3 sec.	12 sec.
Order Status	2 sec.	10 sec.
Delivery	2 sec.	5 sec.
Stock Level	2 sec.	5 sec.

Table 7.3: TPC-C keying time and mean of think time

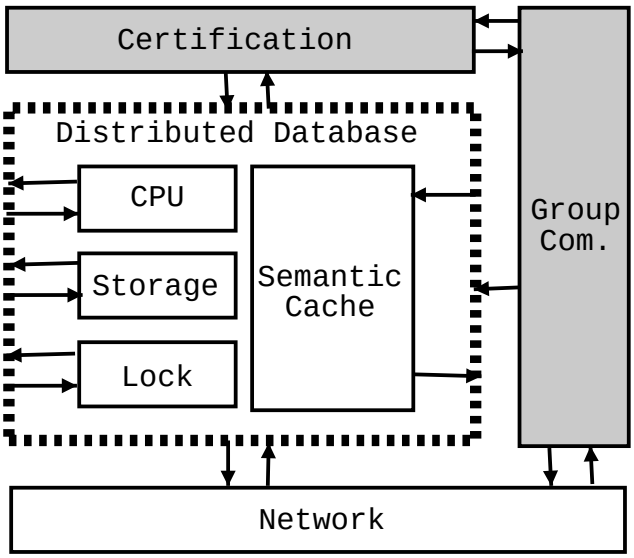


Figure 7.1: Architecture of the model.

the fragments which are not locally available at the initiator site, some relations must be globally replicated. In this case, the *Warehouse*, *Stock*, *Item* and *Customer* relations must be globally replicated to preserve original application semantics in which an order may be serviced by stock from any warehouse or a customer can pay through any warehouse.

As with TPC-W, TPC-C is being used only as the basis for a realistic application scenario in order to evaluate **Escada** design decisions and not as a benchmark. The constraints required for throughput, performance, wait time, response time, screen load and background execution of transactions are not considered here and thus the results are not comparable with other system results obtained with TPC-C.

7.2 Simulation Database Model

In this section, we describe the simulated components of the model which provide a realistic environment for the prototype components under study. These components are depicted as white boxes in Figure 7.1. The simulation model is developed using the Java platform [96] and the Scalable Simulation Framework (SSF) kernel [26].

7.2.1 Database Clients

The database client is attached to a database server and produces streams of transaction requests. The client blocks after each request being issued and until the server answers, thus modeling a

single threaded client process. Following the reception of the answer, the client is then paused for some amount of time (think-time) before issuing the next transaction request.

Each transaction is modeled as a sequence of operations which are scheduled to consume CPU or storage. Finishing the sequence of operations of a transaction is a commit marker. Besides determining the end of a transaction, it also determines the start of the appropriate termination protocol. The contents of each transaction request are read from a previously generated trace file. This trace contains the details of each transaction's operation as follows: (*i*) the set of accessed items; (*ii*) indication whether it is a read or write operation; (*iii*) an offset associated to each accessed item to simulate tables; (*iv*) the amount of CPU used; and (*iv*) think-times between each request. In Section 7.4, we explain how these contents are obtained.

During the run of the simulation, the client logs the time at which a transaction is submitted, the time at which it terminates, the outcome (either abort or commit) and the transaction identifier obtained from the trace file. In this way, the latency, throughput and abort rate of the server can then be computed for one or multiple clients, and for all or just a subclass of the transactions.

7.2.2 Distributed Database Server

The database server handles multiple clients and is modeled as a scheduler, with a collection of resources (i.e., storage and CPUs), a locking policy, a semantic caching and a distributed execution manager. Upon receiving a transaction request each operation is scheduled to execute on the corresponding resource.

Processor operations are scaled according to the configured CPU speed and the operations are executed in a round-robin fashion by any of the configured CPUs. A processor operation can be preempted, namely, to assign the CPU to a higher priority task, like the network protocol handler.

The storage element is used for fetching and storing items in operations, and is defined according to its latency and number of allowed concurrent requests. Furthermore, a cache hit ratio determines the probability of a read request being handled instantaneously without consuming storage resources.

Operations fetching and storing items are also submitted to the lock manager. Depending on the locking policy being used, the execution of the transaction can be blocked between operations (e.g., the multiversion locking policy [9], available on PostgreSQL, does not lock read operations [43]). In addition, locks are atomically acquired before executing any of the operations, and released, also atomically, when the transaction is committed or aborted.

When a requested item is not replicated on the server handling the transaction and is also not available in the semantic caching, the distributed execution manager chooses other sites which can service it. For simplification, there is no optimization is this mechanism and the first site in the set of possibles sites (i.e., sites with a replica of the fragment which contains the requested item) is chosen. It is important to notice that different sets of items can be serviced by different sites according to the designed fragmentation and replication. Furthermore, it is also worth to notice that the semantic caching is used in order to reduce communication among the servers. It does not intend to improve performance of the storage.

When the commit marker is reached, the appropriate termination protocol is started. In case of the database server operating as a DBSM, this involves the identification of read and written items, as well as the values of the written items. Since certification is handled by real code, the representation of item identifiers and values of updated items must accurately correspond to those of real traffic. In Section 7.4, this is described in more detail. For a complete discussion about

termination protocols in the **Escada** project see Chapter 3.

During the simulation run, the usage and length of queues for each resource is logged and can be used to examine in detail the status of the server.

7.2.3 Network Model

The network model used is the SSFNet network simulation [26], which provides a set of components for modeling the Internet protocols and networks at and above the IP packet level of detail. Basic components for link layer and physical layer modeling are also provided. This includes physical network components, such as nets, links, hosts and routers, as well as protocol layers, such as IP, UDP and TCP. Complex network models can be configured using such components to mimic existing networks or to explore particularly large or interesting networks. These components are based on the SSF simulation kernel presented in the next section.

In addition to the configuration of the network, it is also possible to add application components to generate realistic background traffic. Components can also be attached to routers and hosts to log packets. The resulting format is the same used in real networks and thus the log files can be examined using a variety of existing tools.

7.2.4 Simulation Kernel and Centralized Simulation

Generally speaking, the simulation kernel is based on the SSF, which provides a simple infra-structure for discrete-event simulation [26]. It comprises five base interfaces: *Entity*, *Process*, *Event*, *inChannel* and *outChannel*. The entity is an object that owns processes and channels. The communication among the entities in the simulation model occurs writing to the outChannel(s) and reading events from the inChannel(s). When an event arrives in an inChannel, it calls the process responsible to handle the incoming events. Basically, a process has a callback method which is executed one or more times during the simulation.

Each component, depicted in Figure 7.1, is built using an entity that owns a process, and one or more sets of incoming and output channels. The arrows outline the channels and their mapping among the entities. These basic blocks, entities, processes, events and channels are also used by the SSF Network in order to build its components and architecture (e.g., routers, networks and hosts).

In case of the real components, there is an adapter for each one, which is responsible for the integration and is modeled using the basic blocks. This adapter acts as a gateway in a sense that it needs to convert the structure of information used in the real environment to the structure used in the simulation and vice-versa. It must also manage the simulation clock, scheduling appropriate CPU events in order to account for the time consumed processing the real operations.

The main point of this model is the ability to combine simulated environment components, using a discrete-event simulation model, with real code, for those components that are under study. This is the centralized simulation model of [4]. In this model, when an event needs to be handled by real code, such as the submission of a transaction for certification or the reception of a network message by the group communication protocol, the execution of the real component is timed employing a profiling timer and the result is used to mark the simulated CPU busy during the corresponding period, thus preventing other real code events or simulated processing to be attributed concurrently to the same CPU. In other words, this is done computing the amount of time spent in the real components, and scheduling a prioritary CPU event, which means that any CPU activity

that is not related to real code must be preempted and re-scheduled as soon as possible using a round-robing scheduler, and in this manner giving place to the real event.

7.3 Protocol Prototypes

We now present the characteristics of our prototypes and the integration into the simulation model. These prototypes are real code that can be run within the centralized simulation model as well as stand-alone applications on top of a real network.

7.3.1 Distributed Certification

The distributed certification procedure runs in two stages. In the first stage, just after a transaction has been executed and is ready to commit, its associated data is gathered and atomically multicast to the group of replicas. Then, upon delivery, the second stage of the certification procedure is run by all elements of the group and decides if the transaction can or cannot be committed.

In detail, when a transaction enters the committing stage, identifiers of read and written tuples are obtained as well as the identification of the tables. Our prototype assumes that each of these is a 64-bit integer. The values of the written tuples are also obtained. However, in the simulation the size of the tuples (see Table 7.2) is used to calculate the amount of padding data that should be putted in the messages so its size resembles the one obtained in a real system.

However, sometimes the size of the read set may render its multicast impractical. In this case, a single identifier for each table is sent. For what follows, unless otherwise stated, we consider a threshold of 150 items, resorting to a single identifier whenever the number of items is above this value.

7.3.2 Atomic Multicast Protocol

The atomic multicast protocol is based on Groupz [75], a suite of group communication protocols developed at the University of Minho. It is implemented in two layers: a view synchronous multicast protocol and a total order protocol. The bottom layer, the view-synchronous multicast, works in two phases. First, messages are disseminated, taking advantage of IP multicast in local area networks and falling back to unicast in wide-area networks. Then, reliability is ensured by a receiver initiated mechanism [76] and a scalable stability detection protocol [46]. Flow control is performed by a combination of a rate-based mechanism during the first phase and window-based mechanism during the second phase. View synchrony uses a consensus protocol [83] and imposes a negligible overhead during stable operations.

Total order is obtained with a fixed sequencer protocol [11, 58]. In detail, one of the sites issues sequence numbers for messages. Other sites buffer and deliver messages according to the sequence numbers. View synchrony ensures that a single sequencer site is easily chosen and replaced when it fails.

7.4 Model Instantiation and Validation

We are concerned with the overhead imposed by the termination and the communication protocols on transaction processing. It is thus very important that the load imposed by executing transactions is comparable to the measured load of executing the protocols. For that reason, the preferred solution is to use profiling to determine CPU usage by each transaction.

We do this in four steps:[1]

1. A modified version of the benchmark is run on an unmodified database server. Each transaction is augmented with extra queries that return the read set and write set. It is important to notice that despite this process being compatible with the definitions presented in Chapter 3, it is not automatic. On the contrary, the extraction is done by manually changing the queries. This process creates a trace with items to be read and written. In particular, the write values are obtained during execution of the simulation using the write set and the tuple length according to Table 7.2. The CPU and think-time are missing.

2. The original version of the benchmark (without additional queries) is run on an instrumented version of the database server, which records in detail the real time and CPU time consumed by each query. The resulting log is annotated with the name of the transaction.

3. A distribution of CPU times is computed for each transaction type. This is used to generate CPU times for each of the transactions of the trace obtained in step 1.

4. The think-time is injected according to the definitions of the benchmark.

The technique used to obtain the amount of CPU consumed by the execution of each transaction is tightly related to the database engine. In PostgreSQL [43], each process handles a single transaction from start to end. This reduces the problem of profiling a transaction to that of profiling a process in the host operating system.

In detail, we used the CPU timestamp counter which provides accurate measure of elapsed clock cycles. By using a virtualization of the counter for each process [69], we also obtain measurements of process virtual time (i.e., the time elapsed when the process is not scheduled to run is not accounted for). To minimize the influence in the results, the elapsed times are transmitted over the network only after the end of each query (and thus out of the measured interval), along with the text of the query itself.

The time consumed by the transaction's execution is then computed from the logs. By examining the query itself, each transaction is classified. Interestingly, the processor time consumed during commit is almost for all transactions (i.e., less than 2ms). In read-only transactions the real time of the commit operation equals processing time, meaning that no I/O is performed. This does not happen in transactions that update the database. The observation that the amount of I/O during processing is negligible confirms that the database is correctly configured and has a small number of cache misses.

After discarding aborted transactions and the initial 15 minutes, the resulting histogram allows an empirical distribution to be obtained and used later for simulation. However, some transaction classes perform some work conditionally and thus result in bimodal distributions. Therefore, we split each of these in two different classes. The resulting transaction classes can therefore be approximated by an uniform distribution.

[1] In fact, these steps are didactic and in practice some of them can be combined.

For validation we configured our model according to the equipment used for testing. This corresponds to a server with 2 Pentium III at 1GHz processors and with 1GB of RAM. As the cache hit ratio observed is very high, we configure the simulation hit ratio to 1. This means that read items do not directly consume physical resources (CPU or storage), as this is already accounted for in the CPU times as profiled in PostgreSQL. It is important to notice that we are not modeling resources such as buffers or caches.

For storage we used a fiber-channel attached box with 4×36GB SCSI disks in a RAID-5 configuration. The file system used to hold the database (executable and data) is ext3 (Linux version 2.4.21-pre3). Throughput for the storage was determined by running the IOzone disk benchmark [54] on the target system with synchronous writes of 4KB pages and a variable number of concurrent process. This resulted in a maximum throughput of 9.486MBps.

Finally, we use these values and assumptions to instantiate the model and then we validate it by comparing runs of a single site with runs of a real PostgreSQL database [43]. For a detailed discussion about the validation processes and the results obtained, see [89].

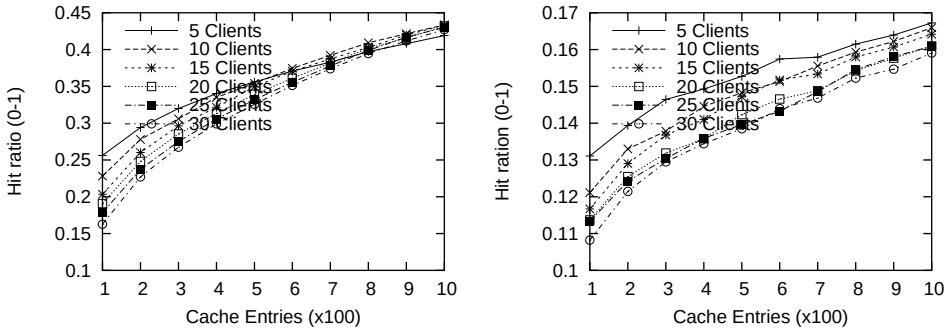
7.5 Experimental Results

In this section, we present our experimental results. First, we analyze our semantic caching approach in a single database machine in order to asses its behavior (i.e., hit ratio and overhead) while changing the number of entries in the cache, the number of clients and the workload. Second, we proceed to the evaluation of the PDBSM and the PDBSM with RAC. Specifically, we evaluate the distributed execution mechanisms and the advantages of our semantic caching approach, considering the results established with the previous experiments. In order to set a baseline, we run some experiments with the assumption that a site is locally able to always complete the execution of a transaction, which avoids possible overheads introduced with the distributed execution mechanisms. Furthermore, as our proposal in this thesis is to assess partial replication in large-scale systems, we must also establish another baseline using the DBSM. Unfortunately, from the best of our knowledge, there are no realistic comparisons between the DBSM and traditional replication protocols. We present such a comparison which will afterwards use as the baseline to benchmark the PDBSM.

The set of the experiments is, therefore, organized in three distinct phases. Initially, we evaluate our semantic caching approach in Section 7.5.1. In Section 7.5.2, we evaluate the DBSM against the traditional replication protocols. In Section 7.5.3, we show the experiments conducted with the PDBSM, namely, we initially use the the assumption that a database site can locally complete the execution of the transaction, and after that, we release this assumption and conduct the set of experiments based on the distributed execution mechanism proposed in Chapter 3.

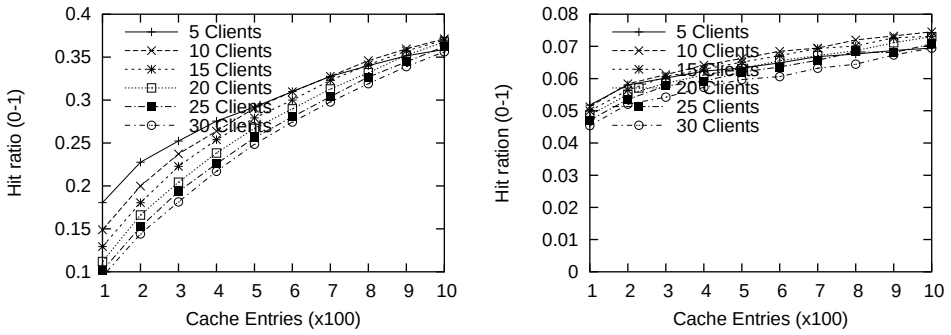
7.5.1 Semantic Caching

The first set of experiments conducted was based on the TPC-W profile and a single database machine. The hit ratio presented in Figures 7.2, 7.3 and 7.4 was measured varying the number of entries in the cache and the number of clients. Observing the graphics, we see that increasing the number of entries also increases the number of hits. The difference in hits among the graphics is justified by the amount of update activities in the mixes. The Figures Figures 7.2.(b), 7.3.(b) and 7.4.(b) show the transaction "Bestsellers" which has a large number of hit ratio for the browsing and shopping mixes and for that reason will be used in the examples of this section. The high



(a) All activities (b) Transaction BestSellers

Figure 7.2: TPC-W Browsing Mix



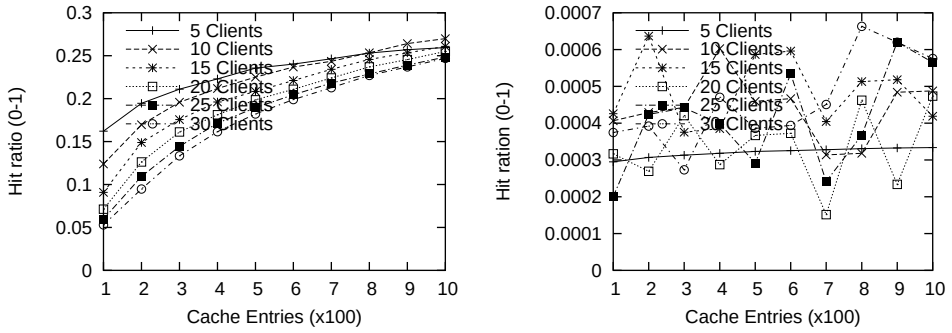
(a) All activities (b) Transaction BestSellers

Figure 7.3: TPC-W Shopping Mix

variance in Figure 7.4.(b), since the hit ratio is below 0.07%, is practically negligible.

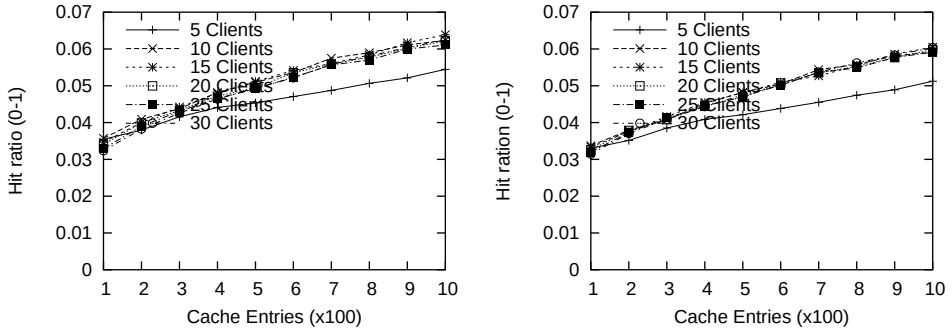
The second set of experiments was based on the TPC-C profile. The hit ratio presented in Figure 7.5 was also measured varying the number of entries in the cache and the number of clients. The same behavior observed with the TPC-W can be noticed here. However, the hit ratio is lower as a consequence of the higher number of update activities of the TPC-C. It is also chosen a single particular transaction, in this case the “New Order”, with the purpose of showing that it follows the same behavior.

Variations in the number of clients produces an interesting and important result. For the same cache size the hit ratio is lower when the number of clients increases. This behavior is explained by the characteristics of the TPC which increases the database size whenever the number of clients increases and uses a non-uniform random distribution to populate the database and to generate queries. For that reason, to achieve the same upper bound hit ratio, we need more entries in the cache. Unfortunately, while increasing the number of entries in the cache we can have a negative impact on processing time since more possible matches must be evaluated for select, insert, delete and update operations. This performance problem is also amplified by our current implementation that is not optimized and uses a coarse grain to control concurrency, which means that the locks are held longer than it is necessary and, as a consequence, the performance is harmed.



(a) All activities (b) Transaction BestSellers

Figure 7.4: TPC-W Ordering Mix



(a) All activities (b) Transaction New Order

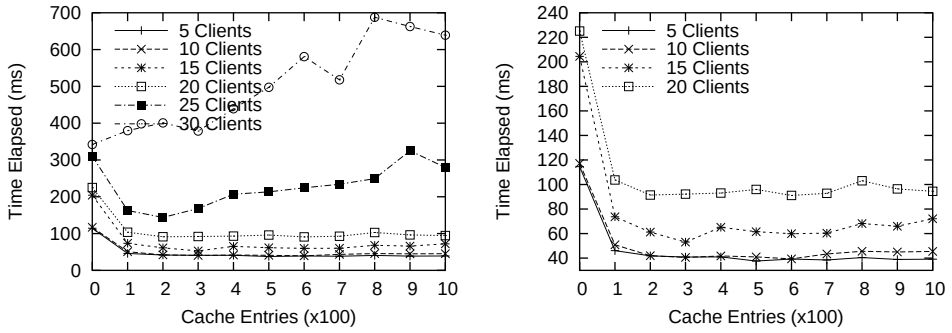
Figure 7.5: TPC-C

This problem is observable in Figure 7.6 where we present the elapsed time for the transaction[2] "Bestsellers" in scenarios with different configurations of cache. The elapsed time to answer the queries of this transaction is drastically reduced with the use of the cache when the number of clients is below 20. In some cases, this reduction is higher than 100 ms, from around 200 ms to 100 ms (i.e., 50%), which is a great improvement. However, with 25 clients this difference is not quite visible and with 30 clients the problems are noticeable.

In the case of the TPC-C, the situation becomes worse. Basically, the cache hits of the TPC-C comes from the transaction "New Order" and are lower than the hits provided by the TPC-W, that is, the values are below 7% regardless of the number of clients and entries in the cache. This scenario combined with the characteristics of the transaction contributes to the unaceptable delays presented in Figure 7.7. The transactions are composed of simple queries which do not manipulate high amounts of data neither are CPU bound. For that reason, the use of the cache in this situation is not recommended.

In spite of that, observing the experiments we see that the time spent to search a cache with 1000 entries using the hierarchical filter and test 19 entries is around 4 ms, which is a good value. For instance, considering a WAN (see Table 7.4) with latency of 30 ms plus the time necessary

[2]Instead of adopting the term web interaction and thus being compatible with the nomenclature proposed by the TPC, we prefer to use the term transaction since it is more suitable to our context.



(a) Problems and Profi ts

(b) Profi ts

Figure 7.6: TPC-W Shopping Mix (Bestsellers)

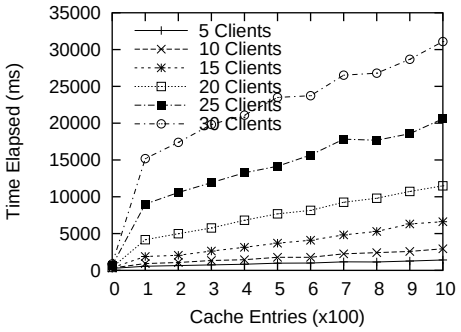
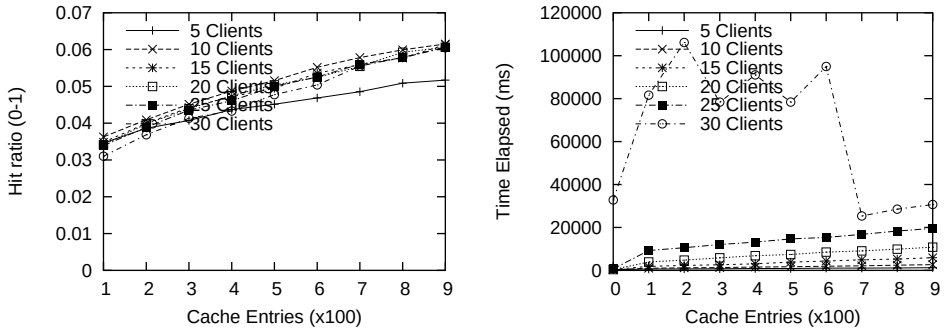


Figure 7.7: TPC-C New Order

to process the queries and transfer the requests and results, it is clear that our semantic caching is extremely useful. Unfortunately, when we allow all queries sent to the database to be stored in the cache, this incurs the overhead of testing entries that will produce a match and entries that will not. The number of tests increases and therefore the contention could turn its use impractical. It is therefore important previous knowledge about the application semantics in order to cache only the queries which have high probability of occurrence.

In what follows, we conduct some experiments storing in cache all queries without further restrictions, in a scenario with low bandwidth, in order to figure out if the semantic caching it is a profitable solution even when the overhead and the contention problems are not circumvented. In other words, we want to evaluate if without restricting the entries in the cache, our semantic caching approach can still compensate the low bandwidth. In particular, this is done with clients communicating with the server through a network link of 512 Kpbs.

In Figure 7.8, we present the results of an evaluation of the TPC-C simulating a network with 512 Kbps. The performance does not increase and for that reason, we can conclude that just the queries that have a high probability of generating cache hits or the queries that consume a high amount of resources (e.g., storage, CPU or network bandwidth) are suitable to be cached and tested against the semantic caching.



(a) All activities

(b) Transaction New Order Time

Figure 7.8: TPC-C with 512Kbps

Parameters	Configuration	Values
CPU	Number	2
	Time Slice	0.5 ms
	Policy	Round Robin
	Speed	Pentium III 1 GHz
Storage	Parallel Requests	1
	Throughput	9.486 MBps
	Blocks	4 KB
WAN	Bandwidth	according to the traffic
	Latency	30 ms
LAN	Bandwidth	1 Gbps
	Latency	1 ms
Database	Lock Manager	multi-version (PostgreSQL)
	Table Threshold	150
	Cache Hit Ratio	$\backsimeq$100%

Table 7.4: Configuration Parameters - Resources

7.5.2 DBSM

We now proceed with the experiments of the DBSM. Our goal here it is twofold. First, we show that instead of doing an upgrade of a central database to scale up a system, it can be worth using commodity machines and the DBSM. Second, we run some experiments with a distributed database using an ideal implementation of the distributed locking in order to establish a baseline. In Tables 7.4, 7.5 and 7.6, we summarize important assumptions and values used to configure the simulations. Unless otherwise stated, the information presented in these tables are implicit assumed for what follows.

In Figure 7.9, notice that the 1 CPU system handles less than 3000 tpm. Latency, as shown in Figure 7.9(b), grows due to queuing. In contrast, the 3 CPU system scales linearly and thus there is no increase in latency. The replicated system also allows for a linear increase in throughput. Unfortunately, the number of aborts increases with the replicated system which is a problem of the optimistic execution and serialization.

Figure 7.10(a) shows the average usage of the involved CPUs. This justifies the throughput and latency results, showing that a single CPU is a bottleneck, due to interaction of variability and locking mechanisms. Furthermore, the similarity between the results obtained with the 3 CPUs

Transactions	Distribution	Mean (ns)	Stdv
Delivery	Normal	93518611	7354168
New Order	Normal	24520044	7266947
Order Status 01	Normal	1608646	582965
Order Status 02	Uniform	1668984	254385
Payment 01	Uniform	7384053	1703256
Payment 02	Uniform	7109045	1595916
Stock Level	Normal	19230043	1407436

Table 7.5: Configuration Parameters - CPU's consumption per transaction

Transactions	Distribution	Mean (ns)	Stdv
Delivery	Uniform	5655792	13730916
New Order	Uniform	5907053	5506115
Order Status 01	Uniform	4820850	1320468
Order Status 02	Uniform	4537986	861376
Payment 01	Uniform	6538242	10390480
Payment 02	Uniform	6227296	2111338
Stock Level	Uniform	4437135	875610

Table 7.6: Configuration Parameters - Idle time per transaction

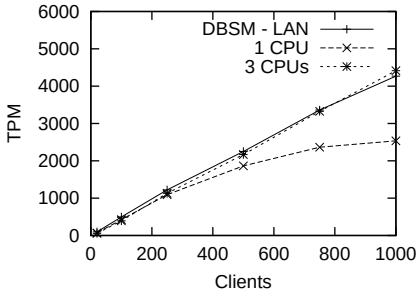
system and the replicated system have shown that there is a low overhead of the termination and atomic multicast protocols.

To establish a comparison between the DBSM and the distributed locking protocols, we use a set of experiments based on the TPC-C workload in a WAN with 9 sites, where the fragments are replicated according to Figure 7.11. The clients access the first site in each sub-network. The letters A, B, C, G represent distinct fragments which are replicated following a ring configuration. To avoid repeating the same graphics here, and since the DBSM results will be used as a baseline for the next section, these values are presented there.

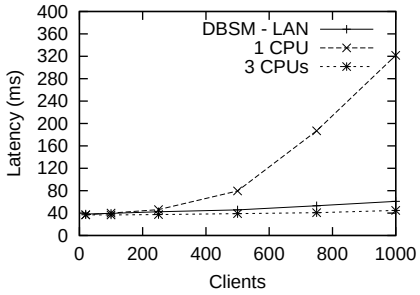
In Figure 7.12, we show that the DBSM can circumvent the scalability problems of the traditional replication protocols, preserving the strong consistency and the ability to update the database from any replica. Although the DBSM does not outperform the Ideal Locking System, it is close enough. Furthermore, it is also important to notice that this traditional replication protocol uses an ideal implementation and whatever the real implementation is, it will present a worse behavior: (*i*) all locks are acquired atomically, avoiding the complexity of deadlock detection; (*i*) a delay, that equals the latency to propagate a message between two points and that represents the time required to the lock acquisition, is applied to the transaction after the locks being granted; (*iii*) upon the commit being issued and before the locks being released, another delay is applied, in this case, representing the time necessary to propagate the write values to all replicas. For example, we do not consider deadlock detection and resolution, that according to [42] is the major responsible for the problems of traditional replication. Furthermore, we adopt a centralized lock manager disregarding thus any fault tolerance requirements and associated overhead.

7.5.3 PDBSM

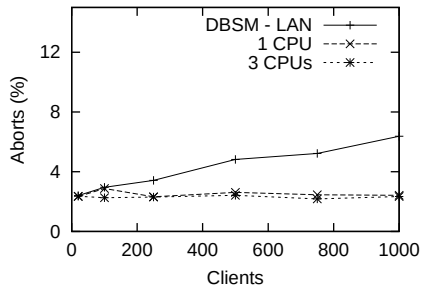
In this section, we analyze the PDBSM. First, we run some experiments with the DBSM to establish a baseline. Second, we run some experiments with PDBSM in which we assume that the queries are fully serviced by the database sites that accept the requests. Basically, this also done



(a) Throughput



(b) Latency



(c) Abort rate

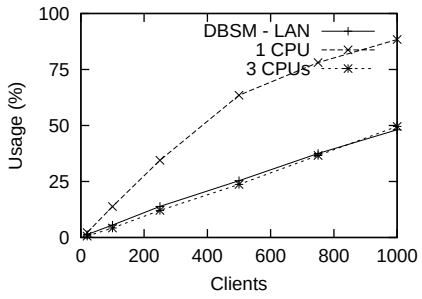
Figure 7.9: DBSM - Performance results

in order to establish a baseline. Finally, we release this assumption to understand the impact of the distributed query processing on the overall performance. Before conducting the experiments, we present an analytical analysis of the **Escada** protocols (i.e., PDBSM and PDBSM with RAC), which allows us to comprehend their benefits and outcomes.

Resource Analysis

We outline in this section an analysis of resource consumption, namely, bandwidth and storage, for the termination protocols presented. To compare the protocols we consider a network setting that privileges access locality. The network is composed of a wide area network (WAN) with moderate bandwidth and high latency, aggregating several local area networks (LANs) with much higher bandwidth and much lower latency. We assume that all the replicas of a fragment, which is not fully replicated in order to preserve the application semantic are in a LAN. Thus we admit that the bandwidth requirements for data propagation between copies of the same fragment are irrelevant when compared with the effect of traffic crossing long distance links. It is worth noticing that this assumption is considered to facilitate our analytical process and it is released in the following sections. Even though, the analysis presented is rather important since such a setting is close enough to a well planned and desired replication setting. Unfortunately, it is not always possible to achieve such a scenario, and for that reason, we run a set of simulations to understand resource consumption and evaluate the overall performance.

The transactions read set, write set, and write values have been divided in two subsets: (i) a subset of fully replicated data items called RS_G, WS_G and WV_G, and (ii) a subset containing



(a) CPU

Figure 7.10: DBSM - Resource usage

partially replicated items called RS_L, WS_L and WV_L. We represent by RAC the bandwidth required by the RAC protocol.

The following formulas present the required WAN bandwidth for the termination protocols proposed:

$$\text{DBSM} \equiv RS_G + WS_G + WV_G + RS_L + WS_L + WV_L \tag{7.2}$$

$$\text{PDBSM} \equiv RS_G + WS_G + WV_G + RS_L + WS_L \tag{7.3}$$

$$\text{PDBSMRAC} \equiv RS_G + WS_G + WV_G + RAC \tag{7.4}$$

Comparing formulas 7.2 and 7.3, it can be seen that in the proposed network setting the PDBSM protocol using independent certification has a lower bandwidth consumption as some write values, .i.e. WV_L, never leave LANs and thus never cross long distance links. Basically, this reduction is an inherent characteristic of partial replication protocols, which attempt to reduce the ratio between global and local information.

In contrast, from formulas 7.2 and 7.4, the PDBSM protocol using coordinated certification is expected to outperform the DBSM protocol as long as the RAC's required bandwidth does not exceed the requirements for propagating the read and write sets, and the write values of partially replicated fragments:

$$\text{PDBSMRAC} < \text{DBSM} \Rightarrow RAC < RS_L + WS_L + WV_L$$

Finally, the comparison of formulas 7.3 and 7.4, reveal that the coordinated certification is preferable when the bandwidth required for the RAC does not exceed that for transmitting the read and write sets. Specifically, we consider a simulation model in which for every database site

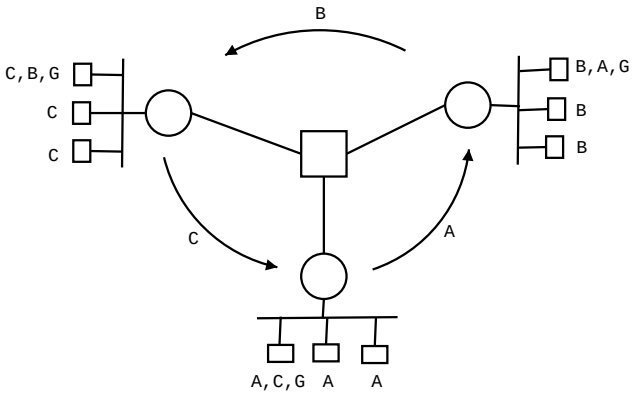


Figure 7.11: Partial replication and its fragments

there are some sites collocated in the same LAN and other sites (say, m) at the other end of the long distance link. In this case, using multisend to broadcast messages, every atomic broadcast implies that m times the estimated atomic broadcast message size crosses the long distance link (we ignore the message from the coordinator establishing the message order as it will be the same in all protocols).

The information presented leads to conclude that DBSM consumes more network resources than PDBSM and that PDBSM should perform better when there is contention in the network.

Having analyzed the required bandwidths, we now consider the expected latencies of the protocols we are evaluating. Every protocol starts broadcasting the transaction using the fast atomic broadcast protocol, and as this protocol propagates the messages concurrently with the ordering mechanism, we expect that it will mask the differences in latency that should happen due to message size and in some cases to the designed termination protocol. For example, the PDBSM with RAC protocol regardless of being the implementation of RAC offering the lowest cost in terms of latency [82], incurs in the additional overhead of the RAC and could present higher latencies. Using the fast atomic broadcast protocol, in a network without network congestion, we intend to reduce or even eliminate possible latency differences.

Furthermore, while observing the storage, it is possible to conclude that partial replication, both PDBSM and PDBSM with RAC, outperforms the original DBSM approach, since each site does not need to be concerned with all the write values and thus reducing storage activities and possible bottlenecks. To the growth of the system, this is an important consideration. The reasoning is that expansions are usually realized with the addition of more database sites, what would increase the number of simultaneous transactions and therefore the activities of each individual storage. However, the storages may not have the capacity to handle the additional activities and may become bottlenecks. The same formulas used to analyze the bandwidth consumption can also be used to determine the storage activities.

Results

The set of experiments are based on the TPC-C workload in a WAN with 9 sites, where the fragments are replicated according to Figure 7.11. Initially, we suppose that each site has all the fragments to answer the requests and afterwards we release this assumption.

Until 750 clients, notice, according to Figure 7.12, that the PDBSM, PDBSM with RAC and DBSM perform similarly. The explanation for this behavior is that the amount of information

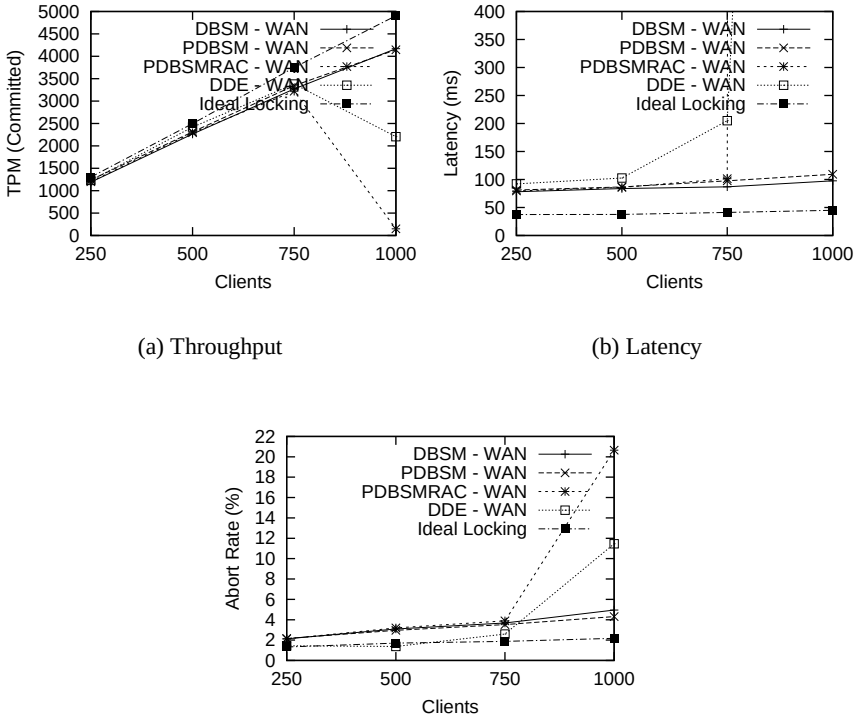
which needs to be globally replicated is high and there is no contention in the network. However, it would be expected that the additional step of the RAC would increase the latency and reduce the throughput as the network does not represent a bottleneck. In this case, the use of the FastAtomic Broadcast contributes to compensate this overhead.

In Figure 7.13, regarding CPU usage, the RAC protocol does not present a problem. The amount of CPU required is similar to the DBSM and the PDBSM. Regarding the bandwidth consumption, we see that the amount of bandwidth required by the PDBSM with RAC is similar to the DBSM and higher than the PDBSM, which is easily explained by the fact of the RAC introducing additional messages. However, it is important to notice that the amount of bandwidth required by the PDBSM is lower than the amount of bandwidth required by the DBSM and this result could be improved if the application semantics allowed. The difference is around 115 Kbps when the bandwidth required by the DBSM is around 1.6 Mbps (i.e., for 750 clients). In Figure 7.14, we present a simulation that increases the size of the tuples, i.e. doubles the size, simulating for example changes in the character encoding to assert our assumption about the application semantics. The difference with the new tuples is around 530 Kbps when the bandwidth required by the DBSM is around 2.9 Mbps (i.e., for 750 clients).

Unfortunately, the PDBSM with RAC degenerates when the number of clients increases above 750. It has contention problems with 1000 clients, since the certification queue and the lock queue present high values, with an average of 90 transactions waiting to be serviced. This is a problem of the additional step introduced with the RAC and of the certification 's overhead which increase the time to process a transaction and generate this behavior.

Finally, we expect to indeed reduce the bandwidth consumption, releasing the assumption that each site can locally complete the execution of a transaction. For that reason, we conduct a set of experiments using the PDBSM, i.e. DDE (Distributed Execution), in order to exploit possible benefits of the partial replication. The PDBSM behaves similar to the other experiments until 500 clients and after that, it presents the same contention problems, since the additional steps of the distributed execution (i.e., contact remote sites, gather the results and stabilize) increases the time to process the transactions. It has been observed certification and lock queues. It is important to notice that the amount of bandwidth required by the PDBSM with the distributed execution is lower, which can be observed in Figure 7.13. The difference when compared to the DBSM is around 837 Kbps when the bandwidth required by the DBSM is around 1.2 Mbps (i.e., for 500 clients). This is simple explained by the nonexistence of global relations and the locality allowed by the TPC-C.

These set of experiments allow us to conclude that the PDBSM augmented with the distributed execution, in what fallows just PDBSM unless otherwise stated, represents an excellent improvement when compared to the DBSM. The bandwidth required by PDBSM is reduced around 69%. Unfortunatelly, it presents contention problems while increasing the number of clients, which could be avoided using a flow control mechanism. Roughly speaking, the idea of the flow control mechanism is to restrict the number of concurrent requests inside the system which avoids its collapse and therefore sustains the maximum throughput. However, as a side effect probably the client's response time would increase. In order to sustain the throughput and also avoid this undesirable latency, we could use our semantic caching approach. In this case, it would reduce the needs of the distributed execution. Unfortunatelly, our current implementation has performance problems as explained before and a better implementation is required. Regarding the others alternatives of the PDBSM, we can conclude that the RAC is not a good approach since its additionals steps introduce latency which implies in the collapse of the system while increasing the number of clients. Furthermore, it does not reduce the required bandwidth. The PDBSM with the no-voting protocol is the approach recommend since it does not show the problems that affect the PDBSM



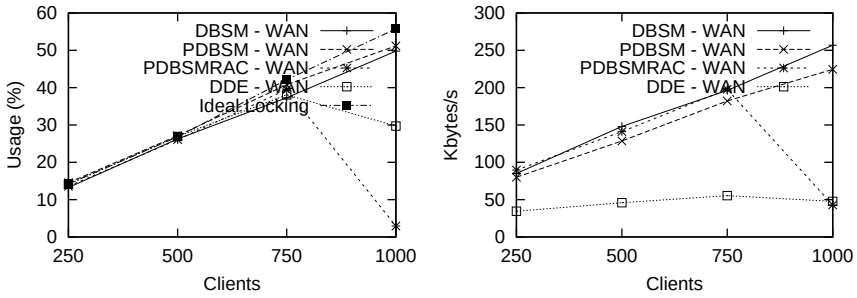
(a) Throughput

(b) Latency

(c) Abort rate

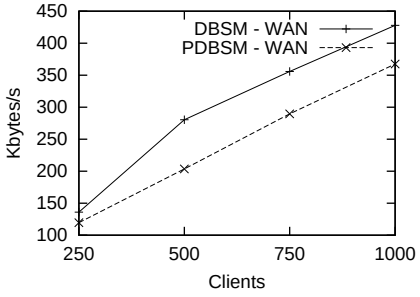
Figure 7.12: PDBSM - Performance results

with RAC.



(a) CPU (b) Network

Figure 7.13: PDBSM - Resource usage



(a) Network

Figure 7.14: PDBSM - Network Bandwidth

Chapter 8

Conclusion

Industries, hospitals, public organizations and commerce, for instance, conduct their activities based on information which is gathered from different sources and ultimately is stored into a database. From a technological point of view, the databases are usually considered a core component, which means that a failure may generate a tremendous negative impact on the system as a whole. For those reasons, it is important to have techniques and mechanisms to deal with failures. Database replication is an invaluable technique to implement fault-tolerant databases, being also frequently used to improve performance. Unfortunately, when strong consistency and the ability to update the database at any of the replicas are considered the traditional replication protocols do not scale up.

Database replication based on group communication, appears as a promise to circumvent these problems. Briefly, the DBSM allows an optimistic local execution, postponing the interaction with remote concurrent transactions. That is, each transaction request is optimistically executed by a single site and interaction with other sites is only initiated after the commit request. Upon receiving the commit, the outcome of the transaction (i.e., "write" values, "read" and "write" sets) is propagated to all replicas using atomic multicast. A certification procedure is run upon delivery by all sites to determine conflicts with other concurrently executed transactions, and thus whether the transaction should be committed or aborted. Atomic multicast guarantees that all sites deliver the outcome of the transaction in the same order. In the case of a conflict, the certification uses this order to decide which transaction commits or aborts. The determinism of the certification ensures a strong consistency and as the deterministic execution is confined to the certification, no restrictions impairing performance are imposed on scheduling during the execution stage.

This is the context of the **Escada** project, which aims to design and implement transaction replication mechanisms suited for large scale distributed systems. In particular, the project intends to exploit partial replication techniques to provide strong consistency criteria without introducing significant synchronization and performance overheads. In this thesis, we have augmented the **Escada** project with a distributed query processing protocol, releasing the assumption that all the fragments accessed by a transaction are located at a single site.

In order to do that, we have developed a distributed query processing mechanism which uses a two step-optimization. Roughly, the two step-optimization facilitates the integration of the distributed query processing into a centralized database management system, allowing the query to be locally optimized without further modifications to the local optimization engine. After the local optimization, which corresponds to the first step, the second step must decide where the pre-processed query must have its operations executed. In this case, for each operation we have used a

simple approach which chooses the first correct site that is able to handle the request. The information about the fragments and replicas is fully replicated among the catalogs. The following approach has been proposed in order to update the catalogs: (*i*) each site is responsible to compute the statistics for the local fragments and periodically these information is propagated; (*ii*) some operations such as the creation, deletion or modification of object' structures (e.g., creation of a table or modification of its attributes) are applied synchronously which avoids inconsistency among the replicas. This process takes advantage of the atomic multicast used to serialize the transactions, avoiding the overhead of another communication step.

Combined with the two-step optimization, we also have developed a mechanism to the distributed execution that mimics a nested transaction: (*i*) the initiator (i.e., the site used by the client to send transaction's request) can spawn one subtransaction per site; (*ii*) only the initiator can spawn subtransactions, which avoids the possibilities of deadlocks inside the same transaction; (*iii*) subtransactions execute optimistically at remote sites and the concurrency control mechanism of the initiator site controls its own transactions; (*iv*) upon the initiator abort, all the subtransactions are also aborted; (*v*) upon a child abort, all the subtransactions are also aborted.

In order to minimize the impact of the distributed execution on the overall performance, which may increase resource usage and mainly bandwidth consumption, we have built a distributed cache mechanism. It is based on semantic entries, which means that instead of using tuples or pages to identify the entries in cache, the predicates of the queries are used as identifiers and the cache is populated using the results of the queries. Doing this, we have avoided the management overhead of the tuples, which usually involves retrieval, update and replacement per tuple. In contrast to page caching, we also have reduced the management overhead and further overcome the problem of space consumption, which is a consequence of the page fixed size, while disregarding the size of the result set and always allocating pages. The cache is populated using the results of SPJ queries, which is a combination of a set of (s)elect, (p)rojection and (j)oin operations. This class of queries allows autonomous recomputation, which means that we can update the entries in cache using just the results of the queries and the information available in cache. This is an important point since it avoids to contact remote sites to answer features requests. However, our approach could be easily extended in order to populate the cache with other classes of queries. In fact, it is not required further modifications since our algorithm is generic enough to detect that some queries could not be autonomously recomputed and hence for each these queries the algorithm identifies if the updates affect it and in a positive case invalidates it, otherwise nothing is done.

We also have provided the correct integration of the distributed cache with the PDBSM, preserving its consistency criteria. In other words, we have considered a shared cache per database site that is autonomously updated or invalidated upon transaction certification and during transaction execution, the possible changes are visible per transaction. Basically, to manage the distributed caches among the sites, we have relied on the propagation of the outcome of the transaction.

We have proposed the use of a smart certification, which means sending the queries that manipulates the relations that are referenced instead of their tuples in consequence of the amount of information that needs to be transfered and using the queries in an attempt to reduce the number of aborts due to the use of entire relations. Roughly, the certification procedure attempts to see if the conflicting updates are relevant to the queries, in which case the transaction is aborted, otherwise it is committed.

It is important to notice that the distributed execution and the semantic cache have been developed as prototypes and in order to integrate them into a real database system, we have analyzed and suggested the steps to do that using the PostgreSQL as the target database. First of all, we have proposed an algorithm to extract the "write" values, "read" and "write" sets. Although it

may seem a simple process, the extraction of the read set involves complex procedures. The main reason is that some features (e.g., nested subquery blocks, grouping, control over duplicates and quantifiers) cannot be mapped to the select, project, join subset of the relational algebra, which is easily manipulated. We have suggested to extended the PostgreSQL's grammar to be possible to create fragments and to extend its rule system, which allows to specify actions that must be done instead or with the original request, according to the operation and when a pre-defined condition evaluates to true. In our case, the extended rule is used to allow references to relations prior to the fragmentation, called logical relations, and in a transparent manner establishes the map among them and their fragments. Finally, regardless of the details involved in the extension of the PostgreSQL, we also have identified that the consistency criteria, *one-copy serializability*, claimed by the DBSM is not achievable, unless the indices and the relations being considered as read and written information. We also have suggested different alternatives to deal with possible high amounts of read set: (*i*) use the snapshot isolation level, (*ii*) send the read set by comprehension (e.g., using SQL predicates or a reference to the relation), (*iii*) propagate just the write values and rely on the initiator site to decide about its fate, that is, abort or commit.

Furthermore, we have presented and analyzed two different termination protocols and their impact on the overall system. The first one, a direct contribution of this thesis, called simply PDBSM, works similar to the DBSM which means that each machine can independently decide on the outcome of the transaction without requiring a voting phase. In order to do that the "read" and "write" sets are multicast, but the "write" values are just propagated to the sites replicating the involved fragments. The second one, originally proposed in the **Escada** project, called PDBSM with RAC, propagates the "write" values, 'read" and "write" sets to the sites replicating the involved fragments, which means that a single site may not unilaterally decide about the outcome of a transaction and therefore a voting phase is required.

The set of experiments conducted with the semantic caching without combine it with the replication have shown that the time spent to search a cache with 1000 entries using the hierarchical filter and test 19 entries is around 4 ms, which is a good value. For instance, considering a WAN (see Table 7.4) with latency of 30 ms plus the time necessary to process the queries and transfer the requests and results, it is clear that our semantic caching is extremely useful. Unfortunately, when we allow all queries sent to the database to be stored in the cache, this incurs the overhead of testing entries that will produce a match and entries that will not. The number of tests increases and therefore the contention could turn its use impractical. It is therefore important previous knowledge about the application semantics in order to cache only the queries which have high probability of occurrence. The hit ratios observed with the TPC were low, around 3% – 7%, which makes the cache mechanism infeasible in order to improve performance. The elapsed time to answer the queries of the transaction "Bestsellers" (hit ratio 10% - 16%) in the TPC-W is drastically reduced with the use of the cache. In some cases, this reduction is higher than 100 ms seconds which is a great improvement.

In order to establish a baseline, we have shown with a set of experiments using realistic workloads (i.e., TPC-C) that DBSM can indeed circumvent the scalability problems of the traditional replication protocols, preserving the strong consistency and the ability to update the database from any replica. In LAN environments, this new approach of replication has presented throughput and latency equivalent to a single host with the advantage that we might easily combine commodity machines to scale up the system instead of upgrading the host.

In large scale distributed systems with several replicas distributed in a wide area network, fully replicated databases have not seemed to be suitable and we have evaluated partially replicated databases. Our evaluation has shown that the PDBSM reduces the bandwidth and storage usage. For instance, the reduction of bandwidth is around 115 Kbps when the bandwidth required by

the DBSM is around 1.6 Mbps (i.e., for 750 clients). Unfortunately, the PDBSM with RAC has contention problems. This is a consequence of the additional step introduced with the RAC and of the certification's overhead which increase the time to process the transactions and generate this behavior.

The PDBSM augmented with a distributed execution mechanism indeed reduced the bandwidth, the reduction is around 837 Kbps when the bandwidth required by the DBSM is around 1.2 Mbps (i.e., for 500 clients). Unfortunately, it also has contention problems, since the additional steps of the distributed execution (i.e., contact remote sites, gather the results and stabilize) increases the time to process the transactions. It has been observed certification and lock queues.

From these experiments we have concluded that the PDBSM augmented with the distributed execution, in what fallows just PDBSM unless otherwise stated, represents an excellent improvement when compared to the DBSM. It is an interesting technique in order to provide database replication in large scale distributed systems with several replicas distributed in a wide area network. In contrast to fully replicated databases, it reduces the required network bandwidth and storage usage. The bandwidth required by PDBSM is reduced around 69%. Unfortunatelly, it presents contention problems while increasing the number of clients, which could be avoided using a flow control mechanism. Roughly speaking, the idea of the flow control mechanism is to restrict the number of concurrent requests inside the system which avoids its collapse and therefore sustains the maximum throughput. However, as a side effect probably the client's response time would increase. In order to sustain the throughput and also avoid this undesirable latency, we could use our semantic caching approach. In this case, it would reduce the needs of the distributed execution. Unfortunatelly, our current implementation has performance problems as explained before and a better implementation is required. Regarding the others alternatives of the PDBSM, we have concluded that the RAC is not a good approach since its additionals steps introduce latency which implies in the collapse of the system while increasing the number of clients. Furthermore, it does not reduce the required bandwidth. The PDBSM with the no-voting protocol is the approach recommend since it does not show the problems that affect the PDBSM with RAC. The simplicity of the protocol allows it to scale up without further overhead when compared to the PDBSM with RAC. However, it is important to notice that its advantages are highly dependent of the application's semantic, which means that the application must allow to exploit the benefits of partial replication.

From the previous conclusions, it simple to see that this work presents excellent results which show the benefits of using the PDBSM in large scale distributed systems. From the knowledge gained through this work, we realize that there are some important points that could be analyzed and developed in order to further improve the benefits of the PDBSM:

- Regarding the termination protocols, it would be interesting to evaluate other group communication facilities such as the semantic broadcast in order to take advantage of the semantic of the application and hence avoiding additional communication steps; or the generic broadcast avoiding to totally order transactions that are not concurrent, which usually increases transaction's latency.

- Integrate the current prototype into the PostgreSQL according to the ideas presented and improve the performance of the semantic cache avoiding to hold locks longer than it is necessary.

- Analyze and develop possible distributed deadlock algorithms that exploit the characteristics of the database replication based on group communication.

- Implement and evaluate the smart certification proposed in a attempt to reduce the number of aborts.

Appendix A

The completed set of algorithms discussed in Chapter 3 is presented as follows:

module $DistributedExecutionInPDBSM$

exports all

definitions

types

33.0 $nodeOperation :: classOperation : enumOperations$
.1 $relationId : relation$

.2 inv $opr \triangleq$
.3 $(opr.classOperation \in \{\text{SELECT}, \text{INSERT}, \text{UPDATE}, \text{DELETE}\} \wedge opr.relationId \neq [])\ \vee$
.4 $(opr.classOperation \in \{\text{COMMIT}, \text{ABORT}\} \wedge opr.relationId = [])$;

34.0 $enumOperations = \text{SELECT} \mid \text{INSERT} \mid \text{UPDATE} \mid \text{DELETE} \mid$
.1 $\text{BEGIN} \mid \text{COMMIT} \mid \text{ABORT}$;

35.0 $enumCertification = \text{NoVote} \mid \text{Vote}$;

36.0 $transControl = transId \xrightarrow{m} controlInformation$;

37.0 $controlInformation :: access : relation \xrightarrow{m} site$
.1 $dataSet : storedSets$
.2 $siteOrig : site$

.3 inv $ctr \triangleq (\forall r \in \text{dom } ctr.access \cdot r \neq []) \wedge$
.4 $(\forall s \in \text{rng } ctr.access \cdot s \neq [])$;

38.0 $transOrig :: idOrig : transId$
.1 $siteOrig : site$
.2 $siteExecution : site$

.3 inv $tx \triangleq (tx.siteOrig \neq []) \wedge (tx.idOrig \geq 0)$;

39.0 $storedSets :: rs : relation \xrightarrow{m} pk\text{-set}$
.1 $ws : relation \xrightarrow{m} pk\text{-set}$
.2 $wv : relation \xrightarrow{m} tuple\text{-set}$

.3 inv $sset \triangleq$
.4 $(\forall\, idrs \in \mathsf{rng}\ sset.rs \cdot idrs \neq \{\}) \wedge$
.5 $(\forall\, idws \in \mathsf{rng}\ sset.ws \cdot idws \neq \{\}) \wedge$
.6 $(\forall\, idwv \in \mathsf{rng}\ sset.wv \cdot idwv \neq \{\});$

40.0 $collectedSets :: rs : relation \xrightarrow{m} pk\text{-set}$
.1 $ws : relation \xrightarrow{m} pk\text{-set}$
.2 $wv : relation \xrightarrow{m} tuple\text{-set}$

.3 inv $cset \triangleq$
.4 $(\forall\, idrs \in \mathsf{rng}\ cset.rs \cdot idrs \neq \{\}) \wedge$
.5 $(\forall\, idws \in \mathsf{rng}\ cset.ws \cdot idws \neq \{\}) \wedge$
.6 $(\forall\, idwv \in \mathsf{rng}\ cset.wv \cdot idwv \neq \{\});$

41.0 $replicas = relation \xrightarrow{m} site\text{-set}$

.1 inv $rep \triangleq$
.2 $(\forall\, s \in \mathsf{rng}\ rep \cdot s \neq \{\}) \wedge$
.3 $(\forall\, sl \in \bigcup \mathsf{rng}\ rep \cdot sl \neq []) \wedge$
.4 $(\forall\, r \in \mathsf{dom}\ rep \cdot r \neq []);$

42.0 $tuple :: attpk : pk$
.1 $data : \mathsf{token}$

43.0 $transId = id;$

44.0 $pk = id;$

45.0 $site = \mathsf{char}^*;$

46.0 $relation = \mathsf{char}^*;$

47.0 $id = \mathbb{N}$

48.0 state DDb of
.1 $transDb : transControl$
.2 $repDb : replicas$
.3 $certMode : enumCertification$
.4 $localDb : site$
.5 $transMapDb : transId \xrightarrow{m} transId$

.6 inv $ddb \triangleq$
.7 $(\forall\, tid \in \mathsf{dom}\ ddb.transDb \cdot tid \in \mathsf{dom}\ ddb.transMapDb) \wedge$
.8 (let $tid \in \mathsf{dom}\ ddb.transDb$ in
.9 $(\forall\, rra \in \mathsf{dom}\ ddb.transDb\,(tid).access \cdot rra \in \mathsf{dom}\ ddb.repDb) \wedge$
.10 $(\forall\, rrb \in \mathsf{dom}\ ddb.transDb\,(tid).dataSet.rs \cdot rrb \in \mathsf{dom}\ ddb.repDb) \wedge$
.11 $(\forall\, rrc \in \mathsf{dom}\ ddb.transDb\,(tid).dataSet.ws \cdot rrc \in \mathsf{dom}\ ddb.repDb) \wedge$
.12 $(\forall\, rrd \in \mathsf{dom}\ ddb.transDb\,(tid).dataSet.wv \cdot rrd \in \mathsf{dom}\ ddb.repDb) \wedge$
.13 $(\forall\, ssa \in \mathsf{rng}\ ddb.transDb\,(tid).access \cdot ssa \in \bigcup \mathsf{rng}\ ddb.repDb) \wedge$
.14 $(ddb.transDb\,(tid).siteOrig \in \bigcup \mathsf{rng}\ ddb.repDb))$
.15 end

functions

49.0 $regSite : transOrig \times site \to transOrig$
.1 $regSite\,(tx, s) \triangleq$
.2 mk-$transOrig\,(tx.idOrig, tx.siteOrig, s)$;

50.0 $regControl : transOrig \times transControl \to transControl$
.1 $regControl\,(tx, txControl) \triangleq$
.2 $txControl \dagger \{tx.idOrig \mapsto$ mk-$controlInformation\,(\{\mapsto\},$ mk-$storedSets\,(\{\mapsto\}, \{\mapsto\}, \{\mapsto\}), tx.siteOrig)\}$
.3 pre $tx.idOrig \notin \mathsf{dom}\ txControl$;

51.0 $disregControl : transOrig \times transControl \to transControl$
.1 $disregControl\,(tx, txControl) \triangleq$
.2 $\{tx.idOrig\} \lhd txControl$;

52.0 $regContact : relation \times site \times relation \xrightarrow{m} site \to relation \xrightarrow{m} site$
.1 $regContact\,(rel, s, access) \triangleq$
.2 $access \dagger \{rel \mapsto s\}$;

53.0 $disregContact : transOrig \times transId \xrightarrow{m} transId \to transId \xrightarrow{m} transId$
.1 $disregContact\,(tx, mapLocal) \triangleq$
.2 $\{tx.idOrig\} \lhd mapLocal$

operations

54.0 $processOperation : transOrig \times nodeOperation \xrightarrow{o} \mathbb{B}$
.1 $processOperation\,(tx, opr) \triangleq$
.2 (cases $opr.classOperation$:
.3 BEGIN $\to$ return $(initProtocol\,(tx))$,
.4 COMMIT, ABORT $\to$ return $(terminateProtocol\,(tx, opr.classOperation))$,
.5 others $\to$ return $(executeCommand\,(tx, opr))$
.6 end)
.7 pre $tx.siteOrig \in \bigcup \mathsf{rng}\ DDb.repDb$;

```
55.0  initProtocol : transOrig ⟶ᵒ 𝔹
 .1   initProtocol (tx) ≜
 .2     (dcl r : 𝔹 := false;
 .3      r := initProtocolDb (tx, DDb.transMapDb);
 .4      if (r = true)
 .5      then DDb.transDb := regControl (tx, DDb.transDb);
 .6      return (r) )
 .7   pre  tx.idOrig ∉ dom DDb.transDb ∧ tx.idOrig ∉ dom DDb.transMapDb
 .8   post tx.idOrig ∈ dom DDb.transDb ∧ tx.idOrig ∈ dom DDb.transMapDb ;

56.0  initProtocolDb (tx : transOrig, mapLocal : (transId ⟶ᵐ transId)) r : 𝔹
 .1   pre  tx.idOrig ∉ dom mapLocal
 .2   post tx.idOrig ∈ dom mapLocal ;

57.0  executeCommand : transOrig × nodeOperation ⟶ᵒ 𝔹
 .1   executeCommand (tx, opr) ≜
 .2     (dcl c : site,
 .3          r : 𝔹 := false,
 .4          s : site-set := {};
 .5      s := possibleSites (tx, opr.relationId);
 .6      c := chooseSite (tx, opr, DDb.repDb, s);
 .7      if (tx.siteExecution = c)
 .8      then (dcl rmaps : relation ⟶ᵐ site := DDb.transDb (tx.idOrig).access;
 .9            r := executeCommandDb (tx, opr, DDb.transMapDb);
 .10           if (r = true)
 .11           then (rmaps := regContact (opr.relationId, c, rmaps);
 .12                 DDb.transDb(tx.idOrig).access := rmaps))
 .13     else (dcl settx : transOrig := regSite (tx, c),
 .14               rmaps : relation ⟶ᵐ site := DDb.transDb (settx.idOrig).access;
 .15           if ({c} \ rng DDb.transDb (settx.idOrig).access ≠ {})
 .16           then (r := processOperationRemoteSite (settx, mk-nodeOperation (BEGIN, []), c);
 .17                 if (r = true)
 .18                 then (rmaps := regContact (opr.relationId, c, rmaps);
 .19                       DDb.transDb(settx.idOrig).access := rmaps;
 .20                       r := processOperationRemoteSite (settx, opr, c)))
 .21           else (r := processOperationRemoteSite (settx, opr, c);
 .22                 if (r = true)
 .23                 then (rmaps := regContact (opr.relationId, c, rmaps);
 .24                       DDb.transDb(settx.idOrig).access := rmaps)));
 .25     return (r) )
 .26  pre  opr.relationId ∈ dom DDb.repDb ∧ tx.idOrig ∈ dom DDb.transDb
 .27  post opr.relationId ∈ dom DDb.transDb (tx.idOrig).access ;
```

```
58.0  possibleSites : transOrig × relation →o site-set
 .1   possibleSites (tx, rel) ≜
 .2     (if (rel ∈ dom DDb.transDb (tx.idOrig).access)
 .3      then return ({DDb.transDb (tx.idOrig).access (rel)})
 .4      else if (tx.siteExecution ∈ DDb.repDb (rel))
 .5           then return ({tx.siteExecution})
 .6           else return ({r | r ∈ DDb.repDb (rel)}) )
 .7   pre let access = rng DDb.transDb (tx.idOrig).access in
 .8       (tx.idOrig ∈ dom DDb.transDb) ∧ (rel ∈ dom DDb.repDb) ∧ (access ⊆
⋃rng DDb.repDb) ;

59.0  chooseSite (tx : transOrig, opr : nodeOperation, rep : replicas, s : site-set) r : site
 .1   pre opr.relationId ∈ dom rep ∧ s ≠ {} ∧ s ⊆ ⋃rng rep ∧ tx.siteOrig ∈ ⋃rng rep
 .2   post r ∈ s ;

60.0  processOperationRemoteSite (tx : transOrig, opr : nodeOperation, s : site) r : 𝔹
 .1   post true ;

61.0  executeCommandDb (tx : transOrig, opr : nodeOperation, mapLocal : (transId →m
transId)) r : 𝔹
 .1   pre tx.idOrig ∈ dom mapLocal
 .2   post true ;

62.0  terminateProtocol : transOrig × enumOperations →o 𝔹
 .1   terminateProtocol (tx, opr) ≜
 .2     (dcl r : 𝔹 := false;
 .3      cases opr:
 .4        ABORT → return (executeRollBack (tx, rng DDb.transDb (tx.idOrig).access)),
 .5        COMMIT →
 .6             (r := executeStabilization (tx);
 .7              if (r = true)
 .8              then r := processCertificationRemoteSite (tx)
 .9              else executeRollBack(tx, rng DDb.transDb (tx.idOrig).access) )
 .10     end ;
 .11     return (r) )
 .12  pre tx.idOrig ∈ dom DDb.transDb
 .13  post tx.idOrig ∉ dom DDb.transDb ;

63.0  executeRollBack : transOrig × site-set →o 𝔹
 .1   executeRollBack (tx, setSites) ≜
 .2     (if (setSites ≠ {})
 .3      then (let s ∈ setSites in
 .4            if (s ≠ DDb.localDb)
 .5            then processOperationRemoteSite(tx, mk-nodeOperation (ABORT, []), s)
 .6                 executeRollBack(tx, setSites \ {s}) )
```

```
.7      else (DDb.transDb := disregControl (tx, DDb.transDb);
.8            executeRollBackDb(tx, DDb.transMapDb) );
.9      return (true) )
.10 pre  tx.idOrig ∈ dom DDb.transDb ∧ tx.idOrig ∈ dom DDb.transMapDb
.11 post tx.idOrig ∉ dom DDb.transDb ∧ tx.idOrig ∉ dom DDb.transMapDb ;

64.0 executeRollBackDb (tx : transOrig, mapLocal : transId -m-> transId) r : 𝔹
.1  post tx.idOrig ∉ dom mapLocal ;

65.0 executeStabilization : transOrig -o-> 𝔹
.1  executeStabilization (tx) △
.2    (dcl stabilizeSites : site-set := rng DDb.transDb (tx.idOrig).access,
.3         r : 𝔹 := false;
.4     if (DDb.certMode = NoVote)
.5     then r := computeStabilizationNoVote (tx, stabilizeSites)
.6     else r := computeStabilizationVote (tx, stabilizeSites);
.7     return (r) )
.8  pre  tx.idOrig ∈ dom DDb.transDb
.9  post let setrs = dom DDb.transDb (tx.idOrig).dataSet.rs,
.10           setws = dom DDb.transDb (tx.idOrig).dataSet.ws in
.11       ∀ rrs ∈ setrs, rws ∈ setws ·
.12            rrs ∈ setws ∧ rws ∈ setrs ;

66.0 computeStabilizationNoVote : transOrig × site-set -o-> 𝔹
.1  computeStabilizationNoVote (tx, sites) △
.2    (if (sites = {})
.3     then return (true)
.4     else let site ∈ sites in
.5          let collectedSet = processStabilizationRemoteSite (tx, site) in
.6          if (collectedSet ≠ nil )
.7          then (rebuildTransaction(tx, collectedSet) ;
.8                computeStabilizationNoVote(tx, sites \ {site}) )
.9          else return (false) );

67.0 computeStabilizationVote : transOrig × site-set -o-> 𝔹
.1  computeStabilizationVote (tx, sites) △
.2    if (sites = {})
.3    then return (true)
.4    else let site ∈ sites in
.5         let collectedSet = processStabilizationRemoteSite (tx, site) in
.6         if (collectedSet ≠ nil )
.7         then computeStabilizationVote(tx, sites \ {site})
.8         else return (false) ;
```

68.0 $rebuildTransaction : transOrig \times collectedSets \xrightarrow{o} \mathbb{B}$

.1 $rebuildTransaction\ (tx, collectedSet) \triangleq$

.2 (dcl $r : \mathbb{B} := \text{true}$,

.3 $setrs : relation \xrightarrow{m} pk\text{-set} := DDb.transDb\ (tx.idOrig).dataSet.rs$,

.4 $setws : relation \xrightarrow{m} pk\text{-set} := DDb.transDb\ (tx.idOrig).dataSet.ws$;

.5 $DDb.transDb(tx.idOrig).dataSet.rs := setrs \cup_m collectedSet.rs$;

.6 $DDb.transDb(tx.idOrig).dataSet.ws := setws \cup_m collectedSet.ws$;

.7 return (r))

.8 pre let $setrs = \text{dom}\ DDb.transDb\ (tx.idOrig).dataSet.rs$,

.9 $setws = \text{dom}\ DDb.transDb\ (tx.idOrig).dataSet.ws$ in

.10 $(tx.idOrig \in \text{dom}\ DDb.transDb) \wedge$

.11 $(\neg\ (\text{dom}\ collectedSet.rs \subseteq setrs)) \wedge$

.12 $(\neg\ (\text{dom}\ collectedSet.ws \subseteq setws))$;

69.0 $processStabilizationRemoteSite\ (tx : transOrig, cst : site)\ r : [collectedSets]$

.1 post true ;

70.0 $processCertificationRemoteSite\ (tx : transOrig)\ r : \mathbb{B}$

.1 post true

end *DistributedExecutionInPDBSM*

Appendix B

```
CREATE LOGICAL TABLE "logicaldemo"
(
 col01 int,
 col02 int,
 col03 char[30]
)
-- Creates the logical relation

CREATE TABLE "hots01.realdemo"
(
 col01 int,
 col02 int,
 col03 char[30]
)
CREATE TABLE "hots02.realdemo"
(
 col01 int,
 col02 int,
 col03 char[30]
)
-- Creates the fragments in each site.

CREATE LOGICAL RULE "rule-demo-select" AS
 ON select
 TO "realdemo"
 DO INSTEAD
select col01,col02,col03 from "host01.realdemo" where col01 = 10
                  union
select col01,col02,col03 from "host01.realdemo" where col01 = 20
-- There are 2 hosts and the relation "logicaldemo"
-- is fragmented between these hosts.
-- The where clause in these expressions define the fragments.

CREATE LOGICAL RULE "rule-demo-insert-01" AS
 ON insert
 TO "realdemo" where col01 = 10
 DO INSTEAD
  insert into "host01.realdemo"(col01,col02,col03)
  values(new.col1,new.col02,new.col03)
-- The information "new." identifies the values sent to be inserted.

CREATE LOGICAL RULE "rule-demo-insert-02" AS
```

```
 ON insert
 TO "realdemo" where col01 = 20
 DO INSTEAD
  insert into "host02.realdemo"(col01,col02,col03)
  values(new.col1,new.col02,new.col03)
-- The information "new." identifies the values sent to be inserted.

CREATE LOGICAL RULE "rule-demo-insert-03" AS
 ON insert
 TO "realdemo"
 DO INSTEAD NOTHING
-- The modifier "nothing" guarantees completeness.
-- It avoids insertion of tuples that does not have
-- the column col01 = 10 or col01 = 20.

CREATE LOGICAL RULE "rule-demo-delete-01" AS
 ON delete
 TO "realdemo" where col1 = 10
 DO INSTEAD
  delete "host01.realdemo" where col1 = 10
-- It is important to remember that final statement will combine
-- this clause with the one from the original statement.

CREATE LOGICAL RULE "rule-demo-delete-02" AS
 ON delete
 TO "realdemo" where col01 = 20
 DO INSTEAD
  delete "host02.realdemo" where col01 = 20
-- It is important to remember that final statement will combine
-- this clause with the one from the original statement.

CREATE LOGICAL RULE "rule-demo-update-01" AS
 ON update
 TO "realdemo" where col01 = 10
 DO INSTEAD
  update "host01.realdemo" set col01 = new.col01, col02 = new.col02
  col03 = new.col03 where col01 = 10
-- It is important to remember that final statement will combine
-- this clause with the one from the original statement.

CREATE LOGICAL RULE "rule-demo-update-02" AS
 ON update
 TO "realdemo" where col01 = 20
 DO INSTEAD
  update "host01.realdemo" set col01 = new.col01, col02 = new.col02
  col03 = new.col03 where col01 = 20
-- It is important to remember that final statement will combine
-- this clause with the one from the original statement.

-- To avoid this laborious work we propose to use this.
CREATE ESCADA TABLE demo
(
 (
  col01 int,
  col02 int,
  col03 char[30]
```

```
  )
)
FRAGMENTS ("col01 = 10","host01"),("col01 = 20","host02")
-- Disjointness can be easily implemented, testing pairs
-- of clauses brought together with a conjunction and
-- guaranteeing that all the tests evaluates to false.
```

Bibliography

[1] B. A. Davey ad H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University, 2002.

[2] Michel E. Adiba and Bruce G. Lindsay. Database Snapshots. In *Very Large Database Conference*, 1980.

[3] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Materialized View and Index Selection Tool for Microsoft SQL Server 2000. In *ACM SIGMOD International Conference on Management of Data*, 2001.

[4] G. Alvarez and F. Cristian. Applying Simulation to the Design and Performance Evaluation of Fault-tolerant Systems. In *IEEE International Symposium on Reliable Distributed Systems*, 1997.

[5] Shivnath Babu and Jennifer Widom. Continuous Queries Over Data Streams. *ACM SIGMOD International Conference on Management of Data*, 2001.

[6] Catriel Beeri, Philip A. Bernstein, and Nathan Goodman. A Model for Concurrency in Nested Transactions Systems. *Journal of the ACM*, 1989.

[7] Randall G. Bello, Karl Dias, Alan Downing, James J. Feenan Jr., William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized Views in Oracle. In *Very Large Database Conference*, 1998.

[8] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *ACM SIGMOD International Conference on Management of Data*, 1995.

[9] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[10] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and Jr. James B. Rothnie. Query Processing in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, 1981.

[11] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[12] J. A. Blakeley, Neil Coburn, and P. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Transactions on Database Systems (TODS)*, 1989.

[13] Phillip Bogle and Barbara Liskov. Reducing Cross Domain Call Overhead using Batched Futures. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, 1994.

[14] Michael J. Carey and Hongjun Lu. Load Balancing in a Locally Distributed DB System. In *ACM SIGMOD International Conference on Management of Data*, 1986.

[15] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. Griffiths Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost. Support for Repetitive Transactions and Ad Hoc Queries in System R. *ACM Transactions on Database Systems (TODS)*, 1981.

[16] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 1996.

[17] Chung-Min Chen and Nick Roussopoulos. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *Extending Database Technology*, 1994.

[18] Rada Chirkova, Alon Y. Halevy, and Dan Suciu. A Formal Perspective on the View Selection Problem. *Very Large Database Journal*, 2002.

[19] Choera. http://www.choera.com, 2003.

[20] Richard L. Cole and Goetz Graefe. Optimization of Dynamic Query Evaluation Plans. In *ACM SIGMOD International Conference on Management of Data*, 1994.

[21] George Copeland, William Alexander, Ellen Boughter, and Tom Keller. Data Placement in Bubba. In *ACM SIGMOD International Conference on Management of Data*, 1988.

[22] Thomas H. Cormen, Charles E. Leierson, and Ronald L. Rivest. *Introduction to Algorithms*. Mc Graw Hill, 1990.

[23] IBM Corporation. International Business Machines Corporation. http://www.ibm.com, 2003.

[24] Micrsoft Corporation. SQL Server 2000 - Books Online.

[25] Oracle Corporation. Query Optimization in Oracle9i. http://otn.oracle.com/products/bi/pdf/o9i_optimization_twp.pdf, 2004.

[26] J. Cowie, H. Liu, J. Liu, D. Nicol, and Andy Ogielski. Towards Realistic Million-Node Internet Simulation. In *Proc. of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, 1999.

[27] Shaul Dar, Michael J. Franklin, Björn Thór Jónsson, Divesh Srivastava, and Michael Tan. Semantic Data Caching and Replacement. In *Very Large Database Conference*, 1996.

[28] Andre Eickler, Alfons Kemper, and Donald Kossmann. Finding Data in the Neighborhood. In *Very Large Database Journal*, 1997.

[29] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 1976.

[30] K. P. Eswaran. Placement of Records in a File and File Allocation in a Computer Network. In *Information Processing*, 1974.

[31] D. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. Economic Models for Allocating Resources in Computer Systems, 1996.

[32] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Pratical Tools and Techniques in Software Development*. Cambridge University, 1998.

[33] Michael J. Franklin, Michael J. Carey, and Miron Livny. Local Disk Caching for Client-Server Database Systems. In *Very Large Database Conference*, 1993.

[34] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance Tradeoffs for Client-Server Query Processing. In *ACM SIGMOD International Conference on Management of Data*, 1996.

[35] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query Optimization for Parallel Execution. In *ACM SIGMOD International Conference on Management of Data*, 1992.

[36] The GENITOR Research Group in Genetic Algorithms and Evolutionary Computation. http://www.cs.colostate.edu/˜whitley/Pubs.html, 2003.

[37] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. Fast Incremental Maintenance of Approximate Histograms. In *Very Large Database Conference*, 1997.

[38] Garth A. Gibson and Rodney Van Meter. Network Attached Storage Architecture. *Communications of the ACM*, 2000.

[39] Jonathan Goldstein and Per-Åke Larson. Optimizing Queries using Materialized Views: A Practical, Scalable Solution. In *ACM SIGMOD International Conference on Management of Data*, 2001.

[40] G. Graefe and K. Ward. Dynamic Query Evaluation Plans. In *ACM SIGMOD International Conference on Management of Data*, 1989.

[41] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 1993.

[42] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *ACM SIGMOD International Conference on Management of Data*, 1996.

[43] PostgreSQL Global Development Group. PostgreSQL. http://www.postgresql.org, 2003.

[44] PostgreSQL Global Development Group. PostgreSQL Developer. http://developer.postgresql.org/docs/pgsql/src/backend, 2003.

[45] Rachid Guerraoui. Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG95)*, 1995.

[46] K. Guo. *Scalable Message Stability Detection Protocols*. PhD thesis, Cornell University, Computer Science Department, 1998.

[47] Sha Guo, Wei Sun, and Mark A. Weiss. Solving Satisfiability and Implication Problems in Database Systems. *ACM Transactions on Database Systems (TODS)*, 1996.

[48] Ashish Gupta and Inderpal Singh Mumick, editors. *Materialized Views Techniques, Implementations, and Applications*. MIT Press, 1999.

[49] Vassos Hadzilacos and Sam Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical report, Cornell University, 1994.

[50] Alon Y. Halevy. Answering Queries using Views: A Survey. *Very Large Database Journal*, 2001.

[51] Alan R. Hevner, O. Q. Wu, and S. B. Yao. Query Optimization on Local Area Networks. *ACM Transactions on Information Systems*, 1985.

[52] Toshihide Ibaraki and Tiko Kameda. On the Optimal Nesting Order for Computing N-Relational Joins. *ACM Transactions on Database Systems (TODS)*, 1984.

[53] Yannis E. Ioannidis and Eugene Wong. Query Optimization by Simulated Annealing. In *ACM SIGMOD International Conference on Management of Data*, 1987.

[54] IOzone Filesystem Benchmark. http://www.iozone.org, 2003.

[55] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. An Adaptive Query Execution System for Data integration. In *ACM SIGMOD International Conference on Management of Data*, 1999.

[56] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 1984.

[57] A. Correia Jr, A. Sousa, J. Pereira, R. Oliveira, and F. Moura. Evaluating Certification Protocols in the Partial Database State Machine. Technical report, Departamento de Informática, Universidade do Minho, 2003.

[58] M. Kaashoek and A. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *IEEE International Conference on Distributed Computing Systems*, 1991.

[59] Arthur M. Keller and Julie Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *Very Large Database Journal*, 1996.

[60] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *Very Large Database Conference*, 2000.

[61] B. Kemme, A. Bartoli, and O. Babaoglu. Online Reconfiguration in Replicated Databases Based on Group Communication. In *IEEE International Conference on Dependable Systems and Networks*, 2001.

[62] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *IEEE International Conference on Distributed Computing Systems*, 1999.

[63] Bettina Kemme and Gustavo Alonso. A Suite of Database Replication Protocols Based on Group Communication Primitives. In *IEEE International Conference on Distributed Computing Systems*, 1998.

[64] Larry Kerschberg, Peter D. Ting, and S. Bing Yao. Query Optimization in Star Computer Networks. *ACM Transactions on Database Systems (TODS)*, 1982.

[65] Anthony Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of the ACM*, 1982.

[66] Donald Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 2000.

[67] Natalija Krivokapić;, Alfons Kemper, and Ehud Gudes. Deadlock Detection in Distributed Database Dystems: A New Algorithm and A Comparative Performance Analysis. *Very Large Database Journal*, 1999.

[68] Rosana S. G. Lanzelotte, Patrick Valduriez, and Mohamed Zaït. On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces. In *Very Large Database Conference*, 1993.

[69] High-resolution timers for the Linux kernel. http://www.cs.wisc.edu/˜paradyn/libhrtime, 2003.

[70] Hongjun Lu and Michael J. Carey. Some Experimental Results on Distributed Join Algorithms in a Local Network. In *Very Large Database Conference*, 1985.

[71] Patrick Valduriez M. Tamer Özsu. *Principles of Distributed Database Systems*. Prentice Hall International, 1999.

[72] Lothar F. Mackert and Guy M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Very Large Database Conference*, 1986.

[73] Rui Oliveira. *Solving Consensus: From Fair-Lossy Channels to Crash-Recovery of Process*. PhD thesis, Département d'Informatique, l'École Polytechnique Fédérale de Lausanne, 2000.

[74] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, Département d'Informatique, l'École Polytechnique Fédérale de Lausanne, 1999.

[75] J. Pereira and R. Oliveira. A Mutable Protocol for Consenus in Large Groups. Technical report, Departamento de Informática, Universidade do Minho, 2003.

[76] S. Pingali, D. Towsley, and J. Kurose. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994.

[77] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule based Query Rewrite Optimization in Starburst. In *ACM SIGMOD International Conference on Management of Data*, 1992.

[78] Microsoft Press. *Microsoft SQL Server 7.0 Resource Guide*. Microsoft Corporation, 1999.

[79] Microsoft Press. *Microsoft SQL Server 2000 Resource Kit*. Microsoft Corporation, 2001.

[80] Margaret H. Eich Priti Mishra. Join Processing in Relational Databases. *ACM Computing Surveys*, 1992.

[81] Nicholas Roussopoulos. An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis. *ACM Transactions on Database Systems (TODS)*, 1991.

[82] A. Schiper. Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 1997.

[83] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *IEEE International Conference on Distributed Computing Systems*, 1993.

[84] Heidi Scott, Patrick Martin, and Berni Schiefer. A Study of the Impact of Direct Access I/O on Relational Database Management Systems. In *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, 2002.

[85] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *ACM SIGMOD International Conference on Management of Data*, 1979.

[86] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 1990.

[87] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial Replication in the Database State Machine. In *IEEE International Symposium on Network Computing and Applications*, 2001.

[88] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic Total Order in Wide Area Networks. In *IEEE International Symposium on Reliable Distributed Systems*, 2002.

[89] A. Sousa, J. Pereira, L. Soares, A. Correia Jr, L. Rocha, R. Oliveira, and F. Moura. Evaluating the Performance of the Database State Machine (DBSM). Technical report, Departamento de Informática, Universidade do Minho, 2003.

[90] Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. Using Broadcast Primitives in Replicated Databases. In *IEEE International Conference on Distributed Computing Systems*, 1998.

[91] Konrad Stocker, Donald Kossmann, Reinhard Braumandl, and Alfons Kemper. Integrating Semi-Join-Reducers into State of the Art Query Processors. In *IEEE International Conference on Data Engineering*, 2001.

[92] M. Stonebraker, E.N. Hanson, and S. Potamianos. The POSTGRES Rule Manager. *IEEE Transactions on Software Engineering*, 1988.

[93] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A Wide-Area Distributed Database System. *Very Large Database Journal*, 1996.

[94] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The Design and Implementation of INGRES. *ACM Transactions on Database Systems (TODS)*, 1976.

[95] StrongRep. http://gsd.di.uminho.pt/StrongRep/index.htm, 2004.

[96] Inc Sun Microsystems. The Source for Java Technology. http://www.java.sun.com, 2003.

[97] A. Swami. Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques. In *ACM SIGMOD International Conference on Management of Data*, 1989.

[98] Anne Strachan Thomas Connolly, Corolyn Begg. *Database Systems: A Pratical Approach to Design, Implementation and Management*. Addison-Wesley, 1998.

[99] TPC-W Code - University of Wisconsin. http://tpcw.deadpixel.de/, 2003.

[100] Transaction Processing Performance Council (TPC). TPC benchmark C Standard Specification Revision 5.0, 2001.

[101] Transaction Processing Performance Council (TPC). TPC benchmark W (Web Commerce) Specification Version 1.7, 2001.

[102] Jeffrey Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988.

[103] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost-Based Query Scrambling for Initial Delays. In *ACM SIGMOD International Conference on Management of Data*, 1998.

[104] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database Replication Techniques: A Three Parameter Classification. In *IEEE International Symposium on Reliable Distributed Systems*, 2000.