



universität
wien

DISSERTATION

Parallel Database Operations in Heterogeneous Environments

Verfasser

Ing. Mag. Werner Mach

angestrebter akademischer Grad

Doktor der Technischen Wissenschaften (Dr. techn.)

Wien, 2009

Studienkennzahl lt. Studienblatt:
Dissertationsgebiet lt. Studienblatt:
Betreuer:

A 786 881
Informatik
Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta

To my wife Sylvia and our children Bernhard, Ursula and Andreas

Inhaltsverzeichnis

1	Introduction	9
1.1	Motivation	9
1.2	Overview	10
1.3	Organization of this Thesis	10
2	Basics of Relational Database Systems	13
2.1	Relational Model	13
2.2	Query Languages	16
2.3	Relational Operations	17
3	Database-System Architectures	25
3.1	Centralized Architectures	25
3.2	Distributed Systems	26
3.3	Differences between Distributed and Centralized Database Systems	27
3.4	Distributed Database Management Systems	28
3.5	Reference Architecture of Data Dictionary and Fragmentation	29
3.6	Parallelism in Database Systems	30
3.7	Parallelism Goals and Metrics: Speedup and Scaleup	33
3.8	Parallel Database Architectures	36
3.9	Heterogeneous Database Architectures	36
3.9.1	An Infrastructure for Scientific Grid Computing	38
3.9.2	A Static Heterogeneous Model	39
3.10	Peer to Peer Database Architecture	40
4	Parallel Database Operations	45
4.1	Parallel Database Operations in Multiprocessor Architectures	45
4.1.1	Cost Model	45
4.1.2	The Influence of Network Cost	47
4.1.3	Parallel Sorting Algorithms	48
4.1.4	Parallel Join Operations	56
4.1.5	Parallel Aggregate Operations	59
4.2	Query Optimization in Parallel and Distributed Database Systems	63
4.2.1	Query Optimization	63
4.2.2	Interquery Parallelism	65
4.2.3	Intraquery Parallelism	65
5	A Heterogenous Architecture for Parallel Database Operations	67
5.1	Modified Parallel Database Operations	67
5.1.1	Modified Sort Operations	67
5.1.2	Modified Join Operations	68
5.1.3	Modified Aggregate Operations	69
5.2	Performance Analysis and Comparison of the Modified Operations	69

5.2.1	Sort Operations	69
5.2.2	Join Operations	72
5.2.3	Aggregate Operations	75
6	Optimization of Workflow Orchestration	81
6.1	Workflow Orchestration	81
6.1.1	Algorithm for the Workflow Orchestration	81
6.1.2	A Graph Representation of the Static Simplified Grid Organization	81
6.1.3	A Binary Tree Search Algorithm	82
6.2	Implementation of the Perfect Binary Tree Search Algorithm	83
6.2.1	Generating Random Graphs	83
6.2.2	Analysis of the Perfect Binary Tree Search Algorithm	84
6.3	The Optimized Workflow Execution Process	86
7	Implementation and Evaluation of the Static Heterogeneous Model	91
7.1	Introduction to SODA	91
7.1.1	Components of Soda	91
7.1.2	Communication in SODA	93
7.1.3	Query Execution in SODA	94
7.1.4	Performance and Performance Analysis in SODA	95
7.2	Prototype	96
7.3	Performance Evaluation	97
8	Extensions of the Static Heterogeneous Model	101
8.1	Performance Indices	101
8.1.1	One Dimensional Result Index	103
8.1.2	Two Dimensional Result Index	104
8.1.3	Using Indices in a Static Heterogeneous Model	104
8.2	Reliability Extension	105
8.2.1	Node Reliability	107
8.2.2	Database Reliability	108
8.2.3	Grid Reliability	109
8.3	Performance Metrics for Grid Workflows	109
8.4	Dynamic Optimization of Workflows	111
9	Conclusion	113
	Bibliography	123

Abstract

In contrast to the traditional notion of a supercomputer, which has many processors connected by a local high-speed computer bus, heterogeneous computing environments rely on "complete" computer nodes (CPU, storage, network interface, etc.) connected to a private or public network by a conventional network interface. Computer networking has evolved over the past three decades, and, like many technologies, has grown exponentially in terms of performance, functionality and reliability. At the beginning of the twenty-first century, high-speed, highly reliable Internet connectivity has become as commonplace as electricity, and computing resources have become as standard in terms of availability and universal use as electrical power.

To use heterogeneous Grids for various applications requiring high-processing power, researchers propose the notion of computational Grids where rules are defined relating to both services and hiding the complexity of the Grid organization from the users. Thus, users would find it as easy to use as electrical power [1].

Generally, there is no widely accepted definition of Grids. Some researchers define it as a high-performance distributed environment. Some take into consideration its geographically distributed, multi-domain feature [2]. Others define Grids based on the number of resources they unify [3].

Parallel database systems [4] gained an important role in database research over the past two decades due to the necessity of handling large distributed datasets for scientific computing such as bioinformatics, fluid dynamics and high energy physics (HEP) [5]. This was connected with the shift from the (actually failed) development of highly specialized database machines to the usage of conventional parallel hardware architectures. Generally, concurrent execution is employed either by database operator or data parallelism [6]. The first is achieved through parallel execution of a partitioned query execution plan by different operators, while the latter is achieved through parallel execution of the same operation on the partitioned data among multiple processors.

Parallel database operation algorithms have been well analyzed for sequential processors. A number of publications have covered this topic, such as [7, 8, 9], which proposed and analyzed these algorithms for parallel database machines. Until now, to the best knowledge of the author, no specific analysis has been done so far on parallel algorithms with a focus on the specific characteristics of a Grid infrastructure.

The specific difference lies in the heterogeneous nature of Grid resources. In a "shared nothing architecture", which can be found in classical supercomputers and cluster systems, all resources such as processing nodes, disks and network interconnection have typically homogeneous characteristics as regards to performance, access time and bandwidth. In contrast, in a Grid architecture heterogeneous resources are found that show different performance characteristics. The challenge of this research is to discover the way how to cope with or to exploit this situation to maximize performance and to define algorithms that lead to a solution for an optimized workflow orchestration.

To address this challenge, we developed a mathematical model to investigate the performance behavior of parallel database operations in heterogeneous environments, such as a Grid, based on generalized multiprocessor architecture. We also studied the parameters and their influence on the performance as well as the behavior of the algorithms in he-

terogeneous environments. We discovered that only a small adjustment on the algorithm is necessary to significantly improve the performance for heterogeneous environments. A graphical representation of the node configuration and an optimized algorithm for finding the optimal node configuration for the execution of the parallel binary merge sort have been developed.

Finally, we have proved our findings of the new algorithm by implementing it on a service-orientated infrastructure (SODA). The model and our new developed modified algorithms have been verified with the implementation.

We also give an outlook of useful extensions to our model e.g. using performance indices, reliability of the nodes and approaches for dynamic optimization of workflows.

Zusammenfassung

Im Gegensatz zu dem traditionellen Begriff eines Supercomputers, der aus vielen mittels superschneller, lokaler Netzwerkverbindungen miteinander verbundenen Superrechnern besteht, basieren heterogene Computerumgebungen auf "kompletten" Computersystemen, die mit Hilfe eines herkömmlichen Netzwerkanschlusses an private oder öffentliche Netzwerke angeschlossen sind. Der Bereich des Computernetzwerks hat sich über die letzten drei Jahrzehnte entwickelt und ist, wie viele andere Technologien, in Bezug auf Performance, Funktionalität und Verlässlichkeit extrem gewachsen. Zu Beginn des 21. Jahrhunderts zählt das betriebssichere Hochgeschwindigkeitsnetz genauso zur Alltäglichkeit wie Elektrizität, und auch Rechnerressourcen sind, was Verfügbarkeit und universellen Gebrauch anbelangt, ebenso Standard wie elektrischer Strom [1].

Wissenschaftler haben für die Verwendung von heterogenen Grids bei verschiedenen rechenintensiven Applikationen eine Architektur von computational Grids konzipiert und darin Modelle aufgesetzt, die zum einen Rechenleistungen definieren und zum anderen die komplexen Eigenschaften der Grid-Organisation vor den Benutzern verborgen halten. Somit wird die Verwendung für den Benutzer genauso einfach wie es möglich ist elektrischen Strom zu beziehen. Grundsätzlich existiert keine generell akzeptierte Definition für Grids. Einige Wissenschaftler bezeichnen sie als hochleistungsfähige verteilte Umgebung. Manche berücksichtigen bei der Definierung auch die geographische Verteilung und ihre Multi-Domain-Eigenschaft [2]. Andere Wissenschaftler wiederum definieren Grids über die Anzahl der Ressourcen, die sie verbinden [3].

Parallele Datenbanksysteme [4] haben in den letzten zwei Jahrzehnten große Bedeutung erlangt, da das rechenintensive wissenschaftliche Arbeiten, wie z.B. auf dem Gebiet der Bioinformatik, Strömungslehre und Hochenergiephysik die Verarbeitung riesiger verteilter Datensätze erfordert [5]. Diese Tendenz resultierte daraus, dass man von der fehlgeschlagenen Entwicklung hochspezialisierter Datenbankmaschinen zur Verwendung herkömmlicher paralleler Hardware-Architekturen übergegangen ist. Grundsätzlich wird die gleichzeitige Abarbeitung entweder durch verteilte Datenbankoperationen oder durch Datenparallelität gelöst [6]. Im ersten Fall wird ein unterteilter Abfragenabarbeitungsplan durch verschiedene Datenbankoperatoren parallel durchgeführt. Im Fall der Datenparallelität erfolgt eine Unterteilung der Daten, wobei mehrere Prozessoren die gleichen Operationen parallel an Teilen der Daten durchführen.

Es liegen genaue Analysen von parallelen Datenbank-Arbeitsvorgängen für sequenzielle Prozessoren vor. Eine Reihe von Publikationen, wie z.B. [7, 8, 9] haben dieses Thema abgehandelt und dabei Vorschläge und Analysen für parallele Datenbankmaschinen erstellt. Bis dato existiert allerdings noch keine spezifische Analyse paralleler Algorithmen mit dem Fokus der speziellen Eigenschaften einer "Grid"-Infrastruktur.

Der spezifische Unterschied liegt in der Heterogenität von Grid-Ressourcen. In "shared nothing"-Architekturen, wie man sie bei klassischen Supercomputern und Cluster-Systemen vorfindet, sind alle Ressourcen wie z.B. Verarbeitungsknoten, Festplatten und Netzwerkverbindungen angesichts ihrer Leistung, Zugriffszeit und Bandbreite üblicherweise gleich (homogen). Im Gegensatz dazu zeigen Grid-Architekturen heterogene Ressourcen mit verschiedenen Leistungseigenschaften. Der herausfordernde Aspekt dieser Arbeit bestand darin aufzuzeigen, wie man das Problem heterogener Ressourcen löst, d.h. diese

Ressourcen einerseits zur Leistungsmaximierung und andererseits zur Definition von Algorithmen einsetzt, um die Arbeitsablauf-Orchestrierung von Datenbankprozessoren zu optimieren.

Um dieser Herausforderung gerecht werden zu können, wurde ein mathematisches Modell zur Untersuchung des Leistungsverhaltens paralleler Datenbankoperationen in heterogenen Umgebungen, wie z.B. in Grids, basierend auf generalisierten Multiprozessor-Architekturen entwickelt. Es wurden dabei sowohl die Parameter und deren Einfluss auf die Leistung als auch das Verhalten der Algorithmen in heterogenen Umgebungen beobachtet. Dabei konnte man feststellen, dass kleine Anpassungen an den Algorithmen zur signifikanten Leistungsverbesserung heterogener Umgebungen führen. Weiters wurde eine graphische Darstellung der Knotenkonfiguration entwickelt und ein optimierter Algorithmus, mit dem ein optimaler Knoten zur Ausführung von Datenbankoperationen gefunden werden kann.

Diese Ergebnisse zum neuen Algorithmus wurden durch die Implementierung in einer serviceorientierten Architektur (SODA) [10] bestätigt. Durch diese Implementierung konnte die Gültigkeit des Modells und des neu entwickelten optimierten Algorithmus nachgewiesen werden.

In dieser Arbeit werden auch die Möglichkeiten für eine brauchbare Erweiterung des vorgestellten Modells gezeigt, wie z.B. für den Einsatz von Leistungskennziffern für Algorithmen zur Findung optimaler Knoten, die Verlässlichkeit der Knoten oder Vorgehensweisen/Lösungsaufgaben zur dynamischen Optimierung von Arbeitsabläufen.

Acknowledgements

First and foremost, I would like to thank my advisor, Erich Schikuta, for showing me the research path, as well as for all his suggestions and assistance during my work and for his advice and expert guidance. At the beginning of my research I had some trouble to find my way through all the information in this field. Erich showed me the right way to search the proper themes. I am very proud that the research resulted in these findings. They were hidden like a treasure, waiting years to be dug out.

I would also like to thank Peter Beran, Jürgen Mangler and Helmut Wanek for their great support. It was a great pleasure to work with them and to have many inspiring discussions. My special thanks also go to Helmut for his competent mathematical knowledge and careful proofreading of our publications.

In addition, I would also like to thank Peter Beran, Ralph Vigne and Michael Koitz for their hard work of implementing our algorithm, and Elvin Sulejmanovic for helping to improve my thesis.

Last, but not least, I would like to thank my wife, Sylvia, and my children, Bernhard, Ursula and Andreas, for their abundant patience and understanding when I spent hours on my computer conducting my research.

I have done all my research work while working full time. Research is completely different to the work I do for a company, and the major barrier I was confronted with was to have enough time to successfully complete this study. Nevertheless, it was great fun.

Werner Mach
Vienna, Austria
June 2009

Remarks

I have tried very hard to find all owners of the pictures to get the allowance to use their pictures in my work. If there is any copyright violation, please contact me.

I use "we" and "our" in my thesis as a gesture of respect to the research community. This convention is part of my attitude towards that community.

In the chapter *Conclusion* I focussed on 10 main points of my thesis. Seven out of these 10 are dedicated to this work while three are statements of my life experience and a little bit of personal philosophy.

List of Publications

Following list of publications are all peer reviewed from at least 3 independent reviewers, which were the results of this thesis. The chapters 5, 6 and 7 are partly derived from these publications.

- Werner Mach and Erich Schikuta. Performance Analysis of Parallel Database Sort Operations in a Heterogenous Grid Environment. In *Sixth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar) in conjunction with the 2007 IEEE International Conference on Cluster Computing (Cluster07)*, Austin Texas, 9 2007. IEEE Computer Society Press.

In this work an analytical comparison of the performance behavior of parallel flavors of the well-known Binary Merge Sort and Bitonic Sort algorithm in a Grid environment was presented. We developed a concise but comprehensive analytical model both for a generalized multiprocessor framework and a simplified heterogeneous Grid Environment. We defined a limited number of characteristic parameters for the model. A smart enhancement of the algorithms exploiting the specifics of the Grid the well-known results of Bitton et al. for a homogenous multi-processor architecture are invalidated and reversed for a heterogeneous Grid environment.

- Werner Mach and Erich Schikuta. Analysis of Parallel Binary Merge Sort in a Grid Environment. In *ISCA 20th International Conference on Parallel and Distributed Computing Systems*, pages 187–192, Las Vegas, Nevada, 9 2007.

In this paper we present an analysis and evaluation of one of the most prominent parallel sorting algorithms, Parallel Binary Merge Sort, in a Grid architecture. It presents an analysis of the parallel binary merge sort algorithm and a comparison of its behavior in a generalized multiprocessor framework and a simplified Grid Environment. Justified by the findings of this paper and resulting from the characteristics in the Grid environment, the performance of Parallel Binary Merge Sort can be remarkably increased by smart orchestration of the workflow on the available Grid resources taking into account the specific node characteristics. These stimulating findings lead to consequences for the design of workflows on the Grid and the development of novel cost-based broker policies of the Grid middleware.

- Werner Mach and Erich Schikuta. Parallel Database Sort and Join Operations Revisited on Grids. In *High Performance Computation Conference (HPCC)*, Houston, Texas, 9 2007.

Sorting and Joining are extremely demanding operations in a database system. They are the most frequently used operators in query execution plans generated by database query optimizers. Therefore their performance influences dramatically the performance of the overall database system. In this paper we present an analysis and evaluation of the renowned parallel sort and join algorithms, Binary Merge Sort and Bitonic Sort, and Nested-Loop and Sort Merge Join in a Grid architecture. In a Grid environment the performance of these algorithms can be improved by choosing an adapted workflow layout on the Grid taking smartly into account the specific node characteristics like the network bandwidth and processing power.

- Werner Mach and Erich Schikuta. Parallel Database Join Operations in Heterogenous Grids. In *The 8th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, Adelaide, Australia, 9 2007. IEEE Computer Society Press.

We are sure there is an urgent need for novel database architectures and application domains, as high energy physics experiments, bioinformatics, drug design, etc.,

with huge data sets to administer, search and analyze. We develop a concise but comprehensive analytical model for the well-known Hash Join algorithm and compare it to Nested-Loop and Sort-Merge Join algorithms. We justify that a meaningful model can be built upon the characteristic parameter sets, describing node processing performance, the I/O and the disk bandwidth, which are the parameters for the optimization of the Grid workflow by a smart brokerage mechanism.

- Werner Mach and Erich Schikuta. Optimized Workflow Orchestration of Parallel Database Aggregate Operations on a Heterogenous Grid. In *The 37th International Conference on Parallel Processing (ICPP-08)*, Portland, Ohio, USA, 9 2008. IEEE Computer Society.

We present an analytical discussion of parallel database aggregate algorithms (count, sum, average, min, max) on Grids, compares the findings to the classical generalized multiprocessor framework of Bitton et. al. , and describes an optimization algorithm to maximize the performance for a heterogenous environment. In the past we analyzed the parallel database operators for sorting and join in-depth and proposed an optimizing workflow orchestration algorithm which allows for increased performance in a heterogenous Grid environment. In this paper we extend our approach to database aggregate operators.

- Werner Mach and Erich Schikuta. Parallel Algorithms for the Execution of Relational Database Operations Revisited On Grids. *Int. J. High Perform. Comput. Appl.*, 23(2):152–170, 2009.

It is a special pleasure to present all our results and main findings in an umbrella work in a highly notable international journal, which is on the A-list of the Faculty of Computer Science at the University of Vienna, Austria. The work has been published in the *International Journal of High Performance Computing Applications* from SAGE Publications Ltd.

1 Introduction

1.1 Motivation

Parallel database systems and their operations are well analyzed for special multiprocessor- and cluster environments. Also the area of distributed database systems are thoroughly investigated. The findings of this research are implemented in many commercial products. The underlying parallel algorithms of the most important operations (like sort, join and aggregate) are optimized and tuned to these environments. The field of parallel database operations in heterogeneous environments does not attract much attention. The relevance of the usage of optimized algorithms in heterogeneous environments has been growing in the last years. The reason for it is the demand for distributed database systems to handle large data-sets for scientific computing such as bioinformatics, fluid dynamics and high energy physics (HEP) [5]. The arising questions are:

- Which optimizations of parallel database algorithms in heterogeneous environments are possible ?
- How can these parallel algorithms be described in a mathematical model ?
- How can the algorithms be compared in terms of performance, flexibility and practical usage ?

The goal was to find answers to these questions:

- Are there present algorithms for parallel database operations that can be optimized for heterogeneous environments ?
- How can the performance and the adaptability be optimized in such environments ?
- How can these findings be used in a query-optimizer ?

We have done following steps in our research to find answers to the previous questions by:

- Analyzing the existing parallel algorithms of the most important database operations in conventional environments.
- Building an analytical model to describe the characteristics of heterogeneous environments.
- Investigating the existing algorithms in terms of performance in this model for the purpose of optimizing it.
- Studying the performance behaviors and the influencing variables of the heterogeneous environments on these algorithms.
- Optimization of the workflow orchestration.
- Implementation of the model to justify and proof the results found.

1.2 Overview

This thesis presents an analytical study of parallel algorithms for relational database operations in a heterogeneous environment and compares the findings to the classical generalized multiprocessor framework. It also describes an optimization algorithm to maximize performance for a heterogeneous environment. This optimization algorithm can be used in a Grid environment for the orchestration of the execution workflow.

We developed a concise but comprehensive analytical model of parallel algorithms for sorting, joining and aggregation. In our approach, we focussed on a limited number of characteristic parameters to keep the analytical model clear. It is shown that an expressive model can be built upon only three characteristic parameter sets: the node processing performance; the network; and the disk bandwidth. Based on these results, the thesis proves that exploiting the heterogeneity of the Grid by smart enhancement increases markedly the performance of the algorithms for database operations.

We believe that the Grid delivers a suitable environment for parallel and distributed database systems. We are sure there is an urgent need for novel database architectures and application domains with huge data sets to administer, search and analyze, such as high-energy physics experiments, bioinformatics and drug design. This situation is reflected by a specific impetus in distributed database research in the Grid. This research started with the DataGrid project [11] and, currently, the OGSA-DAI project [12].

In the past, research was conducted primarily in the area of distributed data management, but now there is a focus on research of parallel database operators targeting Grid architectures (see [13, 14]). We focused on the most important operation in relational database systems, sorting, because it is an extremely demanding operation in such a system. Sorting is one of the most frequently used operators in query execution plans generated by database query optimizers. Therefore, its performance influences dramatically the performance of the overall database system [15, 16]. Generally, sorting algorithms can be divided into main memory based (internal) and disk based (external) algorithms [17]. An external sorting algorithm is necessary if the data set is too large to fit into the main memory. Obviously, this is a common case in database systems. Therefore, in this thesis we present an analysis and evaluation of the most prominent parallel sort, that is, join and aggregate algorithms in a Grid architecture, and compare the results to the well-known analysis and findings of Bitton et al. [18].

These algorithms are investigated and reviewed under the specific characteristics of the Grid environment. And the surprising fact, justified by the findings of this research and resulting from the characteristic situation of the Grid, is that some results of Bitton et al. on the general performance of these parallel algorithms for a homogeneous multi-processor architecture are invalidated and reversed for a Grid environment. In a Grid environment, the performance of these algorithms can be improved by choosing an adapted workflow layout on the Grid, taking into account the specific node and the connecting network characteristics.

These stimulating results lead to consequences for the design of database query execution workflows on the Grid and the development of novel cost-based broker policies of the Grid middleware.

1.3 Organization of this Thesis

In chapter 1 we describe our motivation to do this work, the major questions at the beginning of this research and also a short overview.

The chapters 2 and 4 are designed to give an overview in the state of the art of relational database system infrastructures.

Chapter 2 gives a short introduction in relational database system infrastructures. The relational model from Codd, the most important relational operators and query languages are also described.

Chapter 3 describes modern database system architectures with a section of distributed database management systems

In chapter 4 the common parallel database operations and their algorithms are presented. In the last section query optimization in parallel and distributed database systems are also introduced. The section of query optimization is important to understand our approach of workflow optimization.

Chapter 5 describes heterogeneous architectures for parallel database systems. This chapter includes sections with workflow systems and database systems in heterogeneous environments as well as sections for our developed static heterogeneous model. We introduce the modification of the traditional parallel database operations to optimize their performance in heterogeneous environments. An in depth performance evaluation of the modified algorithms compared to the unmodified algorithms is presented. The last section covers the results of the modified algorithms and a new workflow orchestration algorithm has been defined.

The optimization of the workflow orchestration developed in the previous chapter is shown in chapter 6. For this purpose we developed a graphical representation of our static heterogeneous infrastructure and an algorithm, the so called **Perfect Binary Tree Search Algorithm**, for the optimal node configuration in the modified parallel database sort operation. Based on the result of this search algorithm an optimized workflow execution process is presented.

A real implementation of the static heterogeneous model with the modified and unmodified parallel database operations is presented in chapter 7. In this chapter we introduce a service oriented workflow environment where the operations have been implemented. Sections with the implemented prototype and an in depth performance evaluation are also included in this chapter.

Chapter 8 describes extensions of our static heterogeneous model. The following extensions have been discussed:

- Performance Indices.
- Reliability Extension.
- Performance Metrics for Workflows.
- Dynamic Optimization of Workflows.

In the section of performance indices we show that it is possible to optimize the node selection in the workflow orchestration, if the application (the parallel database operations in our case) and their resource usage are known. The reliability of the involved nodes is an important factor in heterogeneous environments, especially the number of nodes is very large, the node selection should be done carefully and depending on their reliability. A short introduction to performance metrics for workflow is also presented. In the section of dynamic optimization of workflows we give an overview of the necessary requirements.

The thesis ends with a chapter that contains a conclusion of this thesis, further research and the statements of the thesis.

2 Basics of Relational Database Systems

In this chapter we give an explanation of the relational model, the most important relational operators and an overview of database-system architectures. The primary function of this introduction is to understand the concepts of the relational model and their operators and how they are implemented in database systems. We have carefully investigated the implementation of relational operators in parallel architectures.

2.1 Relational Model

The relational model is the most established database system model at the moment and most of the current database systems are based on it [19] because of its solid theoretical foundation and the ability to use mathematical set operations on relations. It was proposed by Codd in 1970 [20].

The appropriate database management system *DBMS* is denoted as the relational database management system *RDBMS*.

The relational model does not only describe the data formally, but it is also directly implemented by most of the commercial database systems, which means that the data in the database system is really organized like in the model.

The dominant data manipulation and data definition language is SQL (Structured Query Language) which was also invented by Edgar F. Codd [21]. The first relational database systems were:

- 1970: *RDMS* developed at M.I.T. from L.A. Kraning and A.I. Fillat [22].
- 1974: *INGRES* (**I**Nteractive Graphics and **RE**trieval **S**ystem) developed from Michael Stonebraker and Eugene Wong at the University of California, Berkeley, California [23].

In *INGRES* the query language *QUEL* (**Q**Uery **L**anguage) has been developed. *QUEL* was the predecessor of *SQL* (**S**tructured **Q**uery **L**anguage). Stonebraker and Wong implemented the ideas of Codd's paper [24].

The relational model has an advantage in respect to the network- or hierarchial models. That is to say that nearly the whole data is mathematically describable by the help of the set theory, which makes relational database systems flexible. Hence most of the definitions in this model are mathematical. The assumption of the relational model is that all data is represented as mathematical n -ary relations, an n -ary relation being a subset of the Cartesian product of n domains.

Operations and requests on the relations are all determined by the relational algebra.

To illustrate the concepts of the relational model a number of different fictitious relations will be used. These examples may not be like they would actually be implemented in a database systems in real world.

A relational database system can be seen as a set of tables, where the whole information is represented, which means that you see the data and the relationships among those data in the tables. Since this model is mathematically describable a table is called **relation**,

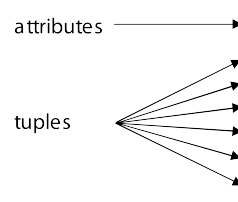
from which the relational model takes its name. Due to special characteristics which a relation must have, every relation is a table but not every table is a relation.

Each relation represents an entity in real world, whereas an entity in a database system is defined as a set of attributes. In each row of a relation there are values which all belong to one object of the entity. The set of values of one row is denoted as a **tuple**. Therefore a relation is a set of tuples.

In each column of the relation there are values, which are of the same kind but each value belongs to another object. Being of the same kind means that all values exist in the set of permitted values, called the **domain** of the attribute.

In order to be a relation a table has the following characteristics:

- the columns do not have to be ordered
- the rows do not have to be ordered
- there are no identical tuples (set property)



CustomerID	FirstName	LastName	City	Country
0001	Maria	Anders	Berlin	Germany
0002	Ana	Trujilo	London	UK
0003	Antonio	Moreno	Mannheim	Germany
0004	Hanna	Moos	London	UK
0005	Victoria	Ashworth	Marseille	France
0006	Elizabeth	Lincoln	null	null

Abbildung 2.1: The relation CUSTOMER

The formality in the relational model is very strict and as mentioned it is based on the mathematical set theory. So the concept of the relation is also formally defined:

A relation instance $r(R)$ is an instance with the relational schema R . A relational schema R , denoted by $R(A_1, A_2, \dots, A_n)$ is a set of attributes $R = \{A_1, A_2, \dots, A_n\}$. For every attribute A_i the domain $Dom(A_i)$ is the set of all permitted values for this attribute.

Each tuple t of a relation is a set of n values $t = \{v_1, v_2, \dots, v_n\}$ where all values v_i are an element of the domain $Dom(A_i)$ or they have the value **null**. So the value null is a member of any possible domain [19]. If any v_i is null, it implies that the value for the attribute A_i is whether not known or it does not exist. Null values can cause a number of problems, which were discussed for example in [25]. Therefore, null values should be eliminated if possible.

A relation can also be defined as the **subset of the Cartesian product** of the domains that define R [26].

$$r(R) \subseteq (dom(A_1) \times (dom(A_2) \times \dots (dom(A_n)))$$

The Cartesian product is the set of all possible combinations of values of the domains.

To explain the formal terms better, in Figure 2.1 there is an example of a relation. So CUSTOMER is an instance of a relation with the relational schema

CUSTOMER(*CustomerID*, *FirstName*, *LastName*, *City*, *Country*)

The domain for the attribute *FirstName* $Dom(FirstName)$ is a set of all possible first names. So $Dom(LastName)$ is a set of all possible last names and $Dom(CustomerID)$, for example, is a set of all possible ID numbers consisting of four numbers. A possible definition of $Dom(FirstName)$ is the set of all possible string values with a given maximum length and with some special characters not being allowed to use. Of course the value null is also a member of this domain.

A domain is also often defined as a set of atomic values, whereby atomic values are not further divisible in a meaningful way as far as the relational model is concerned. This definition can be explained by means of the CUSTOMER relation. If you merged the attributes *FirstName* and *LastName* into one attribute *Name*, the domain $Dom(Name)$ would be made up by all possible first and all possible last names together. These would not be atomic values, as they are further divisible in a meaningful way.

A tuple of the CUSTOMER relation is for example:

$$t = \{0001, Maria, Anders, Berlin, Germany\}$$

As already mentioned the values in a table, more specifically the columns of a relation do not need to have a specific order because of the fact that there are never two identical tuples. So the tuple

$$t = \{Maria, Germany, 0001, Berlin, Anders\}$$

is identically equal to the first one.

As each tuple is unique in a relation, there must be an attribute or a combination of attributes which is always unique in the relation. So any attribute or any combination of attributes which has the characteristic to distinctly identify a tuple is called **superkey**. They can also consist of more than the minimum number of attributes needed to identify tuples. So superkeys consist of any combination of attributes which are a subset of the relational schema and distinctly identify tuples.

OrderID	CustomerID	EmployeeID	OrderDate	ShipCity	ShipCountry
1501	0002	12	1997-11-11	Reims	France
1432	0004	3	1997-11-11	Salzburg	Austria
897	0001	5	1997-11-13	Bern	Switzerland
2221	0001	5	1997-11-13	Bruxelles	Belgium
1145	0002	2	1997-11-13	Paris	France
521	0005	9	1997-11-11	Vienna	Austria

Abbildung 2.2: The relation ORDER with the primary key: OrderID

If you have a relation r with the relation Schema R then the superkey SK of R is always a subset of R . Since two tuples are never identical in R , they must not be identical in SK . Mathematically that means, if $t_1[R] \neq t_2[R]$, then $t_1[SK] \neq t_2[SK]$.

The minimum number of attributes of a superkey is called **candidate key**. Though, the attribute or combination of attributes which really has the purpose to distinctly identify tuples is called **primary key**. The primary key of a relation is mostly the candidate key with the fewest attributes.

Candidate keys must be chosen carefully. The combination of first and last name for example is not sufficient to identify tuples distinctly. The combination of first, last name and date of birth may not be enough too, because there could still be persons with same characteristics. A candidate key could be the combination of the name and the address of a person. Certainly, this candidate key would not be a good primary key, since primary keys should consist of attributes which hardly ever change. So a good primary key would be the social security number or any other identical number of an object. If there are no sensible candidate keys in a relation, an ongoing number could be added as an attribute in the relation. Furthermore it is necessary that the attributes of a primary key are **not nullable**, which means that the value null should not be part of their domain. Otherwise the primary key would not always have the feature to make tuples unique. Of course the attributes of a candidate key (which are not part of the primary key) can be nullable.

In Figure 2.2 there is a relation ORDER. Superkeys of that relation are for example $\{OrderID, OrderDate, ShipCity\}$ or $\{EmployeeID, CustomerID, OrderID\}$. As candidate keys are always a subset of superkeys, a possible candidate key could be $\{OrderID\}$ or $\{CustomerID, OrderDate, EmployeeID\}$. These keys do not have any attributes that you do not need for identifying tuples. The best primary key, as it is the one with the fewest number of attributes, would be $\{OrderID\}$. In the relational model attributes that are part of the primary key are underlined, that is why $\{OrderID\}$ in 2.2 is underlined.

A **foreign key** is an attribute or again a combination of attributes in a relation r_1 which is a primary key in another relation r_2 . The relation r_1 where the foreign key occurs is called the **referencing relation**, while r_2 is the **referenced relation**. The domain of the foreign key in r_1 is always dependent on the values of the primary key in r_2 . That means, that in attributes of the foreign key of r_1 values can not appear which do not exist in attributes of the primary key of r_2 .

According to that the foreign keys are used to represent relationships between the relations of a database system. As depiction of foreign key and primary key dependencies the schema diagram is used.

In Figure 2.3 there is a schema diagram of a part of a database system, namely the relations from Figures 2.1 and 2.2, ORDER and CUSTOMER, and a third relation EMPLOYEE. The primary key of each of the relations is in the top of the box divided by a horizontal line from the other attributes of the relation. The dependencies between the relations are shown by the help of arrows which go from the referencing relation to the referenced relation. So ORDER has an attribute as primary key $\{OrderID\}$ and two foreign keys $\{CustomerID\}$ and $\{EmployeeID\}$. These foreign keys cause the relationship to the referenced relations.

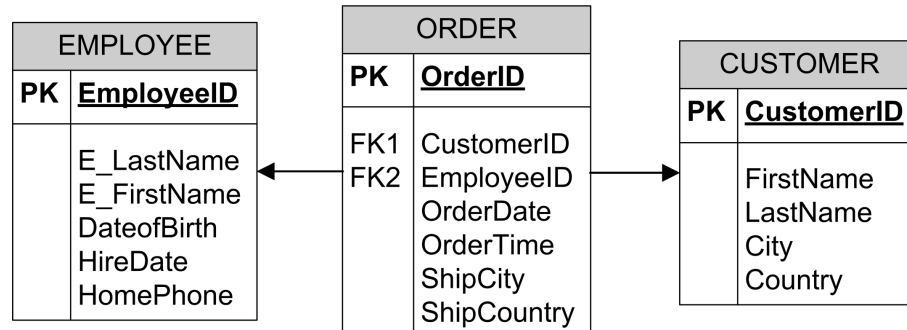


Abbildung 2.3: Schema diagram for the relations CUSTOMER, ORDER and EMPLOYEE

2.2 Query Languages

Every data model needs possibilities for retrieving as well as manipulating data. In order to fulfill these functions there are query languages for every model. For the relational model the dominant query language is SQL, derived from **Structured Query Language**. Most of the relational database management systems support the SQL Language, as it has become a standard for the relational data model. In 1986, the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) published an SQL standard, called SQL-86 in the standard ISO/IEC 9075 [19]. Since then a few new versions followed. At the moment SQL:2008 is the latest release from ISO [27].

There are two classes of query languages. First **procedural languages**, where it is specified by the user what data is needed and how to obtain it. Then the systems perform

the needed operations on the database system and returns the result. In **nonprocedural languages** a statement is given to the database management system, which is translated into a procedure describing the result wished. This procedure manipulates the data eventually.

Most relational database systems offer a query language that includes elements of both types, procedural and nonprocedural [19]. The relational algebra, which is a collection of operations, is not a real query language itself. But it is procedural and the basis for SQL. By the means of the operations, which are provided by the relational algebra, relations can be manipulated and the result is always another relation. The operations are divided into **unary** and **binary** operations. Unary operations are the ones which operate on one relation. The other ones operate on two relations.

2.3 Relational Operations

At first unary operations will be explained. These are the Select Operation and the Project Operation. Afterwards, the binary operations, working with two relations, are explained. These are in particular the Union Operation, Set-Difference Operation, Cartesian-Product Operation, Set-Intersection Operation, Natural Join Operation and the Division Operation.

Select Operation

With the **select operation** all tuples of a given relation, which satisfy a given condition can be selected. So the result of a select operation is always another relation. Furthermore the resulting relation is always a subset of the tuples of the original relation.

For instance, to filter out all customers from our CUSTOMER relation in Figure 2.1, whose home country is "Germany" you can use the select operation. The notation is the following:

$$\sigma_{Country="Germany"}(CUSTOMER)$$

The σ stands for the select operator and then the condition is following which is a boolean expression. This expression is made up of an attribute name, an operator and usually a constant value like "Germany" in our case or another attribute name. Possible operators are $\{=, \leq, \geq, <, >, \neq\}$. At the end of the select operation the name of the relation on which the operation is carried out, is specified. That is in CUSTOMER our case.

The result of our operation is a relation in which all tuples of CUSTOMER are shown that have the value true for the boolean condition $Country = "Germany"$. The attributes of the new relation remain the same as in the original relation, as you can see in Figure 2.4.

<u>CustomerID</u>	FirstName	LastName	City	Countra
0001	Maria	Anders	Berlin	Germany
0003	Antonio	Moreno	Mannheim	Germany

Abbildung 2.4: Result of $\sigma_{Country="Germany"}(CUSTOMER)$

As the condition expression can be made up of two attribute names, you have the possibility to compare two attributes of one tuple. If you wanted to get all customers whose first name is the same as their last name, you would write:

$$\sigma_{FirstName=LastName}(CUSTOMER)$$

In our CUSTOMER relation there is no customer who this condition is true for.

Furthermore several boolean expressions can be combined by using *and*(\wedge), *or*(\vee) or *not*(\neg). So if you want all customers, whose home country is Germany or whose home city is Marseille then you write:

$$\sigma_{Country="Germany" \vee City="Marseille"}(CUSTOMER)$$

The result would be a relation with three tuples. All customers for who either the first condition or the second condition is true would be selected.

That leads us to the following interpretations [26]:

- (condition1 \vee condition2) is true if either condition1 or condition2 is true. Only if both conditions are false, the tuple is not selected.
- (condition1 \wedge condition2) is only true if both conditions are true. If only one of the conditions is false, the tuple is not selected.
- (\neg condition) is true if the condition is false. If the condition is true, the tuple is not selected.

Project Operation

As already mentioned the result of the select operation is a relation with the same attributes as in the original relation. For selecting certain attributes of a relation, the **project operation** is needed.

So for filtering out only the first and last names of the CUSTOMER relation, the project operation is used with following notation:

$$\pi_{FirstName, LastName}(CUSTOMER)$$

The attributes *CustomerID*, *Country* and *City* are left out now.

Since the result is a relation, again all tuples must be distinct. But the projected attributes do not have to be key attributes, so it is possible that there would be duplicate tuples which are eliminated.

So if you project the city and the country of all customers, you write:

$$\pi_{City, Country}(CUSTOMER)$$

The result is shown in Figure 2.5. As there are two customers who are from London, one duplicate is eliminated and only five tuples are in the new relation.

City	Country
Berlin	Germany
London	UK
Mannheim	Germany
Marseille	France
null	null

Abbildung 2.5: Result of $\pi_{City, Country}(CUSTOMER)$

Of course the select and project operations can be combined by the help of the relational algebra. To get a relation with the attributes *FirstName* and *LastName* of those customers who live in the UK you write

$$\pi_{FirstName, LastName}(\sigma_{Country="UK"}(CUSTOMER))$$

So the project operation does not get directly a relation as input, but instead an expression whose result is a relation.

Union Operation

The **union operation** is already a binary operation in contrast to the project and select operation. With the union operation, values of a specific attribute which appear either in one of two different relations or in both, can be found.

If you want a relation which contains all first and last names of persons who are customer and employee, the union operation can be used. But for a union operation $r \cup s$ to be valid, two conditions are required [19].

- The relation r and s must have the same number of attributes.
- The domains $Dom(A_i)$ of r and $Dom(B_i)$ of s must be the same for all attributes.

It does not matter whether the relations are database system relations or even results of relational-algebra operations. So to get the union of CUSTOMER and EMPLOYEE from 2.3, firstly the relations must meet the conditions above. So for both relations only the names are projected and let us say the results are the relations CUSTOMER_{new} and EMPLOYEE_{new}. To get the union of these relations, you write:

$$\pi_{FirstName, LastName}(CUSTOMER_{new}) \cup \pi_{E_FirstName, E_LastName}(EMPLOYEE_{new})$$

The result is again a relation and like in all relations tuples must be distinct, so duplicates are eliminated again. The union operation could be compared to an *or*-operator. It does not matter if a tuple appears in one table or in both, it forms part of the resulting relation.

Set-Difference Operation

If you need all tuples which are part of one relation but not in another you use the **set-difference operation**. Therefore, getting all persons who are customers but not employees is possible by writing:

$$\pi_{FirstName, LastName}(CUSTOMER_{new}) - \pi_{E_FirstName, E_LastName}(EMPLOYEE_{new})$$

In the set-difference operation the two affected relations underly the same conditions as relations in the union operation.

Set-Intersection Operation

The **set-intersection operation**, which can be compared with the *and*-operator is not a fundamental operation of the relational algebra, but it is an additional operation which could also be defined by the means of fundamental operations. With the binary set-intersection operation you can select all tuples which are part of both affected relations. You could get a relation with all tuples which are in the CUSTOMER relation and in the EMPLOYEE relation by writing:

$$\pi_{FirstName, LastName}(CUSTOMER_{new}) \cap \pi_{E_FirstName, E_LastName}(EMPLOYEE_{new})$$

If using the fundamental operations of the relational algebra the same operation could be written in the following way:

$$\pi_{FirstName, LastName}(r) - (\pi_{FirstName, LastName}(r) - \pi_{E_FirstName, E_LastName}(s))$$

while $r = CUSTOMER_{new}$ and $s = EMPLOYEE_{new}$.

To summarize the union operation selects all tuples which are either in one or in both affected tuples. The set-difference operation selects those which are in one specified but

not in another specified relation and the set-intersection operation filters out all tuples which are part of both affected relations.

Cartesian-Product Operation

At the beginning of this chapter it was mentioned that every relation is a subset of the cartesian product of the values that are part of the attributes' domains. This concept which is relevant for the definition of a relation, is also possible for any two relations that are combined. With the **cartesian-product operation** of two relations r_1 and r_2 , each tuple of r_1 is combined with each tuple of r_2 . So by using the cartesian-product operation on the two relations CUSTOMER and ORDER, you get a relation with all possible combinations of the tuples of CUSTOMER and ORDER. The operation would be noted as follows:

$$CUSTOMER \times ORDER$$

Since each tuple from r_1 is combined with each tuple from r_2 there are $n_1 \times n_2$ ways of choosing a pair of tuples, while n_1 is the number of tuples in r_1 and n_2 is the number of tuples in r_2 .

For the cartesian-product operation it should be distinguished by the notation from which relation the attributes originally came from, as there could be attributes which occur in both relations. That is why the original relation before the attributes is noted in the queries. If an attribute name is only in one relation, the relation name before the attribute name could be left out, too.

To find all *OrderId* numbers of a specific customer, you could use the cartesian-product operation. If you want the orders of the customer with the *CustomerId* 0002, you need at first all tuples of the cartesian product relation that have the customer ID 0002:

$$\sigma_{CustomerId=0002}(CUSTOMER \times ORDER)$$

In each tuple of the resulting relation there are two CustomerIDs, Customer.CustomerID and Order.CustomerID. But only if both IDs are the same, the customer of any tuple t_i has really the OrderID of the tuple t_i . So to really get the right set of tuples the customer ID in the ORDER relation must be equal to the customer ID in the CUSTOMER relation:

$$\begin{aligned} &\sigma_{Customer.CustomerID=Order.CustomerID} \\ &(\sigma_{CustomerId=0002}(CUSTOMER \times ORDER)) \end{aligned}$$

In Figure 2.6 and 2.7 you see the difference between the two queries above.

Since operations can be combined, only the first and last name and the order id could be projected by writing:

$$\begin{aligned} &\pi_{CUSTOMER.FirstName, CUSTOMER.LastName, ORDER.OrderID} \\ &(\sigma_{Customer.CustomerID=Order.CustomerID} \\ &(\sigma_{CustomerId=0002}(CUSTOMER \times ORDER))) \end{aligned}$$

Natural Join Operation

The **natural join operation** is again an additional operation which only simplifies a fundamental operation of the relational algebra, namely the just discussed cartesian product operation.

Customer. CustomerID	Customer. FirstName	Customer. LastName	Customer. City	Customer. Country	Order. OrderID	Order. CustomerID	Order. EmployeeID	Order. OrderDate	Order. ShipCity	Order. ShipCountry
0001	Maria	Anders	Berlin	Germany	1501	0002	12	1997-11-11	Reims	France
0001	Maria	Anders	Berlin	Germany	1145	0002	2	1997-11-13	Paris	France
0002	Ana	Trujilo	London	UK	1501	0002	12	1997-11-11	Reims	France
0002	Ana	Trujilo	London	UK	1432	0004	3	1997-11-11	Salzburg	Austria
0002	Ana	Trujilo	London	UK	897	0001	5	1997-11-13	Bern	Switzerland
0002	Ana	Trujilo	London	UK	2221	0001	5	1997-11-13	Bruxelles	Belgium
0002	Ana	Trujilo	London	UK	1145	0002	2	1997-11-13	Paris	France
0002	Ana	Trujilo	London	UK	521	0005	9	1997-11-11	Vienna	Austria
0003	Antonio	Moreno	Mannheim	Germany	1501	0002	12	1997-11-11	Reims	France
0003	Antonio	Moreno	Mannheim	Germany	1145	0002	2	1997-11-13	Paris	France
0004	Hanna	Moos	London	UK	1501	0002	12	1997-11-11	Reims	France
0004	Hanna	Moos	London	UK	1145	0002	2	1997-11-13	Paris	France
0005	Victoria	Ashworth	Marseille	France	1501	0002	12	1997-11-11	Reims	France
0005	Victoria	Ashworth	Marseille	France	1145	0002	2	1997-11-13	Paris	France
0006	Elizabeth	Lincoln	null	null	1501	0002	12	1997-11-11	Reims	France
0006	Elizabeth	Lincoln	null	null	1145	0002	2	1997-11-13	Paris	France

Abbildung 2.6: Result of $\sigma_{CustomerID=0002}(CUSTOMER \times ORDER)$

Customer. CustomerID	Customer. FirstName	Customer. LastName	Customer. City	Customer. Country	Order. OrderID	Order. CustomerID	Order. EmployeeID	Order. OrderDate	Order. ShipCity	Order. ShipCountry
0002	Ana	Trujilo	London	UK	1501	0002	12	1997-11-11	Reims	France
0002	Ana	Trujilo	London	UK	1145	0002	2	1997-11-13	Paris	France

Abbildung 2.7: Result of $\sigma_{Customer.CustomerID=Order.CustomerID}(\sigma_{CustomerID=0002}(CUSTOMER \times ORDER))$

A cartesian-product operation usually contains some select operations, like the one above where the name and the order ID's of all customers with a given CustomerID are wanted. Since the query for the cartesian-product operation is relatively long, the natural join operation can be used. This operation is noted by the join symbol \bowtie . So the natural join operation can be used to combine a cartesian product operation with certain selections [19]. This operation is an abbreviation for forming a cartesian product of two relations and filtering out all tuples where the attributes which appear in both of the original relations have the same value. So to show all customers with the orders that really belong to them, you write:

$$CUSTOMER \bowtie ORDER$$

Other selections can be used like they are used for the cartesian product operation. For example

$$\pi_{CUSTOMER.FirstName, CUSTOMER.LastName, ORDER.OrderID}(\sigma_{CustomerID=0002}(CUSTOMER \bowtie ORDER))$$

has again as a result a relation from Figure 2.7.

Formally the natural join operation can be defined as follows [19]: The natural join of r and s , denoted by $r \bowtie s$, is a relation on schema $R \cup S$. Furthermore it is defined as:

$$r \bowtie s = \pi_{R \cup S}(\sigma_{r.A_1=s.A_1 \wedge r.A_2=s.A_2 \wedge \dots \wedge r.A_n=s.A_n} r \times s)$$

where $R \cap S = \{A_1, A_2, \dots, A_n\}$ This operation is one of the most important relational database operations in practice and theory.

Division Operation

Another additional binary operation is the **division operation**. To illustrate when this operation is used, following example has been chosen:

If you for example need all customers' last names whose orders were shipped to **all cities** in France where orders have ever been shipped to, you would first need a relation with all cities in France where orders have been shipped to:

$$\pi_{ShipCity}(\sigma_{ShipCountry='France'}(ORDER))$$

Furthermore you need a relation where all customers' last names with the cities, their orders have been shipped to, are shown. You get such a relation with the following query:

$$\pi_{CUSTOMER.LastName, ORDER.ShipCity}(CUSTOMER \bowtie ORDER)$$

The two resulting relations of the queries above are shown in Figure 2.8. If you want to get the wanted names eventually, the division operation is made on these two relations:

$$\pi_{CUSTOMER.LastName, ORDER.ShipCity}(CUSTOMER \bowtie ORDER) \div \pi_{ShipCity}(\sigma_{ShipCountry="France"}(ORDER))$$

The result is also shown in Figure 2.8. So the customer with the last name *Trujilo* is the only person with the asked characteristics. That is because *Trujilo* is the only customer who got his orders shipped to **all cities** of the relation in Figure 2.8 (a).

The division operation can be formally defined as well. The resulting relation $t(T)$ of a division operation on two relations $r(R)$ and $s(S)$ has the schema $T = R - S$. So T has all attributes of R , that are not in S . That definition implies that the attributes in S are a subset of the attributes in R , $S \subseteq R$.

A tuple v is only in the resulting relation t if the values in v appear in a tuple of r in combination with every tuple in s .

a)

ShipCity
Reims
Paris

b)

LastName	ShipCity
Anders	Bern
Anders	Bruxelles
Trujilo	Reims
Trujilo	Paris
Moos	Salzburg
Ashworth	Vienna

c)

LastName
Trujilo

Abbildung 2.8: (a) Result of $\pi_{OrderID}(\sigma_{ShipCountry="France"}(ORDER))$
(b) Result of $\pi_{CUSTOMER.LastName, ORDER.OrderID}(CUSTOMER \bowtie ORDER)$
(c) Result of the division operation

Aggregate Functions

A further extension of the relational algebra operations are the **aggregate functions**. There are different aggregate functions which all get a set of values and return a relation with a single value as result. The set which is used by the function does not have to be a set of distinct values, which means that a value can appear more than once in the set.

The aggregate function **sum** returns the sum of the taken values, while the function **avg** for example returns the average of a set of values. Furthermore **count** returns the number of values, the functions **min** and **max** return the minimum and the maximum of a set of values.

For instance, to get the number of all orders, you would write:

$$\mathcal{G}_{count(OrderID)}(ORDERS)$$

To get the number of customers who made an order, an addition is needed. Due to the fact that a customer can make more orders than one, there are often more tuples with the same customer ID. To get the right number of customers, you write:

$$\mathcal{G}_{count-distinct(CustomerID)}(ORDERS)$$

With the *distinct* addition duplicates are eliminated and each customer is counted only once.

While using aggregate functions on relations, there is the possibility to divide the relation into groups. For example the `EMPLOYEE` relation can be divided into groups based on the countries orders where shipped to. Then the number of orders for each group could be returned by the aggregate function *count*. To do so, you write:

$$\text{ShipCountry} \mathcal{G}_{\text{count}(\text{OrderID})}(\text{ORDERS})$$

The result for this operation is shown in Figure 2.9. So the resulting relation consists of following attributes:

- the attribute on which the groups are based, that is to say the part in the left-hand subscript of \mathcal{G}
- one attribute for each aggregate function in the resulting relation which are in the right-hand subscript of \mathcal{G}

ShipCountry	count (ShipCountry)
France	2
Austria	2
Switzerland	1
Belgium	1

Abbildung 2.9: Result of $\text{ShipCountry} \mathcal{G}_{\text{count}(\text{OrderID})}(\text{ORDERS})$

3 Database-System Architectures

The architecture of a database system can be distinguished into centralized and distributed systems. Centralized systems run on a single computer system and are not connected to other computer systems. This definition does not imply that the system consists of only one computer. The difference to distributed systems is that the database (i.e. data) is not distributed over more than one site. So centralized systems are also systems which are on one company site, while the single parts of the system can be scattered over several buildings and over a great areal. Most of the centralized systems at the moment are based on the **client-server** architecture which will be explained in the next part.

Distributed database systems are distributed over several sites, while each site has a local database system and usually operates on that database. But in distributed database systems there exist **global applications**, which means that there are data accesses which concern more than one local database and more than one site.

As database systems can be very large and there are applications which must query these databases, usual database systems like client-server databases are often not fast enough. For that reason **parallel systems** have been improved in the last years. Though, in parallel systems many operations can be performed at the same time. There are different architectures of parallel database systems which will also be explained below.

3.1 Centralized Architectures

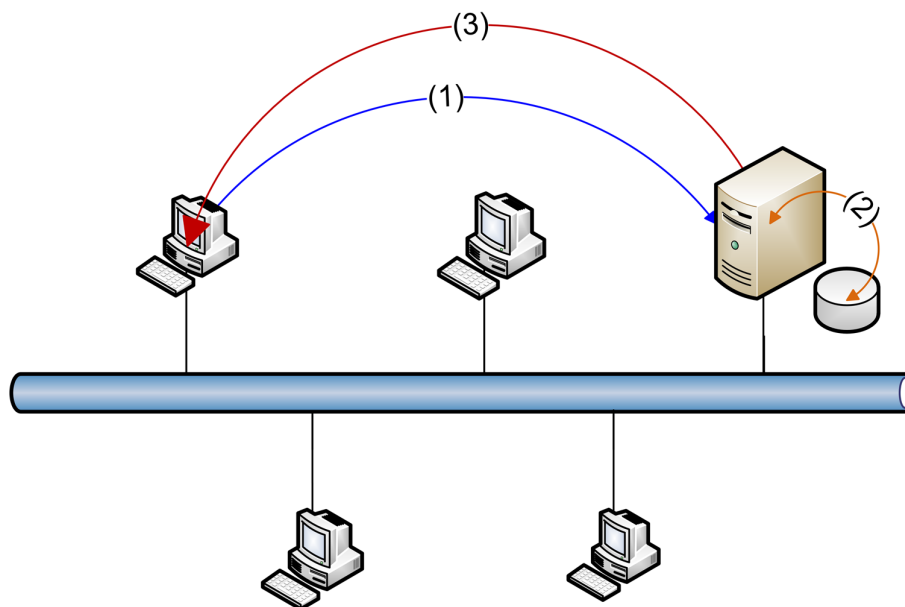


Abbildung 3.1: The process of a client-server database operation with a two-tier architecture

Systems can be split into a **back end** and a **front end**. The back end is the program on the server and manages the access structures, query evaluation and optimization, con-

currency control, and recovery [19]. Moreover, the back end is closer to the system than the front end which is closer to the user.

The front end is *served* by the back end, so it gets data from the back end. The front end consists of the user interface. Furthermore it is responsible for providing tools for reporting, data mining and analyzing. The back end and the front end are either connected via SQL or an application program. Both parts of a client-server database systems, the back end as well as the front end, could also be part of the same personal computer. But it is also a client-server system if the client is a computer for itself and is connected over a network to the server. Usually the computational power to perform the asked operations is provided by the server, so the back end. Figure 3.1 shows such a system, which adopts a **two-tier architecture**: At first the asked operation which the user, a client, wants to perform, is entered. So the instruction is transferred to the server (1) which is connected to the data and which performs the asked operation (2). The result is then sent back to the client (3).

It is also possible that the operation is performed by the client. In such a system the server is only a **data server**. That means that not just the result is returned to the client but the server returns the data which is required to perform an asked operation. For instance, if a client wants to perform a select operation, the whole table on which this selection is performed is returned by the server and the operation is then carried out by the client. That implies that the client requires full back end functionality. If there is a data server system, the costs of communication is very high compared to systems where the server performs the operations.

If there are many users who use the database system, there is mostly a special architecture, where the front end talks to an application server and the application server is the *client* for the back end. Such systems adopt a so called **three-tier architecture**. That is because the server would have to perform a lot of operations at the same time if there are many users and so the time for performing could last too long. Though, by the help of a further application server the processes can be optimized. As you see in Figure 3.2 the user is not directly connected to the database server anymore. The application server performs a big part of the tasks which would be done by the database server in a two-tier architecture and the database server is just directly connected to the application server. So the application server acts like a client, which means that independently from the number of users, the database server is always connected to one client only. So again the application server is connected to the database server over an application program which is usually based on SQL. But the real front end, which is seen by the user, is connected with the application server in another way. Their interface is either a standard network protocol or a self developed protocol for a specific use.

3.2 Distributed Systems

A distributed database system, as already mentioned above, is a system which is usually spread over more than one site and consists of several processors which all have their own local database, but also interconnect with the other processors and their data. Defining distributed database systems is rather difficult. For a system to be distributed the following two aspects must be true [28]:

1. The data is **distributed**, which means that not the whole data is in one node. Usually theses systems are distributed over several sites, although that is not a mandatory requirement. A distributed database system could also be spread over several nodes which are located at the same physical site.
2. There must be a difference between a set of local databases and a distributed database, where the local parts must be connected somehow. This characteristic is called **logical correlation**.

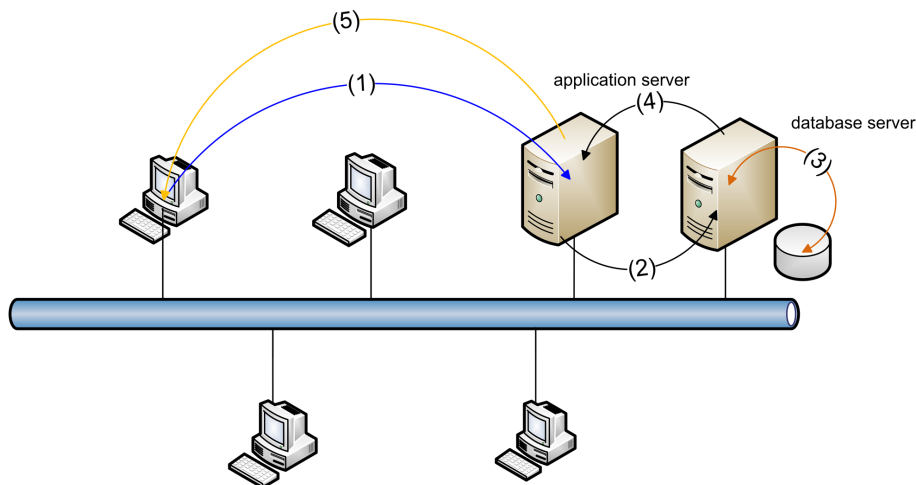


Abbildung 3.2: The process of a client-server database operation with a three-tier architecture

But this definition is still vague and these two facts do not always distinguish distributed database systems from local ones. The fact which is really crucial to distinguish a distributed database system from a set of centralized systems, is that **global applications** exist. Such applications access data for the performed operations from more than one site.

A good example for explaining distributed database systems is a bank which has several branches on different sites. Usually in each branch most of the transactions and operations can be carried out on the local database. But if for instance, a transaction is performed with an account, which belongs to another branch, the local system has to access data from another database, too. The existence of such applications is crucial for a system to be called distributed. Though, it does not matter if nearly most of the operations and transactions are carried out locally.

As mentioned above, the branches do not have to be at different sites. What is really important for distributed databases is that every branch has its own processor with own local data. It is possible that there is a computer center where the processors and databases of all branches are together. But to be a distributed system each branch must still have its local applications and its local data. So one local branch database can be local although the data and the terminals are geographically distributed. Furthermore a system can be distributed although the local data of all branches is at one site.

A system which would not be seen as distributed is for example: All local databases are connected via an interconnection network to an application server. Furthermore, all the terminals of all branches are just connected to the application server. Now the terminals just communicate with the application server which is directly connected to all local databases. In such an architecture the access structure for global as for local applications would be the same for the branches. Due to that characteristic this system would not be distributed.

3.3 Differences between Distributed and Centralized Database Systems

There are some differences between distributed and centralized database systems, which are very important when setting up a distributed system or changing a set of local databases to a distributed database system.

One of the most important reasons for inventing database systems in general was to get **centralized control** over the data. In distributed database systems the idea of centralized

control is a bit different. When having a centralized system, there is one function of a database administrator. But since a distributed database system consists of the local parts, there are often the so called local database administrators and there also can be global databases administrators. The systems can be very different from each other, so there sometimes is no global database administrator at all and the local database administrators are responsible. In other cases, global database administrators can have a high degree of responsibility.

The **data independency** which describes the fact that application programmers are not affected by the physical organization of data as there is a conceptual view of data, is very important for centralized database systems as well as for distributed database systems. Indeed, for distributed database systems there exists another independency, namely the **distribution transparency**. Due to this concept it is possible to write programs, while it is completely irrelevant that the database is distributed.

Avoiding to have redundant data was also one of the main reasons for inventing database systems. The **reduction of redundancy** is also an important feature, in distributed database systems. Although for distributed database systems redundancy of data also has its advantages as it makes the single sites more local and global applications become less important. Furthermore, if one site has a failure or loses the data, it would still be available if it was redundant. In general, redundancy is the better the more important retrieval of the data is and the fewer updates are performed on that data. The reason for this statement is that an update must be performed on all copies of the affected data while for retrieving data any copy can be taken.

The issue of **integrity, recovery and concurrency control** in distributed databases was solved by the help of **transactions**. A transaction is a sequence of operations and when a transaction is started it is either performed entirely or no data is changed at all. This concept ensures that there are no different copies of the same data in different sites. For instance, if a branch of a bank performs a funds transfer, either all the affected data is changed or if there was a failure nothing would happen. Due to this concept it is not possible that for example the money from one account is deducted but is not transferred to the other account.

3.4 Distributed Database Management Systems

A distributed database management system consists of [28]:

- the database management component (DB)
- the data communication component (DC)
- the data dictionary (DD), which represents information about the distribution of data in the network
- the distributed database component (DDB)

If one site wants to access data from another site, the DDMBS is responsible for performing the operation at the right site and routing the result to the enquiring site. There are two different ways of access to data of another site. The first way is that the DDBMS routes automatically to the right site and performs the task. Finally the result is returned to the first site. This approach makes it really irrelevant if the database system is distributed or not. Therefore distribution transparency is provided. The other possibility is that there is an auxiliary program at the site the data is needed from. That approach is often more efficient as the auxiliary program is able to manipulate several records at the same time while by the most DBMS this feature is not provided and that is why the second

way of accessing remote data is often the more efficient way, although the distribution transparency is not provided.

Distributed database systems are divided into **heterogenous** or **homogenous** systems. There are several possibilities why distributed systems could be heterogenous, but concerning distributed database systems, one type of heterogeneity really affects the system, namely if the DBMS are heterogeneous. That case occurs if there are at least two different and autonomous DBMS in a distributed system. This kind of database system is called **federated** Database system. When building a distributed system the same DBMS should be used at all sites, but as often local databases are combined to a distributed database system, the DBMS of the sites can differ from each other. The problems between different DBMS are very hard, especially when different data models are used, so the feature of homogeneity or heterogeneity is very important.

3.5 Reference Architecture of Data Dictionary and Fragmentation

branch-name	account-number	customer-name	balance
Hillside	A-305	Lowman	500
Hillside	A-226	Camp	336
Valleyview	A-177	Camp	205
Valleyview	A-402	Kahn	10000
Hillside	A-155	Kahn	62
Valleyview	A-408	Kahn	1123
Valleyview	A-639	Green	750

Abbildung 3.3: *Account Relation*

Due to distribution transparency which is provided by a distributed database system, it has a special reference architecture whose top level is the global schema. The second level is the fragmentation schema and the third level is the allocation schema. The first three levels are site independent schemas. The **global schema** shows the data as if the database was not distributed. But to allow the sites to work with the data, the data must be split in several parts, called **fragments**. The connection between the global relations and their fragments is defined in the **fragmentation schema**. The third level of the site independent schemas, the **allocation schema**, defines at which sites the different fragments can be found, since a global relation could be distributed over several sites. Furthermore the allocation schema defines whether the fragment is redundant or not. The fourth schema is already site dependent and is called **local mapping schema**. It defines for each local database system, where the physical image to the fragments can be found which they manipulate.

Due to the fact that global relations have to be divided into fragments, queries which operate on global relations also have to be transformed into queries which operate on fragments. The resulting query which operates on fragments is called **canonical expression**, which could be directly executed on the distributed database system. But since the canonical expression is often very inefficient, a lot of transformations are possible to get a simpler expression.

An example for horizontal and vertical fragmentation can be seen in figure 3.4 and 3.5. Given the relation **account** depicted in figure 3.3 the horizontal fragmentation of this relation is done by the two select operations $account_1 = \sigma_{branch-name="Hillside"}(account)$ and

$account_2 = \sigma_{branch-name="Valleyview"}(account)$. The union of these results is the original account relation. The vertical fragmentation is done by project operations and the results $deposit_1$ and $deposit_2$. A natural join with the **tuple-id** attribute on the result relations produces the account relation.

account ₁	branch-name	account-number	customer-name	balance
	Hillside	A-305	Lowman	500
	Hillside	A-226	Camp	336
	Hillside	A-155	Kahn	62

account ₂	branch-name	account-number	customer-name	balance
	Valleyview	A-177	Camp	205
	Valleyview	A-402	Kahn	10000
	Valleyview	A-408	Kahn	1123
	Valleyview	A-639	Green	750

Abbildung 3.4: $account_1 = \sigma_{branch-name="Hillside"}(account)$
 $account_2 = \sigma_{branch-name="Valleyview"}(account)$
 $account = account_1 \cup account_2$

deposit ₁	branch-name	customer-name	tuple-id
	Hillside	Lowman	1
	Hillside	Camp	2
	Valleyview	Camp	3
	Valleyview	Kahn	4
	Hillside	Kahn	5
	Valleyview	Kahn	6
	Valleyview	Green	7

deposit ₂	account-number	balance	tuple-id
	A-305	500	1
	A-226	336	2
	A-177	205	3
	A-402	10000	4
	A-155	62	5
	A-408	1123	6
	A-639	750	7

Abbildung 3.5: $deposit_1 = \pi_{branch-name, customer-name, tuple-id}(account)$
 $deposit_2 = \pi_{account-number, balance, tuple-id}(account)$
 $account = deposit_1 \bowtie deposit_2$

Summary for Distributed Database Systems

Distributed database systems are a very large and complex topic. First of all it is not easy to distinguish between centralized and distributed systems, but the crucial point is whether global applications between local databases exist, or do not. Furthermore there are some differences between centralized and distributed databases which are important and which affect the system. There is another important difference between heterogeneous and homogeneous systems, which makes a big impact. It is important that distributed systems have not only the local DBMS, but also distributed database management systems which are responsible for the communication between the sites. Additionally, as distribution transparency is an important feature for distributed databases, the reference architecture of distributed databases is very important. It defines how global relations are divided into fragments and how the fragments can be found by the local sites.

3.6 Parallelism in Database Systems

As mentioned above database systems can be very large, but they still must be queried in an appropriate way. To find out if the database system has become too large for the used

physical system, there are two main performance numbers:

1. throughput
2. response time

The **throughput** of a system is the number of tasks that can be finished in a given time interval, whereas the **response time** is the required time for completing one single task [19]. Using a client-server system for querying large databases, these two performance numbers would have bad values. For that reason parallel systems are needed. Parallel systems can perform many operations simultaneously, while non-parallel systems perform operations sequentially.

There is a separation into coarse-grain parallel machines and fine-grain parallel (or *massively parallel*) machines. Today, most of the general-purpose computer systems are already **coarse-grain parallel** machines as they have multiple processors. Machines having such a parallelism have a higher throughput than machines without any parallelism, but the response time for a single task remains the same. That is because the single tasks are not distributed on the processors, but each processor can perform a single operation at the same time. So more operations can be performed in a given time interval, but nevertheless a single task is not performed faster than on a non-parallel machine.

Fine-grain parallel machines consist of a large number of processors. Furthermore, the single operations can be parallelized on the processors. That means that an operation of one user is carried out by more than one processor, so it is divided into subtasks. The advantage of fine-grain parallelism is that the amount of response time of an operation declines and of course due to a shorter response time, more tasks can be completed in a given time interval; so the throughput increases.

As described in DeWitt [4] relational queries are ideally suited to parallel execution because they consist of uniform operations applied to uniform streams of data. The operators can be composed into parallel dataflow graphs due the fact that each operation produces a new relation. Two kinds of parallelism can be distinguished:

- pipelined parallelism
- partitioned parallelism

The output from one operator can be streamed into the input of the other operator, therefore the two operators can work in series giving **pipelined parallelism**. **Partitioned parallelism** is partitioning the input data over multiple processors and splitting one operator into many independent operators each working on a part of the input data. The two kind of parallelism are depicted in figure 3.6.

Parallel database systems can have different architectures, while these architectures influence the efficiency of the systems. Stonebraker [29] suggested the following simple taxonomy for the spectrum of designs:

- Shared memory
- Shared disk
- Shared nothing

In **shared memory** architectures, see figure 3.8, the memory is shared among the processors and disks. Due to that sharing a very efficient communication between processors is possible. But as the processors have to share the bus, there is a maximum limit of processors which can be added to the system. Exceeding that limit does not make sense as the system would not get faster due to long waiting times for accessing the memory. Current shared-memory machines are usually scalable to 64 processors.

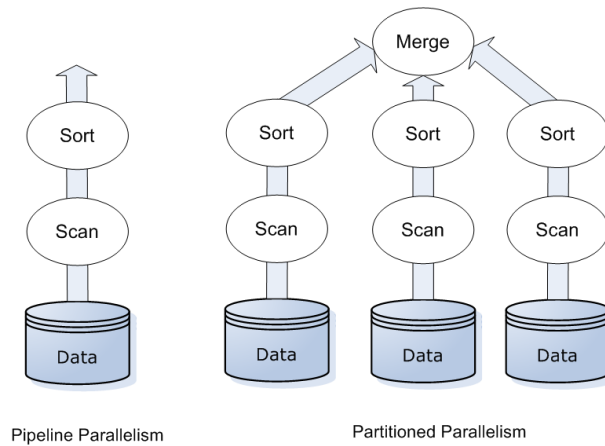


Abbildung 3.6: Pipeline- and Partitioned Parallelism

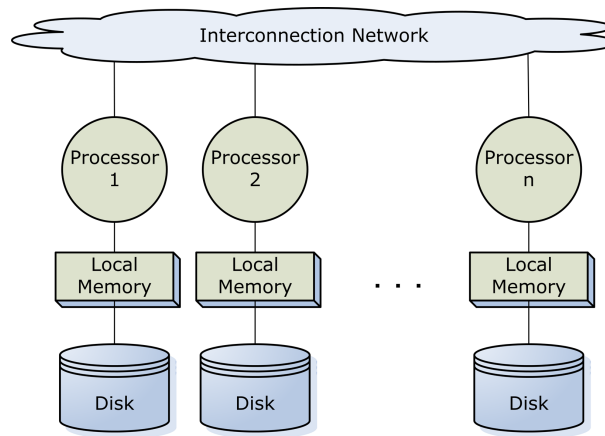


Abbildung 3.7: Shared-Nothing Architecture

Processors of systems with **shared disk** architecture can access all disks directly over a interconnection network. In figure 3.9 the shared memory architecture is depicted. Though, the processors have own memories in contrast to the shared-memory systems. This means that there is no bottleneck at the memory bus anymore. But in this system there is a maximum limit of processors due to the interconnection with the disk subsystem, so there is still a scalability problem. In such architectures more processors can be added than in shared-memory machines. A big advantage of shared disk systems is that they provide a degree of fault intolerance. So if a processor fails, the task can be carried out by any other processor since they have direct access to the disks. In return the communication between the processors is slower than in shared memory architectures.

Shared nothing systems do not provide any shared components between the processors. Each processor has its own local memory and its own local disks. See figure 3.7. Due to that feature, there are no bottlenecks and the machines can support a large number of processors. The disadvantage of such machines is that the communication cost for them is higher than for the other systems as every non-local access works over an interconnection network.

Systems with a **hierarchical architecture** are a combination of the characteristics of the architectures above. Though, in a hierarchical system there exist more levels. At the top level the system could have a shared nothing architecture, while the basis of such a system could be a shared memory system. In the middle there might be a level with

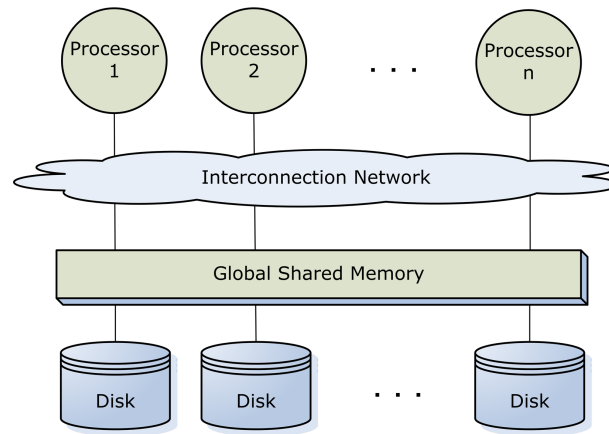


Abbildung 3.8: Shared-Memory Architecture

shared-disk architecture.

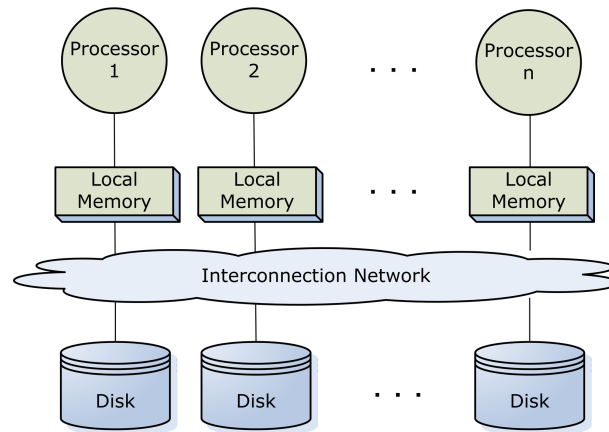


Abbildung 3.9: Shared-Disk Architecture

On the other hand a **NUMA**(Non-Uniform Memory Access) Architecture is a specialized memory architecture for multiprocessor based systems.

In a **NUMA** architecture multiple processors are grouped and these groups have their own local memory attached over a memory bus. These grouped processors are linked together with a high speed link interface. Therefore all the processors can access other memories over this link interface. Therefore the grouped processors have very low latencies for accessing their local memory and higher latencies for accessing the other memories over the high speed interface. This is the reason for why this architecture is called a Non-Uniform Memory Access.

The **NUMA** architecture is in contrast to the **SMP** (Symmetric Multi-Processing) where all memory access is done over a common single memory bus. In an **SMP** architecture the latencies for the memory access are the same for all processors.

3.7 Parallelism Goals and Metrics: Speedup and Scaleup

In this thesis we use the definition of the key properties for parallel systems as defined in DeWitt [4, 30].

There are two properties for ideal parallel systems:

1. (1) *linear speedup*: Twice as much hardware can perform the task in half the elapsed time.
2. (2) *linear scaleup*: Twice as much hardware can perform twice as large the task in the same elapsed time.

Speedup

The speedup describes the effect on processing time of adding computing nodes. This can be defined by the ratio of the time used for a fixed job run on a small system and the time used of the same job on a larger system. The speedup can be measured as:

$$Speedup = \frac{Small_System_Elapsed_Time}{Big_System_Elapsed_Time} \quad (3.1)$$

Speedup holds the problem size constant and grows the system size. So the speedup describes how much an operation is faster if it is performed parallel than if it was performed sequentially. The ideal speedup of an algorithm is linear, which means by increasing the resources of a smaller system M_S n times, the speedup of the new larger system M_L is n . As a **linear speedup** is not always achievable, linear speedup is a very good measure. Though, a speedup which is less than the ideal is called **sublinear speedup**. The objective when increasing the degree of parallelism of a system is that the needed time for processing a task is inversely proportional to the resources allocated. In other words, the speed of a system should grow proportional to the resources. In Figure 3.10 the difference between the ideal linear speedup and the sublinear speedup is shown. To describe the ideal linear speedup mathematically, the time for executing an operation on the smaller machine is T_S and the execution time on the larger machine is T_L . If the speedup is linear, then the speedup, defined as S , is:

$$S = T_S / T_L$$

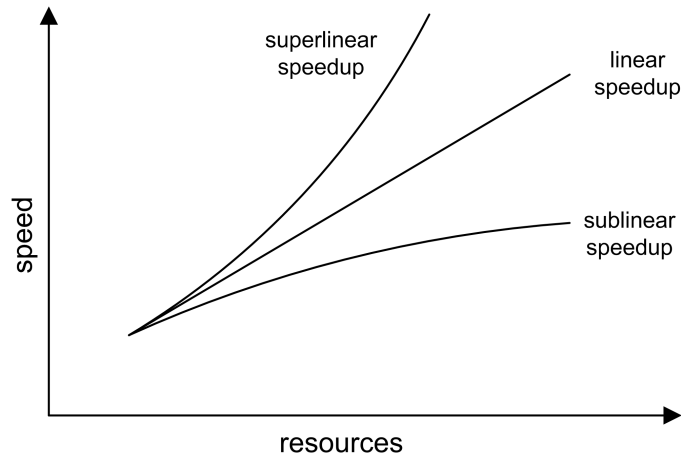


Abbildung 3.10: Difference between linear, sublinear and superlinear speedup [19]

Note that superlinearity speedup can be achieved by a cache effect using different memory hierarchies of the nodes used in the algorithm. As an example the effects of caches on scalability and performance for hash join algorithms are described in [9]. Performing backtracking can also be a reason for superlinearity speedup.

Scale-Up

Scaleup measures the ability to grow both the system and the problem. Scale-up gives an explanation of the scalability of an algorithm, that means if increasing problem size can be practically compensated by increasing resources (e.g. number of processing nodes). This behavior can mathematically be described by the following formula:

$$Scale - up = \frac{Small_Sys_Elap_Time_On_Small_Prob}{Big_Sys_Elap_Time_On_Big_Prob} \quad (3.2)$$

If the scaleup equation evaluates to 1, then the scaleup is linear. The probably most important goal of parallelism is that even if the database and the number of transactions grow, the speed stays acceptable. The scale-up metric shows how able the system is, to perform larger operations in the same amount of time if more resources are provided. More precisely, if there are two machines M_S and M_L and M_L is n times larger than M_S and there are two tasks Q_L and Q_S , where again Q_L is n times larger than Q_S . Q_L is executed by M_L and Q_S is executed by M_S . As long as both machines need the same amount of time to perform their tasks, the parallel system M_L demonstrates a **linear scale-up** on Q_S . So the amounts of time the machines need to execute the task are T_L and T_S . If $T_L = T_S$ the scale-up is linear. But if the amount of time T_L which the machine M_L needs is bigger than the needed amount of time T_S of machine M_S , a **sublinear scale-up** is demonstrated by M_L on Q_S . In Figure 3.11 it is shown that as long as T_S/T_L is constant by growing problem size, the scale-up is linear. The goal is that the scale-up which a parallel system demonstrates is preferably linear.

Barriers to linear speedup and linear scaleup are the threats **startup**, **interference** and **skew**. The first factor is made up by the **start-up cost**; When initiating an operation on a processor there are start-up cost associated which could have a bad impact on speed-up and scale-up. The bad impact would especially occur if there were a lot of processes.

Another factor is the **interference** which was already mentioned in the parallel database architectures above. Bottlenecks could occur if not a shared-nothing system was used. Due to waiting times the two performance numbers could also be badly affected.

The last factor which influences speed-up and scale-up in a negative way is that is often very difficult to break down a single task into same sized parts. This factor is called **skew** as the way the parts are then distributed on the processors is skewed [19]. If skew occurs the speedup of tasks running could be badly impacted, dependent on how big the biggest part of the original task is.

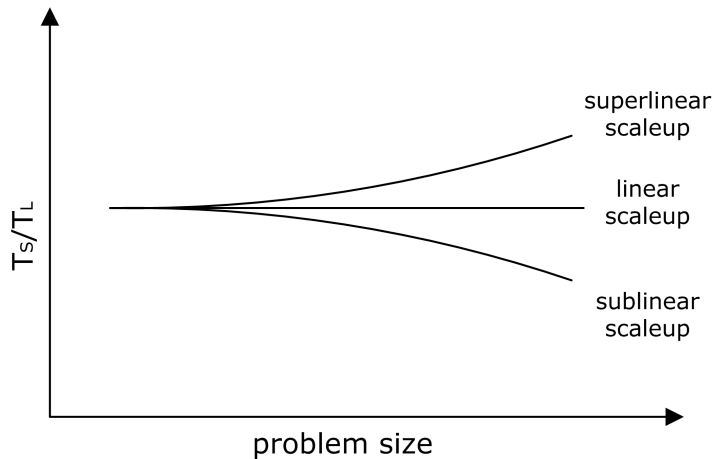


Abbildung 3.11: Difference between linear, sublinear and superlinear scale-up [19]

3.8 Parallel Database Architectures

The work of Bitton et al. [18] is based on a so called generalized multiprocessor organization, which basically comprises a **homogenous** classical supercomputer architecture, where every working node shows the same characteristics:

1. processing power
2. disk I/O,
3. network interconnect bandwidth

That means all these characteristics are the same for all nodes. So for the evaluation of the parallel operations on the generalized multiprocessor organization only the following components are considered:

1. a set of general-purpose processors,
2. a number of mass storage devices,
3. an interconnect device for connecting the processors to the mass storage devices via a high-speed cache

The definition of such an organization is shown in Figure 3.12.

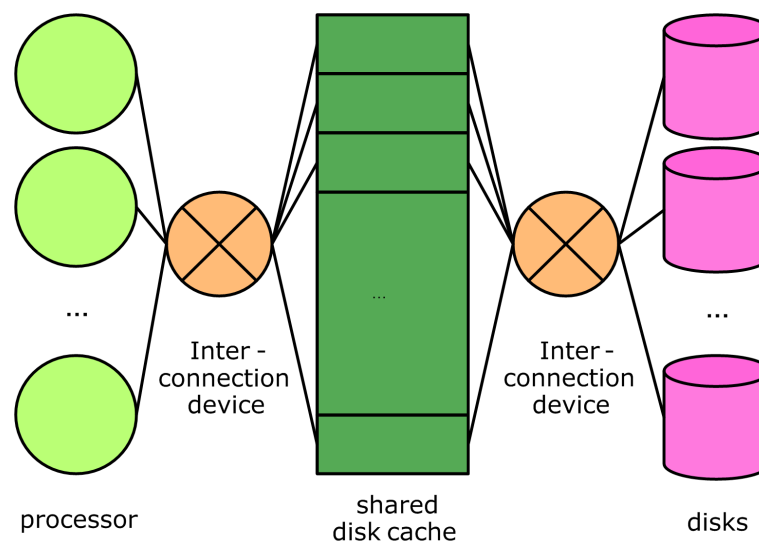


Abbildung 3.12: Generalized Multiprocessor Organization

3.9 Heterogeneous Database Architectures

In chapter 2 the issue of heterogeneity in distributed database systems was mentioned. Here, this issue especially concerning heterogeneity of Database Management Systems (DBMS), will be further discussed. In particular, the problems and the possible solutions of this kind of heterogeneity will be covered.

It is a main goal of heterogeneous DDBSs to provide transparency concerning data distribution but also concerning heterogeneity. That means that there should be a view of the DDBS which makes it irrelevant that the system is distributed and heterogeneous. If different DBMSs are used, this feature is not easy to achieve. Ceri et al. [28] claims in his work that the most convenient way to reach transparency is to select a common data

model and data manipulation language (DML) for all component DBSs. But that selection leads to two main problems, which concern data conversion and program conversion.

In order to make it possible that all component DBs have the same data model and DML, an appropriate selection must be made. The chosen data model and DML should have two main properties. (1) A simple translation from a used DBMS's data model and DML to the chosen data model and DML should be feasible which means that the common data model should be as simple as possible. (2) It must be possible to represent the global, fragmentation and allocation schemata by the means of the selected data model and the DML should own set-oriented primitives. Hence, the relational model and algebra might be good candidates.

The problems concerning the translation to the common data model and DML can be related to data conversion or program conversion. Data conversion problems can be further classified into problems at schema level or at instance level. Problems at **schema level** refer to the fact that for each local DBMS a schema must exist that is equivalent to the schema of the common data model. The **instance level** problems arise due to the automatic conversion of large amounts of data from one representation to another. Two database states are just equivalent if they have equivalent schemata and the stored data represent the same facts. Program conversion problems occur because a conversion between two different DMLs is necessary. Two programs are just equivalent, if they produce the same output for any input provided to them.

Usually heterogeneous DDBS are built bottom-up, meaning that existing systems are linked together to one system. Thus, it is possible that two or more DBMSs represent the same facts. Although the DBMSs might represent the same facts, the descriptions of the facts do not have to agree. Such differences are called conflicts and can be classified in the following way:

- **Name conflicts:** DBMSs describe different facts with the same name (homonyms) or they use different names for the same fact (synonyms).
- **Scale conflicts:** DBMSs use different units of measure
- **Structural conflicts:** DBMSs use different structures to describe the same facts
- **Different levels of abstraction:** DBMSs provide more or less detailed information than the other ones.

If such conflicts occur, databases usually cannot just be altered. Indeed, an **auxiliary database** can be produced. Such auxiliary databases can be used for name or scale conversion, respectively for solving structural and abstraction problems.

There is another kind of problem in heterogeneous DDBS due to **query processing**. Most heterogeneous systems just allow distributed retrievals. Update applications on the other hand are usually done locally. As the updates are done locally, it is possible that inconsistencies of data occur. Then, different policies can be applied for dealing with such data. For instance, the maximum, the average or the minimum could be taken. Another query processing problem is that processing cost can hardly be evaluated. Heterogeneous DDBS consist of different DBMSs and the DBMSs have different performances. Hence, as much as possible should be performed locally as it is possible that some functions are more expensive or not available at remote DBMSs.

The optimization process of queries is responsible for distributing the execution and deciding how and where the query should be executed. Generally, optimization can be subdivided into global optimization and local optimization. The global optimization is responsible for the query execution to be distributed among the different sites of a DDBS. This kind of optimization is just necessary if local execution of the query is not possible. In

contrast to that, the local optimization exists at each site and is responsible for deciding locally the best method for its part of the execution. If the global optimization of an execution is performed after the translation, it has the advantage that there is just one version of the global optimizer. But on the other hand all applications must be translated, even if they could be executed completely at its site of origin, which would lead to unnecessary performance cost. Therefore, there are two more possibilities:

1. **Using a local analyzer:** A local analyzer can decompose the application into local and remote portions. Hence, just the remote portion is translated into the common DML
2. **Classifying applications:** The applications are classified into completely local applications, respectively all other applications.

To summarize, heterogeneity concerning DBMSs can best be solved if a common data model and DML is used. The translation to the common model and DML brings some problems, though. The problems can be due to data conversion or program conversion. A specific problem is the integration of different schemata which represent the same facts using different descriptions. These conflicts can be solved by producing auxiliary databases. Other specific problems are due to query processing. For instance, processing cost in heterogeneous DDBS can hardly be evaluated and are often unnecessarily high. Therefore, optimization of queries is an important issue, especially the point of time when query optimization is performed is crucial since processing cost can be saved.

3.9.1 An Infrastructure for Scientific Grid Computing

EGEE (Enabling Grids for E-Science) [31] (visited June 2009) is an engineering project deploying a complex infrastructure for the scientific community. Until March 2004 the European DataGrid (EDG) project [32] (visited June 2009) was the predecessor of EGEE and was a successful establishment of a functional Grid testbed for more than 20 countries in Europe. The motivation for EGEE is to establish a production-quality Grid for offering reliable Grid services.

The EGEE Structure

The EGEE project was originally proposed by experts in Grid technology from the leading Grid activities in Europe. The project now includes more than 70 project partners organized in twelve partner regions or federations, as shown in figure 2. Furthermore, with the deployment of the EGEE project structure, several of these partners have begun integrating regional Grid efforts in order to provide coordinated resources to the EGEE project. There are several participating resource centers of more than 70 project partners in twelve regions in Central Europe, France, Germany, Switzerland, Ireland, UK, Italy and USA. Pilot initiatives, like the Large Hadron Collider Computing Grid (LCG) [33], serve as a guide for the implementation. These pilots are used to certify the performance and the functionality of the Grid services. The pilots store and analyse petabytes of data from high-energy physics (HEP) experiments in CERN from the Large Hadron Collider [34]. On the current EGEE production service Biomedical Grid applications have also been deployed.

The EGEE Middleware

EGEE uses a re-engineered middleware with his branded name **gLite** [35]. **gLite** is based on best practice experience middleware projects: Globus, Condor, DataTAG and so on. gLite provides a framework for building grid applications across the Internet.

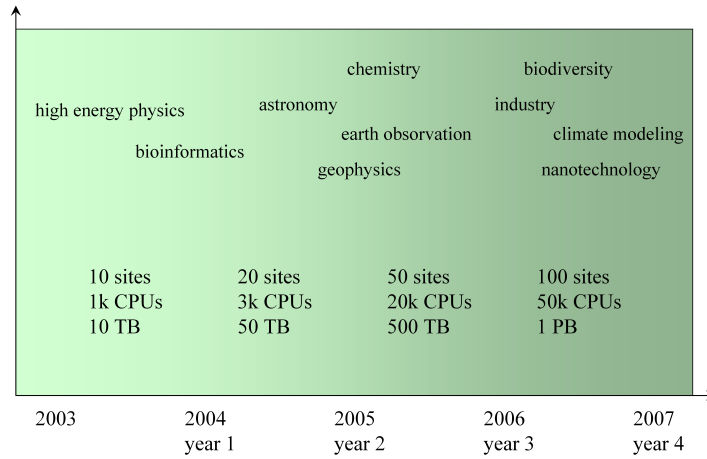


Abbildung 3.13: Schema of the Evolution of the EGEE Grid Infrastructure

In Figure 3.13 the evolution of the **EGEE** Grid infrastructure is depicted. The **EGEE** infrastructure is grown from 10 sites, 1000 CPU's and 10TB Data up to 100 sites, 50000 CPU's and 1 PB data [36].

3.9.2 A Static Heterogeneous Model

The big difference between a Grid environment [37, 38], which is the focus of this thesis, and the Generalized Multiprocessor Organization laid out in section 3.8 is the **heterogeneity** of all included elements, which are processing nodes, interconnect bandwidth and disk performance. For the analytical comparison of the parallel algorithms in focus we restrict our approach to a **simplified** Grid computing organization focusing on the sensitive parameters of the model. These sensitive parameters are processing power, disk and network bandwidth.

We use the term **Static Simplified Grid Organization**, which describes an organization to perform a distributed query on a loosely coupled number of heterogeneous nodes. We define that there is no logical order or hierarchy. There is no logical topology of the nodes (e.g. no master/slave nor an equivalent order). Each node has a fixed number of properties with defined values. The term **Static** is used to describe that the values of each node and the network bandwidth are fixed. That means these values are not changeable during the execution of a query. The sketch of such an organization is shown in Figure 3.14. These assumptions build the basis for our approach to analyze the parallel database operations in a simplified Grid organization.

General architectural overview

The general layout of our architecture is as follows: A number of nodes are connected via a network. A node consists of one ore more CPUs (Central Processing Unit), one or more disks or disk-arrays and a network interface to the interconnect they are connected.

The actual configuration of a node is transparent (that means not "seen" by the outside user). However there exists a database to describe the system at the time of the query execution. All relevant data (parameters and their values) describing the architecture are stored in this database. It is assumed that the following architectural characteristics hold constant during the operation:

- Nodes can perform a dedicated operation (compare, sort and merge two pages).

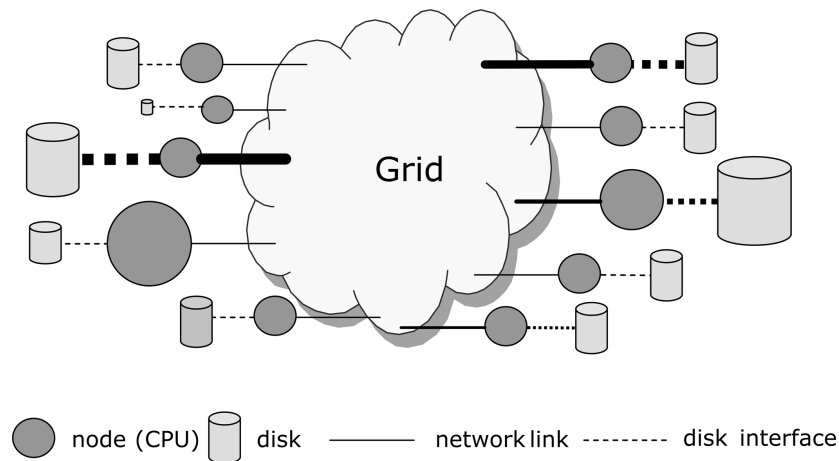


Abbildung 3.14: *Static Simplified Grid Organization*

- Each node has its own mass storage device and the nodes are connected over a network.
- Nodes can send and receive messages.
- The availability and the reliability of each node during the query execution is also guaranteed.

3.10 Peer to Peer Database Architecture

Peer to Peer architectures of database systems are also called **Peer data management systems** (PDBS). Referring to [39] PDBS are on an evolutionary path and can be described as a collection of autonomous local repositories which interact in a peer-to-peer style. The repositories are the peers which all have the same rights. Furthermore peers can join and leave the network without any obligation of carrying out administrative tasks. In [40] pictured peer-to-peer as the most general architecture, where a peer can act as a server, storing a part of the database, but as well as a client, executing application programs and initiating queries. In [41] a similar definition of PDBS is used while in stead of server and client the notion of source and target is used. So every peer is at the same time a source-peer as well as a target-peer.

There are some specific characteristics which PDBS have in general and also in respect to other distributed databases. The first difference in respect to distributed databases concerns the data integration architectures. Usually distributed database systems provide a global schema integrating the local DBS schemas as discussed in section 3.2. On the other hand PDBSs avoid such global schemes, but provide mappings between two sources (Figure 3.15). It is not necessary that all sources are connected as long as the graph illustrating all pairwise mappings connects somehow the source- and the target-peer, no matter how many other peers are in between. In the export schema the peers provide the information they want to share with the outside world. Basically an export schema does not really exist as the dependencies between two peers and therefore the information they want to share is defined in the mapping rules. Due to the global schema of usual distributed database systems it is possible that global requests can be carried out and distributed to all the component DBSs as global subrequests. PDBSs do not have this possibility but a query is submitted to a local peer. Depending on the mapping graph the query is then forwarded to the next peer or not. Furthermore if it is forwarded it can happen in the original form or in a modified way depending on the mapping rules. The

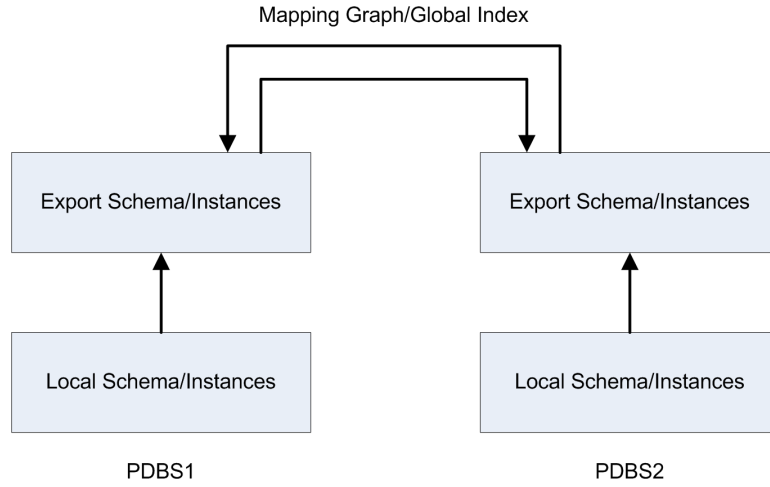


Abbildung 3.15: Data Integration Architecture of PDBS

difference between the two architectures is shown in Figure 3.16. With reference to [42] it is possible that the participation of all peers at query time is necessary to answer a local query. But as peers might have full autonomy and can join and leave the network without problems a complete response to a request is not guaranteed in contrast to global requests.

The distribution of a DBS ranges from centralized architecture over client-server architecture to full-scale distribution. The distribution in PDBS is dependent on the underlying Peer-to-Peer network. [39] classifies existing networks in three categories, namely **pure Peer-to-Peer**, **super-peer systems** and *hybrid systems* while the first one is also called **unstructured** and the latter two are referred to as **structured networks**. The pure Peer-to-Peer systems consist of participants which are all equal and do not store any global indexes and therefore they are full-scale distributed. The super-peer systems on the other hand have some super-peers storing internal indexes where data of other peers are contained. The last class consists of the hybrid systems where servers or clusters are used to store global indexes. Systems which underly structured networks provide a distribution that is classified into client-server architecture and full-scale distribution.

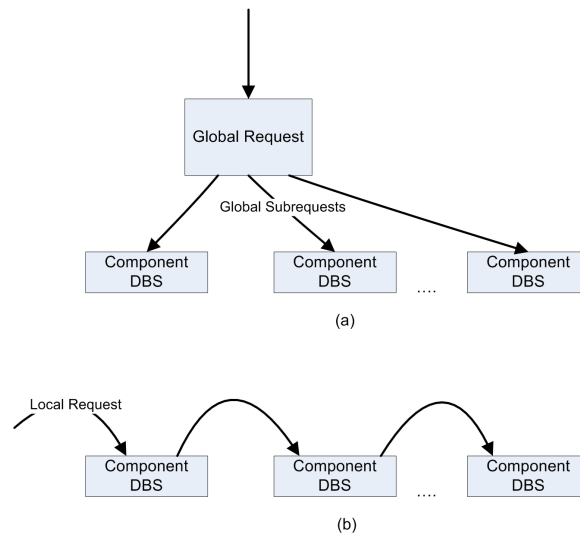


Abbildung 3.16: (a) the distribution of a global request in a usual DDBS. (b) the distribution of a request in a PDBS.

As described above, queries in (unstructured) PDBS are forwarded from peer to peer. Hence, the performance of such a system depends on the mapping path, which can be very large, especially in unstructured networks. That is the reason for why more and more structured networks have been used. The structured networks are based on Distributed Hash Tables (DHT) where hash keys are used to make efficient lookups and therefore efficient routing possible. Moreover, if a structured network underlies the PDBS, a guaranteed answer, that is to say a perfect recall can be guaranteed.

In chapter 4.4.2 it was discussed that data independency for distributed databases is as important as for centralized databases. Concerning Peer-to-Peer systems data independency would be likewise desirable so that the data is independent of its physical location [43], but there are still hardly any systems which provide data independence. A big difference between PDBS and other DBS is that other database systems assume a close world view which means that it can be assumed that the whole relevant data is stored in the database and can be returned if requested. Apart from that, in unstructured PDBS an open world assumption must be made, as nodes can join and leave freely and therefore just certain query answers can be returned [44]. A very big disadvantage of PDBS is that they do not provide support for transactions. Due to the autonomy of the nodes the systems are very loosely-coupled and therefore the issue of integrity, recovery and concurrency control has still not been solved.

Eventually there are Peer-to-Peer centric characteristics which should be discussed. The **degree of coupling** is the first special Peer-to-Peer characteristic. It defines the number of other peers' existences one peer knows about. Usually in PDBS the degree of coupling is not very tight as the nodes can leave and join freely. Therefore, especially unstructured PDBS do not know many of there neighbors. On the other side, in distributed database systems all nodes are known by other nodes.

The next two Peer-to-Peer centric dimensions are coupled together. The **routing strategies** of a PDBS are connected to the **overlaying network topology**. The network topology for unstructured PDBS is not fixed but just a result of the connections between the peers, similar to DDBS. Besides, structured networks like super-peers are built differently. The super-peers which contain more information than the usual peers are connected to each other in a predefined way, for example a ring or a hypercube. There is also the possibility that the whole PDBS has an overlaying network topology but not just some dedicated super-peers. Thus, if there is no predefined network topology the routing strategy of a PDBS can just be flooding, meaning from peer to peer as explained and shown in Figure 3.16. If a structured network is concerned, not just information about one neighbor is available but there is information about more neighbors and therefore some kind of greedy routing can be used, which makes it to look for the right peers more efficient.

The **scalability** is supposed to be a main advantage of peer-to-peer databases. If unstructured networks are involved, PDBS are poorly scalable as the routing of messages is flooding and so the network could become overallocated quickly. By using a super-peer network, the problem becomes smaller because the messages are just flooding the network of super-peers. If a fully structured network is overlaying the PDBS, the queries are routed to selected peers and unnecessary networking cost can be avoided. Thus, structured PDBS are much more scalable than unstructured ones. Another Peer-to-Peer specific characteristic is **anonymity**. Since requests are routed through many peers, the participants can stay hidden.

In conclusion, the peer-to-peer database system is a rather new development. It has some advantages as it is well scalable in terms of number of nodes and distribution. Furthermore there is the advantage that nodes can join and leave the network freely and do not have any administrative tasks. There is no special infrastructure required too, since a global schema does not exist. Moreover, since the data is accessed directly, freshness is guaranteed in

contrast to centralized databases. But on the other hand, PDBS still have some problems. Especially unstructured PDBS, which are not very famous anymore, have problems with scalability and cannot provide a perfect recall. But PDBS have some problems in general as well, namely that although data independence is desirable, it is hardly achieved since they do not have strategies for replication and fragmentation. A very big disadvantage of Peer-to-Peer systems is that there is still no support for transactions. Hence, Peer-to-Peer database systems are still a young issue and many things are still open.

Peer Data Exchange and queries and Updates in a Peer to Peer Database System are described in [42] and [41]. They define a model-theoretic semantics and coordination formulas between nodes for a Peer to Peer database system to avoid inconsistency.

4 Parallel Database Operations

This chapter gives an overview of the most important parallel database operations in an multiprocessor architecture: sort, join and aggregate. The chapter starts with a definition of a cost model and ends with a short introduction in query optimization in parallel and distributed database systems. The explanation of the database operations in conjunction with the cost model is the basis of our analysis for the performance behaviors of the algorithms in a homogeneous and heterogeneous environment. The introduced cost model, with a few adoptions, is also used in the heterogeneous environment and is required to compare the performance to the homogeneous environment.

4.1 Parallel Database Operations in Multiprocessor Architectures

We focus on the most common and well studied parallel relational database algorithms for sorting, joining and aggregates and analyze their performance. These algorithms are described in Bitton et.al. [45]. Bitton uses a generalized parallel multiprocessor organization described in section 3.8. This architecture is very similar to **DIRECT** [46]. Based on the dataflow machine **DIRECT** an earlier study of parallel join algorithms was done by Boral and DeWitt [47].

4.1.1 Cost Model

For comparing the algorithms we use the same definitions of the analysis parameter as described in [18]. We define n as the number of pages with k tuples, and p is the number of processors.

Communication Cost

A processor must request a page, for this purpose a message is necessary, the cost for such an "I/O-related" message is C_{msg}

I/O Cost parameters

- H ... certain hit ratio for the cache
- H' ... fraction amount of time a free page frame will be available in the cache during a writing operation
- R_c ... cost of a cache to processor transfer
- R_m ... cost of a mass-storage transfer

The average cost of a read by a processor is

$$C_r = HR_c + (1 - H)(R_c + R_m) + 2C_{msg}$$

The average cost of writing a page is

$$C_w = H'R_c + (1 - H')(R_c + R_m) + 2C_{msg}$$

Scan Cost

The sequential scan cost within a page are

$$C_{sc} = kC$$

where k is the number of tuples in a page, and C are cost of a simple operation (scan, compare, add).

Merge Cost

All operations require internally sorted pages. Thus the number of comparisons required to perform the merge of two sorted pages of length k is $2k$.

- V is the cost to move a tuple inside a page

Thus the cost of merging a page are

$$C_m = 2k(C + V)$$

Page Reorganization

To keep the tuples in sorted order in a page, we assume half of the tuples in the page will be affected to reorganize after an update or modify operation. The page reorganization cost are

$$C_o = (1/2)k(C + V)$$

and the cost to sort a page internally¹

$$C_{so} = (k \log(k))(C + V)$$

To group some of the above parameters we define analogously to Bitton et.al. [18] we group some parameters to a so called "**2-page operation**" C_p^2 :

$$C_p^2 = 2C_r + C_m + 2C_w$$

Disk, Network and Cache

The bandwidth of the network in a Grid varies from 100 *kbit/s* up to 10000 *kbit/s* for uploading data, and about 200 *kbit/s* up to 10000 *kbit/s* for downloading. Their values have been derived from measuring more than one hundred actual internet service-providers (see [48]). The disk bandwidth in disks available today range from about 100 *Megabit/second* up to 1200 *Megabit/second*. Different spindle speeds and seek-times are available. Table 4.1 shows an overview of today's common disks available and their parameters.

As we use the C_p^2 operation in the Static Simplified Grid, a "disk cache" is not necessary, because a C_p^2 operation is only performed between a local node and a remote node and not from a local node to itself. Thus, we can refine the definition of the average cost of a read by a node

$$C_r = HR_c + (1 - H)(R_c + R_m) + 2C_{msg}$$

$$\text{refined to } C_r = C_{ard} + C_{arn} + 2C_{msg}$$

The average cost of writing a page in a Grid organization is therefore

$$C_w = H'R_c + (1 - H')(R_c + R_m) + 2C_{msg}$$

¹We assume in this thesis all $\log()$ functions are to the basis of 2

Tabelle 4.1: *Common Disk Parameters*

spindle-speed[rpm]	seek-time[ms]	interface bandwidth[Mbit/s]
15000	3.6	1129
15000	3.3	1129
10000	4.5	1075
7200	8.5	998
7200	8.8	757
7200	14	998
7200	14	966
7200	10	629
5400	11	510
5400	12	493
4200	15	289
4200	13	288
3600	12	125

$$\text{refined to } C_w = C_{awd} + C_{awn} + 2C_{msg}$$

This also leads to a new definition of the architectural underlying model which is similar to the Generalized Multiprocessor Organization as the basis for the "2-page operation" C_p^2 analysis of the algorithms. This revised architecture is depicted in Figure 4.1.

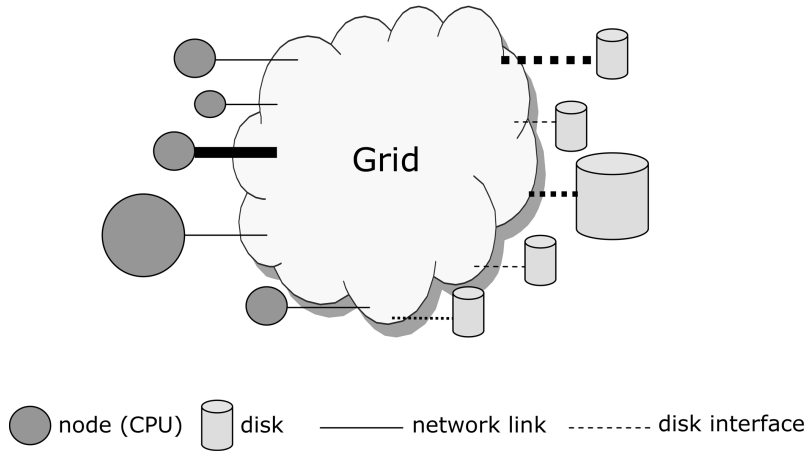


Abbildung 4.1: *Revised Static Simplified Grid Organization*

If we use the values of network bandwidth and disk bandwidth to calculate the cost² with a page size ps of $16Kbyte$ we get the values given in Table 4.2. Based on these findings we can neglect to differentiate between disk read and write cost, because the values are nearly identical.

4.1.2 The Influence of Network Cost

The percentage of message- (Msg), processing power (CPU), disk- (disk) and network-cost (net) as part of a C_p^2 operation in the Revised Static Simplified Grid Organization is shown in Figure 4.2. Only the network speed (sum of C_{arn} and C_{awn}) is varied, the costs of the message transfer C_{msg} , processing power C_m , sum of disk cost C_{ard} , C_{awd} are fixed. The message cost have been left fixed because a message contains only a few bytes. The

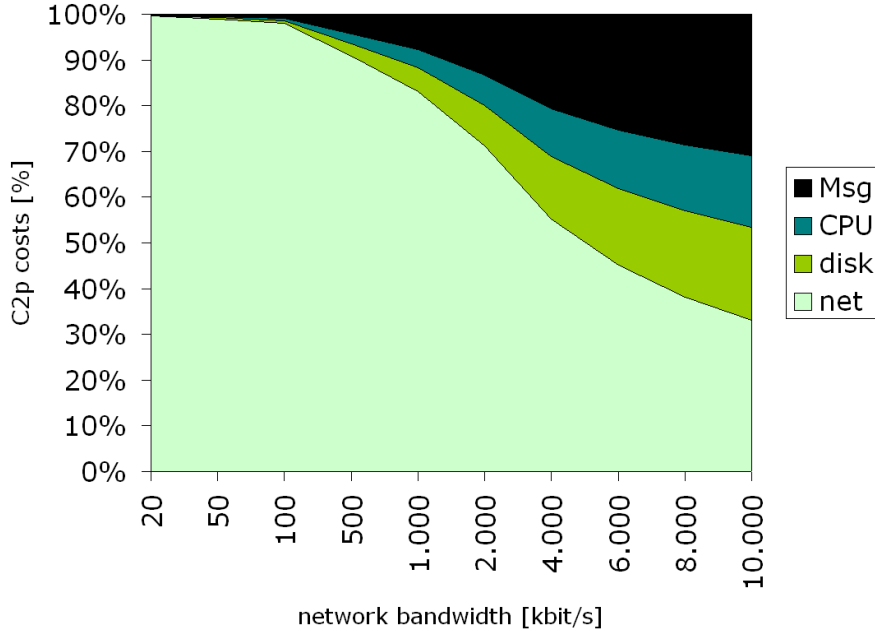
²Time for reading a page from disk is seek-time+rotational delay+transfer time

Tabelle 4.2: *Average Cost of Network and Disk*

name	value[sec]	type of costs
C_{arn}	0.0542	average costs for read network
C_{ard}	0.0197	average costs read disk
C_{awn}	0.2418	average costs write network
C_{awd}	0.0197	average costs write disk

transfer time of such a short message is nearly independent of the network speed, as a result of protocol overhead. The disk cost for transferring a page of 16 *Kbyte* varying only marginally (see table 4.1) and therefore we can neglect them.

The influence of the network speed on the cost of a C_p^2 operation can be summarized by: "the higher the network speed, the lower the impact on the cost of a C_p^2 operation".

**Abbildung 4.2:** *Percentage of C_p^2 cost in a Revised Static Simplified Grid Organization*

4.1.3 Parallel Sorting Algorithms

Sorting is an important operation in a database system. It is frequently used in query execution plans generated by database query optimizers. Therefore its performance influences dramatically the overall performance of a database system [15, 16].

Generally sorting algorithms can be divided into internal and external sorting. Internal sorting is done by using the main memory of a processor while external sorting uses also the disk [17]. If the data set for sorting is too large to fit into the main memory, external sorting is necessary. The common case in database systems is using external sorting.

In the following we concentrate our discussion on two well known parallel sorting algorithms, the parallel binary merge sort and the block bitonic sort algorithm.

4.1.3.1 Parallel Binary Merge Sort

Binary Merge Sort uses several phases to sort nk tuples, where n is the number of pages containing the data set and k denotes the number of tuples per page. We assume that there are much more pages n than processing nodes p (i.e. $n \gg p$) and that the size of the data set is much larger than the available main memory of the nodes.

We assume the very general case that the pages are not distributed equally among the mass storage media of the available nodes and that the tuples are not presorted according to the sorting criteria in the pages. Therefore the algorithm starts with a prepare phase (see Figure 4.3), which distributes the pages equally across all p nodes, sorts the tuples inside every page according to the sort criterion and writes the pages back to disk. After the prepare phase n/p (respectively $n/p - 1$) pages are assigned to each node and the tuples of each page are sorted. The algorithm continues with the suboptimal phase by merging pairs of longer and longer runs³. In every step the length of the runs is twice as large as in the preceding run. At the beginning each processor reads two pages, merges them into a run of 2 pages and writes it back to the disk. This is repeated, until all pages are read and merged into 2-pages-runs. If the number of runs exceeds $2p$, the suboptimal phase continues with merging two 2-page-runs to a sorted 4-page-run. This continues until all 2-page-runs are merged. The phase ends, when the number of runs is $2p$. At the end of the suboptimal phase 2 sorted files of length $n/2p$ exist on each node . In the suboptimal phase the nodes work independently in parallel. Every node accesses its own data only. During the following optimal phase (see Figure 4.4) each processor merges 2 runs of length $n/2p$ and pipelines the result (run of length n/p) to a target node. The number of target nodes is $p/2$. The identification of the target-node is calculated by

$$noder_{target} = \frac{p}{2} + noder_{source}$$

for even source-node-numbers, and

$$noder_{target} = \frac{p}{2} + noder_{source} + 1$$

for odd source-node-numbers.

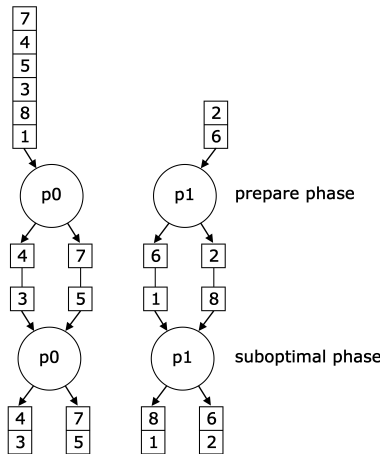


Abbildung 4.3: Prepare and Suboptimal Phase

In the postoptimal (last) phase the remaining $p/2$ runs are merged into the final run of length n . At the beginning of the postoptimal phase, we have $p/2$ runs. During this phase one of the p nodes is no longer used. Each of the other nodes is used only once during

³a **run** is an ordered (respective to the tuples contained) sequence of pages.

this phase. Two forms of parallelism are used. First, all nodes of one step work in parallel. Second, the steps of the postoptimal phase overlap in a pipelined fashion. The execution time between two steps consists of merging the first pages, building the first output-page and sending it to the target-node. During the postoptimal phase every node is used only in one step, that means that every node is idle for a certain time.

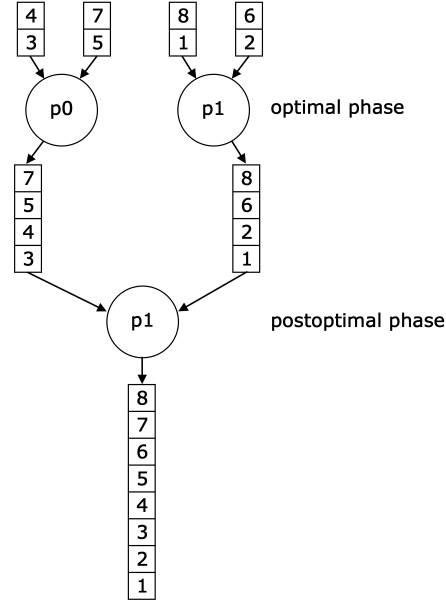


Abbildung 4.4: *Optimal phase*

Thus the algorithm costs are

$$\underbrace{\frac{n}{2p} \log\left(\frac{n}{2p}\right)}_{\text{suboptimal}} + \underbrace{\frac{n}{2p}}_{\text{optimal}} + \underbrace{\log p - 1 + \frac{n}{2}}_{\text{postoptimal}} \quad (4.1)$$

which can be expressed as

$$\frac{n \log n}{2p} + \frac{n}{2} - \left(\frac{n}{2p} - 1\right)(\log p) - 1 \quad (4.2)$$

4.1.3.2 Block Bitonic Sort

For better understanding of the Block Bitonic sort we give a short introduction to sorting networks.

Sorting Networks

The basis of sorting networks are the comparison networks, which consist of wires and comparators. A comparator gets two inputs and gives back two outputs. It performs the functions $x' = \min(x, y)$ and $y' = \max(x, y)$ while x and y are the inputs. In short, the comparator gets two inputs and returns on the top the minimum and on the bottom the maximum. Figures 4.5(a) and 4.5(b) represent exactly such a comparator. The only difference between the two Figures is that the comparator in Figure 4.5(b) is drawn as a single vertical line.

A **comparison network** consists of multiple comparators that are connected by wires. An example for a comparison network is shown in Figure 4.6. The wires a_1, a_2, \dots, a_n are the so-called input wires while b_1, b_2, \dots, b_n represent the output wires. The values of the wires

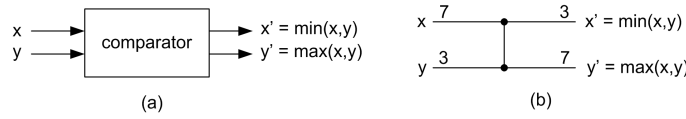


Abbildung 4.5: (a) operator with two inputs and two outputs. (b) same comparator, just drawn as vertical line

are represented in the input sequence $\langle a_1, a_2, \dots, a_n \rangle$, respectively in the output sequence $\langle b_1, b_2, \dots, b_n \rangle$ which means that the notation for the wires as well as for their values are the same.

Referring to Figure 4.6, it can be seen that every comparator is either connected to an input wire or to an output wire of another comparator. On the other hand the output of a comparator can be either an output wire of the network or an input wire for another comparator. For instance the comparator *A* from Figure 4.6 is connected to two input wires while the comparator *C* is connected to output wires from *A* and *B*.

For comparison networks it is of particular importance that the path of connections does not cycle back on itself. Thus, the path must not go through the same comparator twice. If the interconnection graph can be read from left to right like in Figure 4.6, the path is acyclic and the requirement is fulfilled.

Assuming that each comparator takes the same amount of time, that is to say one unit, the "running time" of the network can be calculated. More specifically the depth of the comparison network can be defined. A comparator can just produce its output value when it has received both input values. Given that the input wires appear at time 0, the input values for comparators *A* and *B* would be provided at time 0 and their output values would be produced at time 1. At the same time the depth of those comparators would be 1 as can be seen in Figure 4.6(b). Therefore, at time 1 comparators *C* and *D* have their inputs and can produce their outputs at time 2. The top output of *C* and the bottom output of *D* are already the output wires of the network carrying the output results as can be seen in Figure 4.6(c) and (d). The comparator *E* gets its input from *C* and *D* at time 2 and produces results at time 3 which means that the whole comparison network has a depth of 3.

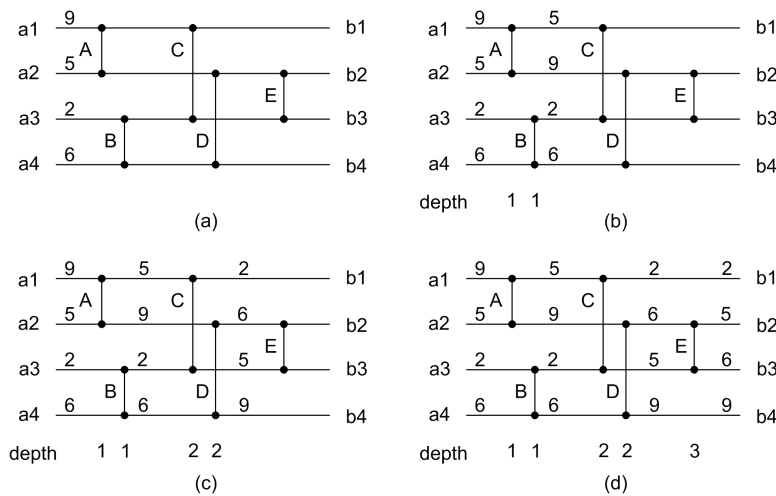


Abbildung 4.6: (a) The comparison network, which is in fact a sorting network, at time 0. (b) comparison network at time 1. (c) comparison network at time 2. (d) comparison network at time 3

As soon as the comparison network produces a monotonically increasing output sequence

($b_1 \leq b_2 \leq \dots \leq b_n$) for every input sequence, it is called a **sorting network**. The example from Figure 4.6 shows a sorting network.

A **bitonic sorting network** is a sorting network which can sort any bitonic sequence. A bitonic sequence is defined as a sequence that is monotonically increasing and then monotonically decreasing. Alternatively a sequence is also called bitonic if it can be made monotonically increasing and then monotonically decreasing by circular shifting. Possible bitonic sequences would be $\langle 3, 4, 7, 5, 2, 1 \rangle$ or $\langle 4, 7, 5, 2, 1, 3 \rangle$. The first sequence is increasing until the maximum of 7 and then decreasing. The second sequence can be cyclically shifted to the first sequence and is so bitonic as well. If a sequence is just monotonically increasing or monotonically decreasing, it is also bitonic.

The bitonic sorting network will be explained using the set $\{0, 1\}$. Due to the **Zero-one principle** [49], a comparison network is able to sort all sequences of arbitrary numbers correctly if it is able to sort all 2^n possible sequences of 0's and 1's correctly. Bitonic zero-one sequences have always one of two different forms. The two possibilities are either $0^i 1^j 0^k$ or $1^i 0^j 1^k$ for $i, j, k \geq 0$.

A bitonic sorter consists of several comparison networks of depth 1, which are called **half-cleaners**. In a half-cleaner input line i is compared with line $i + n/2$ for $i = 1, 2, \dots, n/2$. Figure 4.7 shows two examples where bitonic sequences of 0's and 1's are used as input to a half-cleaner. Those specific comparison networks produce two bitonic output sequences where the smaller outputs are in the top and larger values in the bottom.

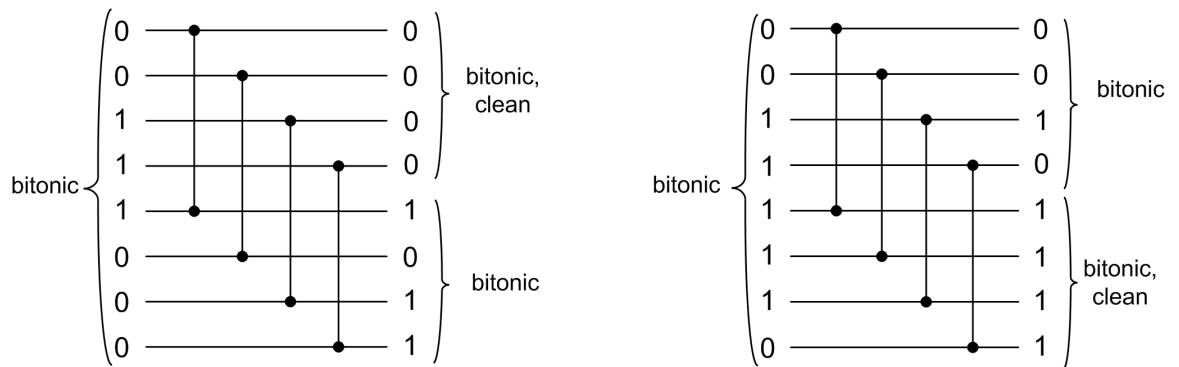


Abbildung 4.7: two comparison networks with depth 1, called half-cleaner.

Actually, there are some more characteristics of a half-cleaner that can be identified and proved by looking at the different possible cases depending on where the midpoint of a bitonic zero-one sequence falls. Assuming that a bitonic sequence with the form $000..011..100..0$ is used as input, there are three different cases where the midpoint can fall. It can occur either in the first "0" subsequence, in the "1" subsequence or in the second "0" subsequence. If it occurs in the "1" subsequence, a further split into two cases is possible. Therefore, there are eventually four different cases which are shown in Figure 4.8. Anyway, if the form of the input sequences would be $111..100..011..1$, the situation would be symmetric.

As can be seen, all of the four possible cases in Figure 4.8 have similarities due to general characteristics of a half-cleaner. That signifies that no matter how big i, j, k of a bitonic sequence $0^i 1^j 0^k$ or $1^i 0^j 1^k$ are, as long as the condition $i, j, k \geq 0$ is satisfied, the following characteristics of a half-cleaner are valid:

- The two output halves of a half-cleaner are always bitonic sequences themselves, both the top as well as the bottom half.
- Every output element of the top half is at least as small as every output value of the bottom half.

- One of the produced output sequences, either the top or the bottom half are clean. That means that it consists either of just 0's or just 1's.

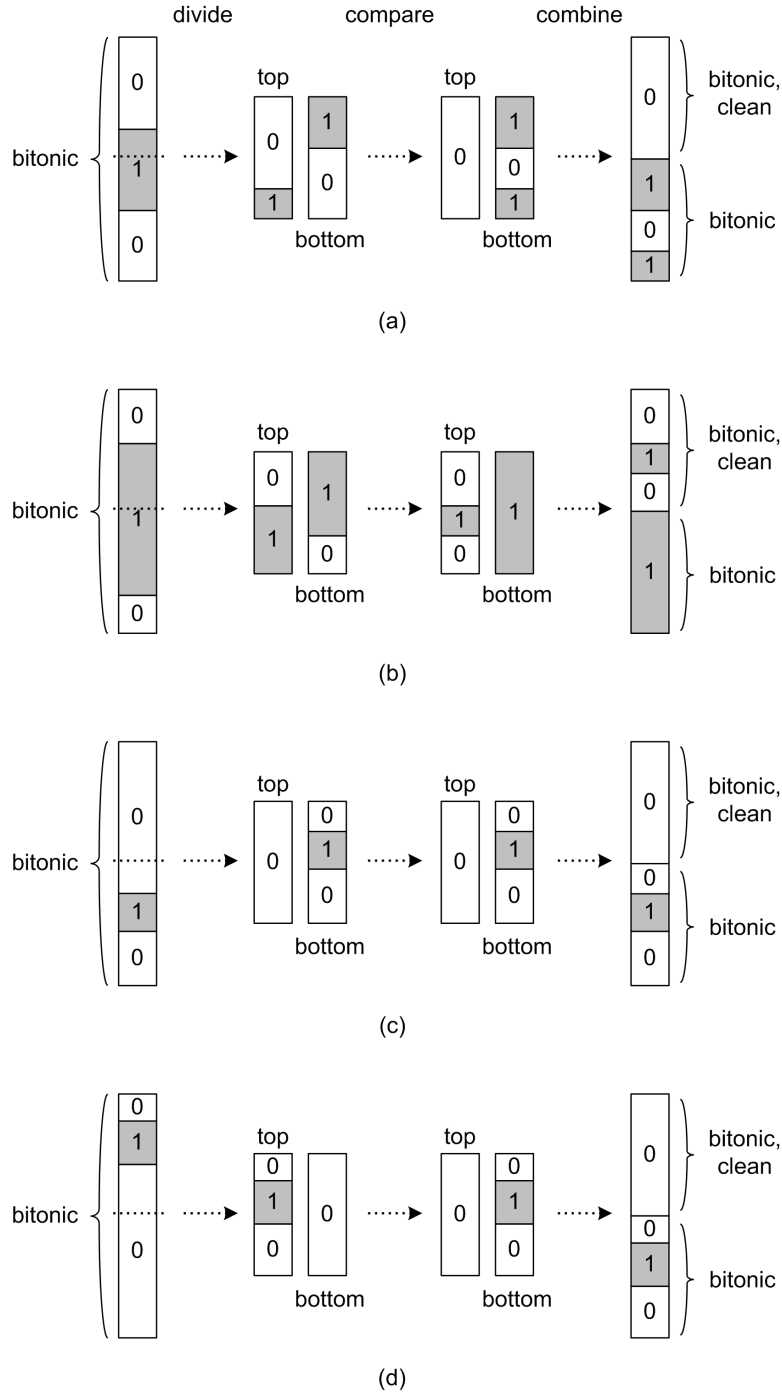


Abbildung 4.8: There are four different cases where the midpoint of a bitonic zero-one sequence can fall. (a)-(b) the division occurs in the middle subsequence of 1's. (c)-(d) the division occurs in one of the "0" subsequences. The outputs consist always of two bitonic parts and one part is always clean. Furthermore every output element of the top half is always at least as small as every output element of the bottom half.

As previously mentioned, a BITONIC-SORTER consists of several half-cleaners. Those half-cleaners are recursively combined as shown in Figure 4.9. In general, a BITONIC-SORTER[n] starts with a half-cleaner which is followed by two copies of BITONIC-

$\text{SORTER}[n/2]$ that operate in parallel. The number of half-cleaners needed in total is dependent on the number n of input-wires. If $n = 8$ like in Figure 4.9, every $\text{BITONIC-SORTER}[n/2]$ consists of three half-cleaners and therefore seven half-cleaners are needed in total. All half cleaners in Figure 4.9(b) are shaded. The depth of such a network is given by:

$$D(n) = \begin{cases} 0 & \text{if } n = 1 \\ D(n/2) + 1 & \text{if } n = 2^k \text{ and } k \geq 1 \end{cases}$$

The solution of this function is $D(n) = \lg n$. [49]

So any bitonic zero-one sequence can be sorted by the BITONIC-SORTER with the depth of $\lg n$. Furthermore it can be deduced from the zero-one principle that any bitonic sequence of arbitrary numbers can be sorted by this sorting network.

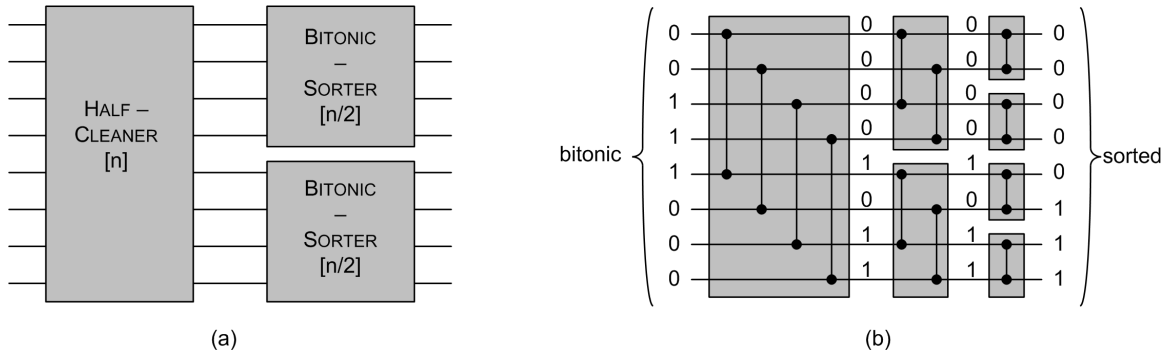


Abbildung 4.9: The bitonic sorting network $[n]$ for $n = 8$ is shown. (a) The recursive construction of the sorting network. (b) The network showing the details and the progressively smaller half-cleaners that sort the input eventually.

Improvements of the bitonic sort has been introduced (like a **recirculating bitonic sorting networks** for minimizing communication) in [50] which reduces the cost complexity to $O(N \log N)$ from $O(N \log^2 N)$, originally. Also the improvement of the preprocessing stage in a parallel block bitonic sort was investigated by Menon [51].

Batcher's bitonic sort

Batcher's bitonic sort algorithm sorts n numbers with $n/2$ comparator modules in $\frac{1}{2} \log n (\log n + 1)$ steps [52]. Each step consists of a comparison-exchange at every comparator module and a transfer to the target-comparator module. The comparator modules are connected by a perfect shuffle arrangement described in [53]. (Figure 4.10). The perfect shuffle uses three types of comparator modules. (Figure 4.11). The comparator module is represented by a node. This comparator module merges two pages and distributes the lower page and the higher page to two target-nodes. The way to define the target-nodes is done by a mask-information. This mask-information is the schema building the perfect shuffle interconnection. This configuration is called block parallel algorithm and can sort n pages with $n/2$ comparator modules in $\frac{1}{2} \log n (\log n + 1) C_p^2$ time units.

Note that the block bitonic algorithm can process at most $2p$ blocks (runs) with p processors. If the number of pages exceeds $2p$ a preprocessing stage is necessary to produce $2p$ sorted blocks with the size of $n/2p$ pages. This preprocessing stage can be done by performing the suboptimal phase of the binary merge sort and the costs are $(n/2p) \log(n/2p)$.

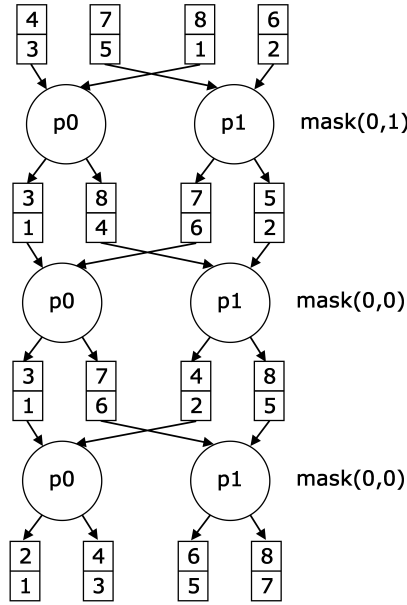


Abbildung 4.10: Bitonic Sort

After this preprocessing stage a external block bitonic sort is applied to $2p$ blocks with size $n/2p$ and this cost are:

$$\frac{n}{2p} \frac{\log 2p}{2} (\log 2p + 1) C_p^2 \quad (4.3)$$

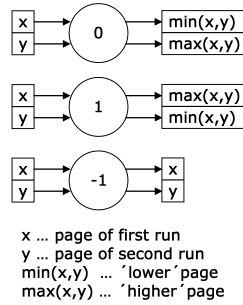


Abbildung 4.11: Perfect Shuffle

The implementation algorithm uses three basic operations, which are described below.

1. **circulate(S)**

From each of the $2p$ files the pages are routed to a node through the perfect shuffle interconnection. Each node p_i creates its MASK(i) each node merges its pages and distributes them according to MASK(i) = 0, 1, -1

2. **shuffle(mask)**

This operation shuffles the MASK-array from the second 2^{p-1} nodes to their target-nodes such that the target-nodes will use the same type of comparator.

3. **mask(j)**

It computes the entries of the MASK-array. Given an integer j , an array of length 2^p is created as follows.

$$\begin{aligned}
\text{MASK} &= -1, -1, -1, \dots, -1 & \text{if } j = -1 \\
\text{MASK} &= 0, 1, 0, 1, \dots, 0, 1 & \text{if } j \leq p - 1 \\
\text{MASK} &= 0, 0, 0, \dots, 0, 0 & \text{if } j = p
\end{aligned}$$

The algorithm for the perfect shuffle is defined by Algorithm 1.

Algorithm 1: Perfect shuffle algorithm

```

1 foreach  $i$  from 1 to  $p$  do
2   Mask(-1);
3   foreach  $j$  from 1 to  $p-i$  do
4     Circulate(S);
5   Mask(i);
6   foreach  $j$  from  $p-i+1$  to  $p$  do
7     Circulate(S);
8     Shuffle(MASK);

```

The total cost for the block bitonic sort with the preprocessing stage are:

$$\frac{n}{2p} \left(\log n + \frac{\log^2 2p - \log 2p}{2} \right) C_p^2 \quad (4.4)$$

4.1.4 Parallel Join Operations

In this section we introduce three join algorithms for relational database systems, a parallel "nested-loop", parallel "sort-merge" and a parallel "Hashing" algorithm. Like the sort algorithms the presented join algorithms are commonly used in database systems. We have carefully investigated join algorithms in parallel environments like [54, 55, 56, 57, 58, 59]

4.1.4.1 Parallel Nested Loop Join

The inner (smaller) relation T , and the outer relation R (larger one) are joined together. The algorithm can be divided into two steps:

1. Initiate

Each of the processors reads a different page of the outer relation R

2. Broadcast and join

All pages of the inner relation T are sequentially broadcast to the processors. After receiving the broadcast page, each processor joins the page with its page from R .

n and m are the sizes (number of pages) of the relations R and R' , and we suppose $n \geq m$. To perform the join of R and R' we assign p processors. If $p = n$, the execution time is:

$$\begin{aligned}
T_{\text{nested-loop}} &= T(\text{read a page of } R) \\
&\quad + mT(\text{broadcast a page of } R') \\
&\quad + mT(\text{join 2 pages})
\end{aligned} \quad (4.5)$$

S is the join selectivity factor and indicates the average number of pages produced by the join of a single page of R with a single page of R' . Joining two pages is performed by merging the pages, sorting the output page on the join attribute and write the sorted page to disk.

$$S = \frac{\text{size}(R \text{ join } T)}{mn} \quad (4.6)$$

If the number of processors p smaller than the number of pages n , step 1) and 2) must be repeated $\frac{n}{p}$ times. Therefore the cost for the Parallel Nested-Loop Join is

$$T_{nested-loop} = \frac{n}{p} \left[C_r + m[C_r + C_m + S(C_{so} + C_w)] \right] \quad (4.7)$$

4.1.4.2 Parallel Sort Merge Join

The algorithm processes in two steps. The first step of the algorithm is to sort the two relations on the join attribute (we assume, that the two relations are not already sorted). After sorting, the second step is performed, where the both sorted relations are joined together and the result relation is being produced. If we use the block bitonic sort for the first step, the cost of the Sort Merge Join are

$$T = \left[\frac{n}{2p} \log n + \frac{m}{2p} \log m + (\log^2 2p - \log 2p) \frac{n+m}{4p} \right] C_p^2 + (n+m)C_r + \max(nm)C_m + mnS(C_{so} + C_w). \quad (4.8)$$

Using the Parallel Binary Merge Sort the cost in term of C_p^2 cost are

$$T = \left[\frac{n \log n}{2p} + \frac{n}{2} - \left(\frac{n}{2p} - 1 \right) (\log p) - 1 \right] C_p^2 + (n+m)C_r + \max(nm)C_m + mnS(C_{so} + C_w). \quad (4.9)$$

4.1.4.3 Parallel Hashing Join

The algorithm is based on the analysis described in [60]. The Architectural model in [60] is very close to that described in Bitton et al. The Hashing Join in [60] uses Bit-Arrays. The method is to hash the join attribute and to use the result as an address into the Boolean array. The marked bit in the array denotes that matching tuples exist. The value of the Boolean arrays is to eliminate most of the data not needed in the result. To keep the algorithm simple only the equijoin is analyzed.

The algorithm is divided in two stages. In the first stage, a cache processor is chosen and the smaller relation is read into the cache memory and hashed on the join attribute. The result are tuples written in buckets of a hashed file. The hashed file is composed of buckets having a variable number of linked pages. For each bucket a page frame is maintained in cache memory. This avoids the need to manage an overflow area. Simultaneously, for each join attribute value ν , a boolean array $B(h(\nu))$ is marked (set to 1), where h is a hashing function applied to the join attribute. The first stage is completed when the entire relation has been hashed.

In the second stage, the Boolean array is broadcast to p processors. The larger relation is sequentially distributed among p processors. Each processor uses two buffers as input pages, one buffer as the output page and one buffer to store the Boolean array. Thus, each processor receives one page of the larger relation.

Algorithm 2: Hash Stage-2 Algorithm

```

1 foreach page in a Bucket do
2   foreach tuple in the page do
3     if join attribute value  $\nu'$  satisfies  $B(h(\nu')) = 1$  then
4       One bucket of the hashed file is accessed by specifying the key to find the matching
       tuple(s);
5     The tuples of each page are then compared with  $\nu'$  to complete the join ;

```

To avoid collisions in hashing more than one hashing function is used. If ν_1 and ν_2 are different join attribute values, we can have $h_{\nu_1} = h_{\nu_2}$. Since the Boolean array is accessed by hashing, collisions can lead to useless access to the hashed file during the second stage. In order to reduce collisions, several hashing functions h_1, h_2, \dots, h_q can be used, each associated with a Boolean array B_1, B_2, \dots, B_q . Then, for each value ν , all of the corresponding bits in each B_i must be set (i.e., $B_1(h_1(\nu)) = 1, B_2(h_2(\nu)) = 1, \dots, B_q(h_q(\nu)) = 1$). Increasing q results in a probability of collisions near zero.

For better comparison we translated the original names of the variables used in [60] to that defined in [18].

The execution time of the algorithm comprises time $T1$ for hashing the smaller relation by the cache processor, time $T2$ for distributing the larger relation among p processors, time $T3$ for accessing the hashed file, and, finally, time $T4$ for writing the result. The time for broadcasting the Boolean arrays is negligible and, thus, is ignored. c page frames are available in the cache for the join operation. Thus the creation of the hashed file consists of creating m buckets, if $c > m$, or c buckets, otherwise. In the first case, the hashed file can be maintained in cache memory during the entire execution of the join operation. In the latter case, the pages of the same bucket would be linked and written to disk, and retrieved using a table of physical addresses. The time for reading a page for R , taking into account the ratio H , is

$$t1 = (1 - H)R_c.$$

The time for hashing a page of k tuples is

$$t2 = k(C + V).$$

The time for writing the hashed file is the time for writing $(m - c)$ pages, since c pages are reserved in cache memory during the join execution. Furthermore, page frames may be available in the cache with the probability H' . Thus the time for writing $(m - c)$ pages from cache to disk is

$$t3 = (m - c)(1 - H')R_c.$$

Then, the execution time for hashing a relation of m pages is

$$T1 = m(t1 + t2) + t3$$

which is

$$T1 = m[(1 - H)R_c + k(C + V)] + (m - c)(1 - H')R_c$$

The cost in term of time of the second stage consist of following:

1. Reading the relation S by p processors in parallel,
2. Accessing the hashed file, and
3. Writing the result relation.

Each processor reads n/p pages of the relation S , and, for each of the t tuples of a page, accesses the Boolean array. The cost are therefore

$$T2 = (C_r + k \cdot C) \frac{n}{p}.$$

For each matching tuple of S an access to the hashed file is needed. The number of matching tuples is defined by the semijoin selectivity factor SS , and each bucket of the hashed file contains $\frac{m}{c}$ pages. Thus, for each page of S , the number of pages read from the hashed file is $k \cdot SS \frac{m}{c} C_r$; and this occurs $\frac{n}{p}$ times for the entire relation S .

$$T3 = \frac{n}{p} \frac{m}{c} C_r \cdot k \cdot SS$$

The time for writing the result relation of size $m \cdot n \cdot JS$ in parallel by p processors is

$$T4 = m \cdot n \cdot JS \frac{C_w}{p}$$

The total time T of the Hashing Join is the sum of $T1, T2, T3$ and $T4$

$$T = m[(1 - H)R_c + k(C + V)] + (m - c)(1 - H')R_c + (C_r + k \cdot C) \frac{n}{p} + \frac{n}{p} \frac{m}{c} C_r \cdot k \cdot SS + m \cdot n \cdot JS \frac{C_w}{p} \quad (4.10)$$

Another approach instead of using hashing for partitioning data is to use range partitioning instead of hash partitioning. One of these approaches is described in [61] as hybrid-range declustering strategy. In this strategy, the user specifies a range of key values for each processor and the range of values of the partitioning attribute for each processor is stored in a so called **range table**. Ghandeharizadeh and Dewitt showed that in mixed workloads the hybrid-range declustering strategy is the better one.

4.1.5 Parallel Aggregate Operations

Two different kinds of aggregates can be distinguished, "scalar" aggregates and aggregate "functions". Scalar aggregates are aggregations (**average**, **max**, etc.) over an entire relation. In contrary, aggregate functions first divide a relation into disjoint partitions (based on some attribute values) and then compute scalar aggregates on the individual partitions. Scalar aggregates compute a single result, while aggregate functions produce a set of results as a temporary relation. The operations for aggregation are depicted in table 4.3. In Bitton et al. [18] QUEL as query language is used. For the purpose of this analysis we stick to QUEL as example language to make the results directly comparable. All findings are easily applicable to other languages, as SQL or even the relational algebra. The syntax of the scalar aggregate in QUEL has the form

$$agg_op(agg_att \textbf{ where } qual)$$

and for aggregate functions

$$agg_op(agg_att \textbf{ by_list where } src_qual \textbf{ where } by_qual)$$

where agg_op denotes an aggregate function as given in table 4.3 and agg_att is the attribute on which the aggregate is been calculated. by_list is a list of attributes and src_qual is a qualification for the source relation. Qualifications may be added ("**where qual**") to compute an aggregate over a subset of tuples in a relation. That means the by_qual eliminates unwanted tuple sets of result partitions. Partitioning a relation on more than one attribute is also possible (e.g., partitioning employees by department and task within department). Also note that the result of an aggregate function may depend on qualifications outside the aggregate (by_qual). In Bitton et al. [18] "simple" qualifications are distinguished from "complex" qualifications. Simple qualifications can be processed simultaneously to the computing of the aggregate. On the other hand complex qualifications require interrelated operations. So the relation must be preprocessed before computing the aggregates. We analyze aggregates with "simple" qualification only. For computing a scalar aggregate two variables are needed: a count variable, and the aggregate variable. The count variable holds the number of tuples contributing to the aggregate value and is used in averaging and initialization. While processing aggregate functions, a third variable is required to identify the partition. We focus on aggregate functions because scalar aggregates are a special case of them.

An aggregation performed over a set of different values of an attribute is called a unique aggregation. QUEL supports three unique aggregates: **countu**, **sumu**, and **avgu**. Unique

Tabelle 4.3: *Aggregate Functions*

Function	Value returned
count()	Number of entries in column.
countu()	Number of unique entries in column.
sum()	Sum of values in column.
sumu()	Sum of unique values in column.
avg()	Average of values in column.
avgu()	Average of unique values in column.
max()	Maximum value in column.
min()	Minimum value in column.
any()	Returns 1 if any rows satisfy the condition expressed by the argument

versions of **any**, **max** and **min** are not necessary.

Tabelle 4.4: *Employee Relation*

name	department	task	salary	mgr
Smith	Toys	Clerk	300	Johnson
Miller	Shoes	Buyer	650	Bergman
Jones	Books	Account	550	Harris
Brown	Shoes	Clerk	400	Conners

Now we describe two algorithms for aggregate functions which distinguish two types of qualifications. This is explained by the following example. Given the relation in Table 4.4 we express the query:

count (*emp.name by emp.mgr where emp.salary > 500*)

This query requests, for each manager, a count of the number of employees earning more than 500. Even if a manager does not have any employees making more than 500, he should not be excluded from the list and his count should be set to 0. If we apply the qualification first and then compute the aggregate function on the result, we would miss those managers since all his employees were removed by the qualification. The result is therefore:

Johnson 0
Bergman 1
Harris 1
Conners 0

As another example, consider

count (*emp.name by emp.mgr where emp.salary > 500*)
where emp.mgr <> "Johnson"

In this query we want to include the count for all managers other than Johnson. The result for this query is:

Bergman 1
Harris 1
Conners 0

Thus, we need to distinguish restrictions on the source tuples from restrictions on the set

of possible partitions. The source qualification ("*src_qual*") is the qualifications inside the aggregate. That means to select a subset of the source relation may have the undesirable side effect of removing necessary partitions (manager Johnson in our example). Qualifications outside the aggregate (the "*by_qual*") are used to eliminate unwanted partitions. That means the result of an aggregate function may depend on qualifications outside the aggregate ("*by_qual*").

On the other hand, scalar aggregates are not affected by the rest of the query, they are self-contained.

If a *src_qual* is specified in an aggregate function, any algorithm must start by determining the set of desired partitions so that all partitions which are removed by applying the *src_qual* (e.g. managers with zero counts) can be included in the result of the query. Determining the set of partitions requires one or two steps, depending on whether the query contains a *by_qual* or not. If the query has a specified *by_qual*, the source relation eliminates unwanted partitions. Then this result relation is projected on the *by_list* attributes to determine the names of the desired partitions. If the query has no *by_qual*, the original source relation is projected.

4.1.5.1 Subqueries with Parallel Merge (Aggregate Algorithm A)

Our first aggregate algorithm is performed in two stages:

1. **Stage 1**

Each of the p processors reads the pages of its source relation and builds one aggregate value for each partition. The result is a set of pages containing partial results.

2. **Stage 2**

The parallel merging of the pages produced in Stage 1 is performed.

Algorithm 3: Aggregate Algorithm A

```

1 if by_qual then
2   | Eliminate unwanted partitions by applying the by_qual predicate;
3 end
4 if src_qual then
5   | Step 1: Project the source relation or the relation produced by executing the by_qual
      | predicate on the attributes of the by_list. This step will produce a temporary result relation
      | with the result and count values for each partition initialized to 0;
6   | Step 2: Apply the source qualification src_qual and compute the aggregate values. If the
      | query has a simple src_qual, it may be processed at the same time the aggregate is
      | computed; otherwise, it is performed as a separate operation before the aggregate is
      | computed;
7   | Step 3: One Processor merges the temporary result relation with its run of  $t$  pages;
8 end
9 Perform the parallel merge;
```

If unique aggregates are used, a separate preprocessing step is required in which the source relation is sorted on the *by_list* in order to eliminate duplicates.

The execution time of the algorithm is derived from two stages. We assume a query with no qualification and an aggregate that is not unique.

$$T(\text{Algorithm A}) = T(\text{produce partial result pages}) + T(\text{parallel merge})$$

In Stage 1 each of the p processors read n/p source relation pages. The processors update the aggregate value for their specific partition and keep the tuples in the pages sorted. There are $x = \min(m, r)$ partitions, where m is the number of partitions and r is the

number of result tuples in the output page. To locate the correct partition a binary search is used, which requires $\log(x)$ comparisons. Then the aggregate value can be updated for the partition. Therefore the cost for processing the source relation pages is:

$$T(\text{processing source relation}) = \frac{n}{p} \{C_r + k[(\log x) + 1]C\} \quad (4.11)$$

The estimation of the number of output pages produced by one processor is $t = \lceil m/r \rceil$ if the partition is uniformly distributed in the relation and every processor "sees" all partitions. Therefore the cost for one processor to write the partial result pages is tC_w . The cost for keeping the pages in sorted order by using a binary search is as follows: Each time a new partition (new *by_list* value) is encountered, the processor must create a new tuple and adds it to the sorted page. The number of moves for each partition is on the average $x(x+1)/4$ where x is the number of partitions (half of the tuples must be moved). The cost to process each of the t pages produced by a processor is:

$$t \left[\frac{Vx}{4}(x+1) + C_w \right] \quad (4.12)$$

The Stage 2 is a parallel merge operation where two partial output runs are combined to a single output run. Each processor must create a sorted run of the t pages it has produced itself. A merge sort requires $(\frac{t}{2})\log(\frac{t}{2}) C_p^2$ operations. To combine the p runs of t pages into one run of t pages, a binary merge (see section 4.1.3) is used. This binary merge requires $\log(p)$ stages. Each of the processors reads two runs of t pages, merges them, and writes a run of length t . The costs of merging two output pages is $C_{m'} = 2r(C + V)$ (where r is the number of tuples in a output page). The cost for the parallel merge is therefore:

$$T(\text{parallel merge}) = (t + \log(p))(2C_r + C_{m'} + C_w) \quad (4.13)$$

Now we are ready to write the final formula for Aggregate Algorithm A:

$$\begin{aligned} T(\text{AlgoA}) = & T(\text{execute by_qual}) && \text{if by_qual} \\ & + T(\text{project on by_list}) && \text{if src_qual} \\ & + T(\text{execute src_qual}) && \text{if complex src_qual} \\ & + T(\text{project}) && \text{if unique aggregate} \end{aligned}$$

and the processing of partitions:

$$\begin{aligned} & + \frac{n}{p} \{C_r + k[(\log(x)) + 1]C\} && x = \min(r, m) \\ & + \frac{n}{p} q C_{sc} && \text{if simple src_qual} \\ & + t(x(x+1)(\frac{V}{4}) + C_w) && t = \lceil m/r \rceil \end{aligned}$$

and the parallel merge operation:

$$\begin{aligned} & + (\frac{t}{2})\log(\frac{t}{2}) C_p^2 \\ & + t(2C_r + C_{m'} + C_w) && \text{if src_qual} \\ & + (t + \log(p))(2C_r + C_{m'} + C_w) \end{aligned}$$

4.1.5.2 Project *by_list* and broadcast source relation (Aggregate Algorithm B)

A typical multiprocessor architecture supports broadcasting pages to multiple processors. This ability is used in this algorithm. First, the source relation is being projected on the *by_list* domains to determine the partitions. Now this list of partitions will be distributed to the p processors. Then the pages of the source relation will be broadcast to all processors and each processor computes the aggregate value for its set of partitions. If the memory space occupied by the list of partitions exceeds the combined buffer space of the processors, then the source relation will have to be broadcast more than once. As in

Aggregate Algorithm A, if the query has a simple *src_qual* predicate, it may be processed concurrently while the aggregate value is being computed. If a unique aggregate is used, the source relation must be sorted on its *by_list* and each processor must compare the tuples to eliminate duplicates.

The cost of the algorithm is (as in Algorithm A, where we assumed a query with no qualification and an aggregate that is not unique):

$$T(\text{Algorithm B}) = T(\text{project by_list}) \\ + T(\text{process partitions})$$

A processor processes every page of the n source relation pages. Each tuple must be placed in the correct partition. There are m/p or r partitions, depending on the number of passes over the source relation. We assume that the partitions are sorted to allow the usage of a binary search. The number of broadcasts of the source relation is $b = \lceil (m/r)/p \rceil$. Then the cost to process the partitions is

$$T(\text{process partitions}) = b\{n[C_r + (\log(x)C_{sc}) + C_w\} \quad (4.14)$$

where $x = \min(r, m/p)$. The total cost for Aggregate Algorithm B is therefore

$$T(\text{AlgoB}) = \begin{array}{ll} T(\text{execute by_qual}) & \text{if by_qual} \\ +T(\text{project by_list}) & \text{determine partitions} \\ +T(\text{execute src_qual}) & \text{if complex src_qual} \\ +T(\text{project source}) & \text{if unique aggregate} \\ +bnC_{sc} & \text{if unique aggregate} \end{array}$$

and the processing of partitions:

$$\begin{array}{ll} +b\{n[C_r + (\log(x)C_{sc}) + C_w\} & \\ +bnqC_{sc} & \text{if simple src_qual} \end{array}$$

4.2 Query Optimization in Parallel and Distributed Database Systems

In this section we give a short overview about query optimization and their used technology based on publications like [19, 1, 62, 63, 40]. Query optimization is an important factor for the success of relational database systems. Query optimizer for parallel database systems are more complicated than that for sequential query evaluation. The general problem of query optimization is known to be NP-hard even for centralized database systems, see Ibiraki [64]. Stonebraker et.al. [6] and Murphy [65] shows that a near optimal solution, in reasonable time, can be found when a given optimized sequential query execution plan is parallelized.

The reasons for the complexity are:

1. How to parallelize the query to execute
2. The cost model is more difficult due partitioning of data

4.2.1 Query Optimization

A query optimizer is one part in the execution of phases in query processing, typical steps or phases are depicted in figure 4.12 which was described in [1].

The first step in query processing is to parse the input. The Parser checks the syntax and translates the input (query expression) into an *internal representation*, in a *operator tree*. As an example the query expression

$$\pi_{customer-name}(\sigma_{branch-city = Brooklyn \wedge balance > 1000}$$

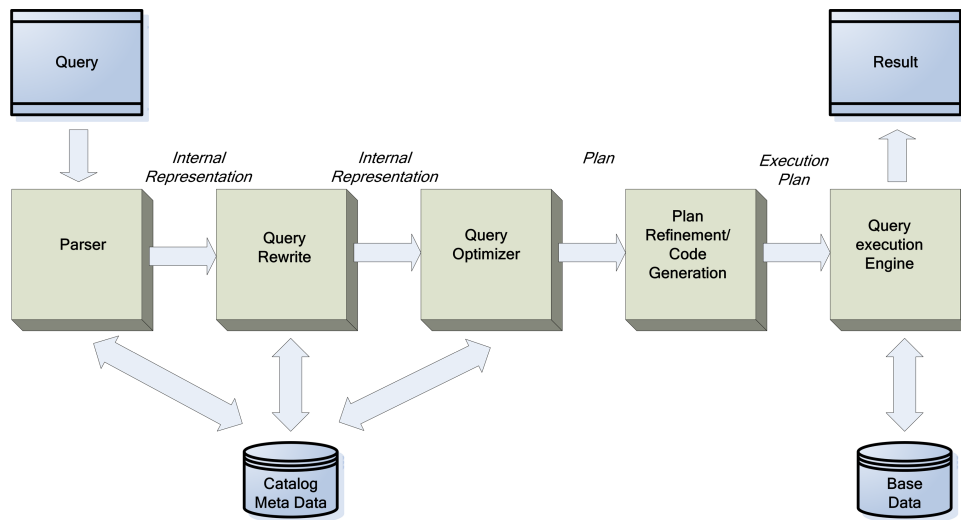


Abbildung 4.12: Phases of Query Processing

$(branch \bowtie (account \bowtie depositor)))$

can be translated in the *operator tree* seen in figure 4.13.

The next step is the Query Rewrite, it transforms the query e.g. by elimination of simplification of expressions, redundant predicates and unnesting of subqueries and views. Query Rewrite optimizes the query independent of the physical state (e.g., the table sizes, presence of indices, locations of copies of tables, speed of machines) of the database system.

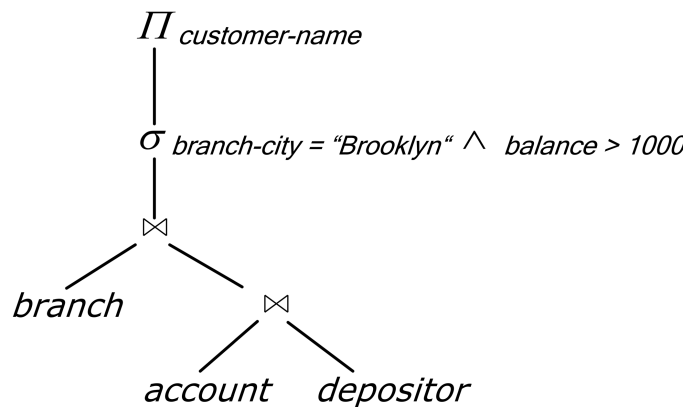


Abbildung 4.13: Operator Tree

The third step is to optimize the query depending on the physical state of the system. The optimizer decides:

1. Which methods (e.g., hashing or sorting) to use to execute
2. Which indices to use to execute a query
3. In which order to execute the operations of a query
4. How much main memory would be allocated
5. Which site each operation is to be executed

The result of the Query Optimizer are alternative plans to execute the query and chooses the best plan using a cost estimation model. A Query Execution Plan specifies in detail how the query is to be executed.

Alternate plans are generated by equivalence translation of the input query. After this translation all result expressions are been annotated with alternate algorithms to have different query plans and therefore different cost estimations. The optimizer takes the best plan in terms of cost. This process is called **Cost Based Optimizing**.

The fourth step is the **Plan Refinement/Code Generation**. This component transforms the plan produced by the optimizer into an executable plan. In most database systems the query execution plan is translated in a assembler-like language.

The **Query Execution Engine** provides generic implementations for every operator (e.g., send, scan, or **NestedLoopJoin**). Query execution engines are based on an iterator model [62]. The iterator model supports pipelining from one operator to another. The operators have the same interface and can be plugged together (consumer-producer). Therefore any plan can be executed and achieve good performance.

All the information for parsing, rewrite and optimize a query is been held in the **Catalog**. The data stored in the catalog is:

1. Schema of the database (tables, views, functions, integrity constraints, etc.)
2. Partitioning scheme of the tables and how they can be reconstructed
3. All physical information (Location of copies from tables, indices, statistics)

4.2.2 Interquery Parallelism

Different queries run concurrently to increase throughput of the system. Interquery parallelism is the easiest form of parallelism support in a database system, particularly in a shared-memory parallel architecture. Transactions on a shared-memory architecture running concurrently in a time-sharing manner. In a shared-memory architecture this transactions can run in parallel.

4.2.3 Intraquery Parallelism

Intraquery parallelism is used to speedup a single query. Two kinds of intraquery parallelism can be distinguished:

1. Inter-operational parallelism
2. Intra-operational parallelism

Inter-operational parallelism: A query can be parallelized by parallelizing individual operations that do not depend on one another.

Intra-operational parallelism: Operations can be parallelized by executing them in parallel on different subsets of the relations.

In this thesis we focused on **Intra-operational** parallelism and their optimization.

5 A Heterogenous Architecture for Parallel Database Operations

In this section we describe the modifications of the most important parallel database operations sort, join and aggregate in a static simplified grid organization and a performance analysis and comparison to the unmodified operations.

5.1 Modified Parallel Database Operations

The parameters for the calculation of the particular cost in the analysis are specified in Table 5.1. The values chosen reflect the characteristic parameters of actual hardware technology.

Tabelle 5.1: *Cost Values for Analysis*

name	value	description
H	0.85	hit ratio for the cache
H'	0.35	fraction amount of time a free page frame will be available in the cache during a write operation
R_c	16ms	costs of a cache to processor transfer
R_m	28ms	costs of a mass-storage transfer
C_{msg}	15ms	costs of I/O related message
k	128	tuples per page
C	$10\mu s$	costs of a simple operation (compare,add)
V	$225\mu s$	costs of moving a tuple inside a page
p_s	16kbyte	page size
t_l	150byte	tuple length
S	0.001	join selectivity factor
SS	0.1	semijoin selectivity factor
m		number of partitions (aggregate)
r		number of result tuples per page (aggregate)
q	10	number of operations to apply a simple qualification (aggregate)
t		number of output pages in aggregate

5.1.1 Modified Sort Operations

Based on the work of Bitton et al. in a generalized multiprocessor organization the block bitonic sort has in any case a better performance than the binary merge sort (see Figure 5.1 which is based on the analysis in [18]). To analyze the mapping of the algorithms onto a simplified Grid organization we have to pay special attention to the three phases of the algorithms as laid out in section 4.1.3. The last phase (postoptimal) of the Binary Merge Sort algorithm is split into three parts (see equation 5.1) because only one processor is necessary for $\frac{n}{2}$ cost.

$$\underbrace{[\log p - 1 + \frac{n}{2}]}_{\text{postoptimal}} \rightarrow \underbrace{\log p - 1}_{\text{postoptimal}_I} + \underbrace{\frac{n}{2}}_{\text{postoptimal}_{II}} \quad (5.1)$$

If for this "bottleneck" we choose the nodes of the Grid with the best network bandwidth available, the effect on the overall performance has to be at least noticeable. We specifically emphasize that even one single processing node with high network performance is worthwhile to exploit this effect. It is intuitively clear that this situation can be seen as normal in a heterogenous Grid organization, where nodes with different performance characteristics are the rule.

This leads to the clear policy for orchestration of a Grid workflow for a parallel binary merge sort to use nodes with the highest network performance in the *postoptimal_{II}* phase as laid out in algorithm 4.

On the other hand, using one high performance node in the bitonic sort gives no performance gain at all, because this node is slowed down by all other nodes working in parallel in a staged manner.

The effect that the binary merge sort now outperforms the block bitonic sort in a simplified Grid organization is shown in Figure 5.2 by the line labeled "binary merge modified" (please notice the logarithmic scale of the values).

This effect can easily be explained by Amdahl's law too, where simply said the performance of a parallel algorithm is dependent on its sequential part. More specific, Amdahl's law is stating that the speedup is limited by the sequential part. It is intuitively clear that even the most powerful node will limit the performance increase if the number of used nodes for the whole parallel algorithm is increasing. A speedup and scale-up analysis will clear up this issue.

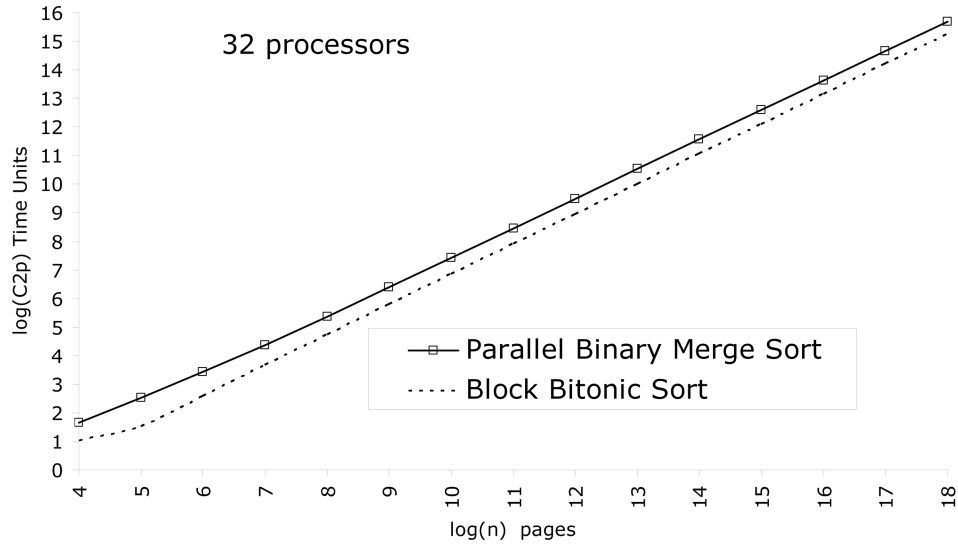


Abbildung 5.1: Sort in a Generalized Multiprocessor Organization

5.1.2 Modified Join Operations

In the simplified grid organization the Merge-Sort join algorithm based on the bitonic-sort outperforms the Merge-Sort join based on the binary merge sort up to 2^6 processors, see

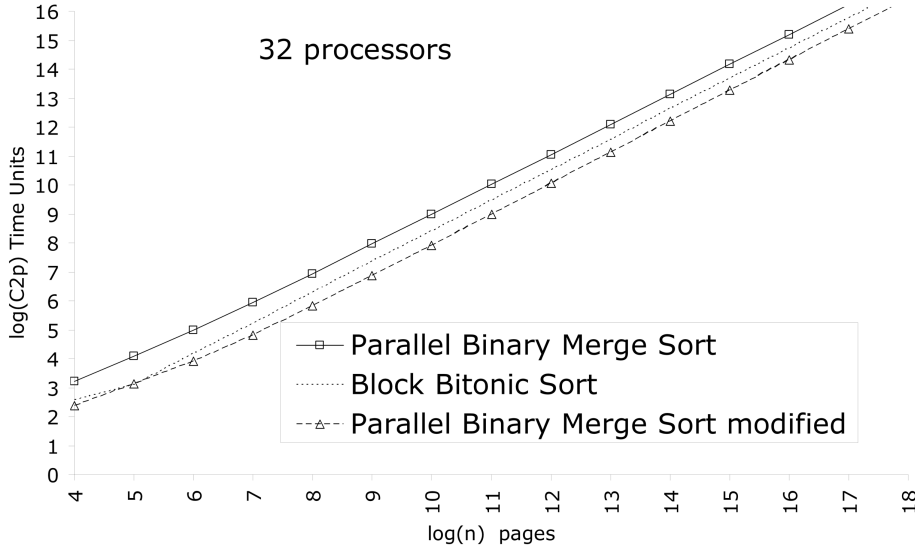


Abbildung 5.2: Sort in a Simplified Grid Organization

”Parallel Merge-Sort Join (Binary Merge Sort) modified” in Figure 5.3.

As in the multiprocessor organization we have analyzed the join algorithms in the simplified grid organization with a selectivity factor between 0.05 and 0.9. The effect is the same as for the generalized multiprocessor organization, the difference between the merge-sort algorithm (Parallel Merge-Sort Join (Bitonic) and Parallel Merge-Sort Join (Binary Merge)) remains constant (if $S \geq 0.05$), see Figure 5.4. The reason for it is, that the merge cost have an insignificant influence on the overall cost of the algorithm. On the other hand, in this case the Parallel Merge-Sort Join (Binary Merge) modified is always faster (in terms of C_p^2 costs) than the unmodified version. The Hashed Join algorithm in a Simplified Grid Organization performs similarly as in the multiprocessor organization. And also in the case of using nodes with a buffer greater than ten percent of R the Hashed Join outperforms all other analyzed algorithms.

5.1.3 Modified Aggregate Operations

We chose Aggregate Algorithm A for analysis within a Simplified Grid Organization. The reason is clear, the project operation is done by a parallel binary merge sort, and as seen in the performance analysis of the sort algorithms in a Simplified Grid Organization, the last processor is the bottleneck in the algorithm. We modify the workflow orchestration of Aggregate Algorithm A to exploit the highest bandwidth in the network accordingly. As a result the algorithm is now faster than without modifying it, which can be seen in Figure 5.5.

5.2 Performance Analysis and Comparison of the Modified Operations

5.2.1 Sort Operations

5.2.1.1 Speedup and Scale-up of the Sort Algorithms

The speedup of the parallel binary merge versus block bitonic sort algorithm is shown in Figure 5.6. As expected by the discussion above the speedup of the parallel bitonic sort is

16384 pages join 1024 pages

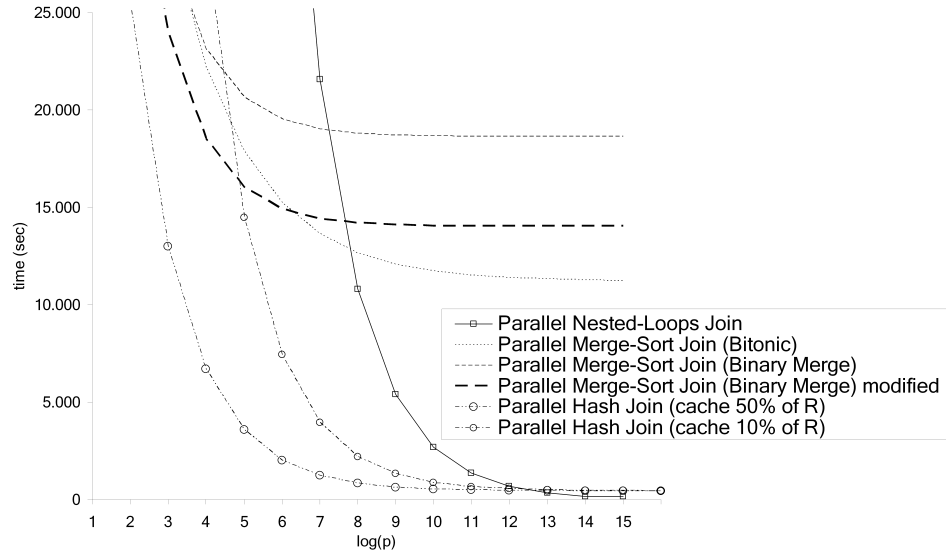


Abbildung 5.3: Join in a Simplified Grid Organization with $S=0.001$

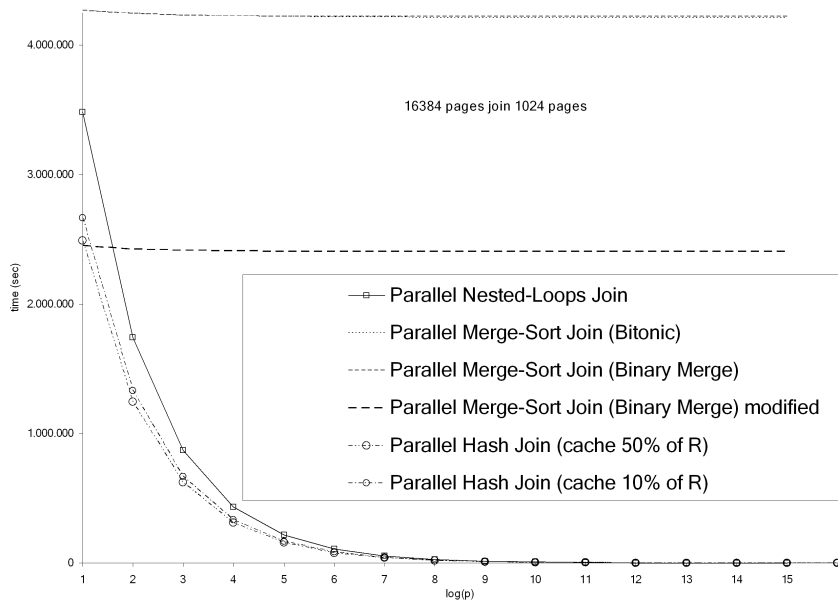


Abbildung 5.4: Join in a Simplified Grid Organization with $S=0.5$

far beyond the speed-up of the parallel binary merge because of the sequential part of the postoptimal phase. The use of a processing node with higher performance is only limiting the effect in absolute numbers but has clearly no influence on the algorithmic performance behavior. It has to be noted that the numbers of processing nodes shown in the speedup

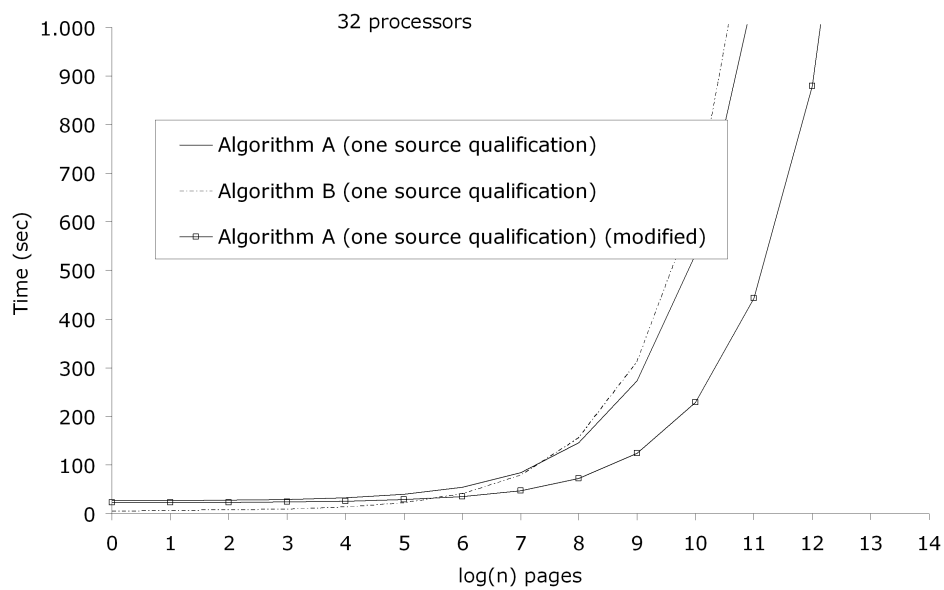


Abbildung 5.5: Aggregate in a Simplified Grid Organization

Figure 5.7 are unrealistically high and for a more realistic situation as shown in Figure 5.2 the smart use of high performance processors is definitely worth the effort.

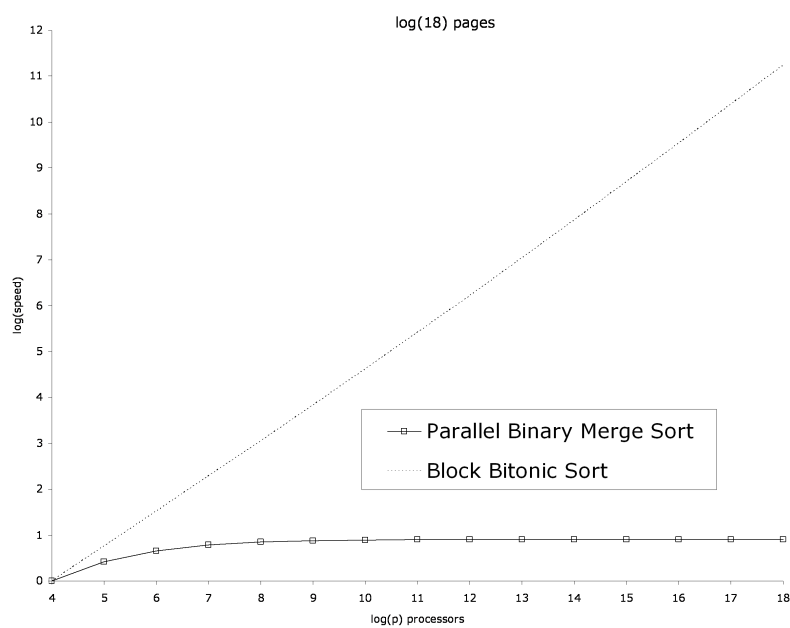


Abbildung 5.6: Sort Speedup in a Generalized Multiprocessor Organization

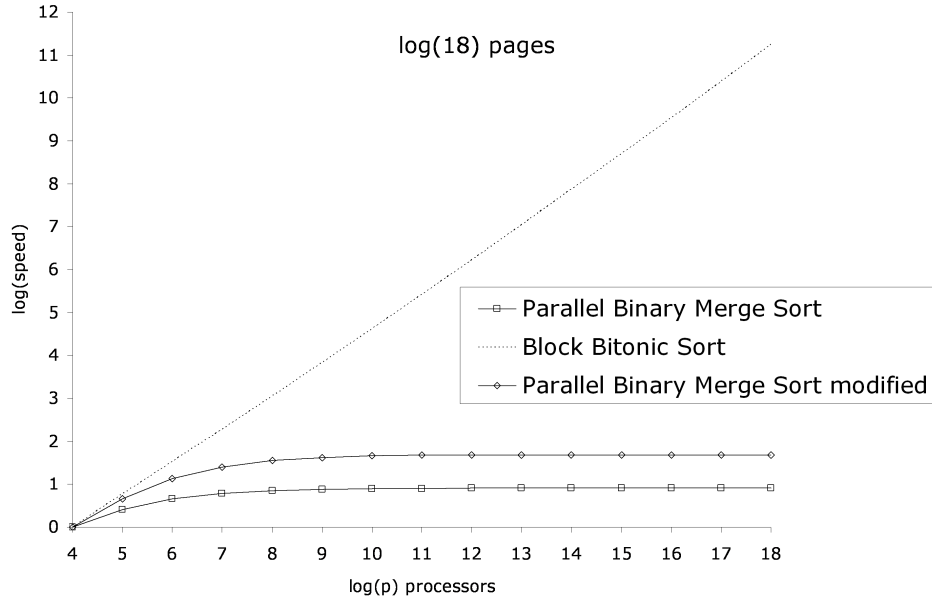


Abbildung 5.7: Sort Speedup in a Simplified Grid Organization

An analysis of the scale-up numbers as depicted in Figure 5.8 shows a similar result. Also here the scale-up of the modified parallel binary merge outperforms the classical parallel binary merge in absolute numbers only, but can not cope with the bitonic sort.

5.2.2 Join Operations

Similar to the effects on the performance of the Sort Algorithms in a Generalized Multiprocessor Organization, the Merge-Sort Join algorithm with the block bitonic sort outperforms the Nested-Loop Join and also outperforms the Merge-Sort Join based on the binary-merge sort algorithm, unless the number of processors available is close to the larger relation size. Figure 5.9 shows the join algorithms with a selectivity factor of 0.001. If the ratio between the relation sizes is significantly different from 1, the nested-loop algorithm outperforms the merge-sort (except for a small numbers of processors). For lower selectivity factor values, the merge-sort algorithm performs better than the nested-loop algorithm because the merge step (handled by a single processor) has to output fewer pages. The Hashing Join algorithm is strongly influenced by the available memory of the nodes used in the algorithm. In 5.9 we can see the performance of the Hashed Join with ten percent or fifty percent of the pages in the smaller relation R of memory available in the nodes. We remember that the algorithms, except the Hashing Join, only have three pages as buffer. In the case of using nodes with buffers greater than ten percent of R the Hashed Join outperforms all other analyzed algorithms.

In a generalized multiprocessor organization we have analyzed the algorithms with a selectivity factor between 0.001 and 0.9. The effect is, that the difference between the two merge-sort algorithms stays constant (if $S \geq 0.05$). The reason is, that the merge cost are only marginal to the overall cost of the algorithm.

In the simplified grid organization the Merge-Sort join algorithm based on the bitonic-sort outperforms the Merge-Sort join based on the binary merge sort up to 2^6 processors,

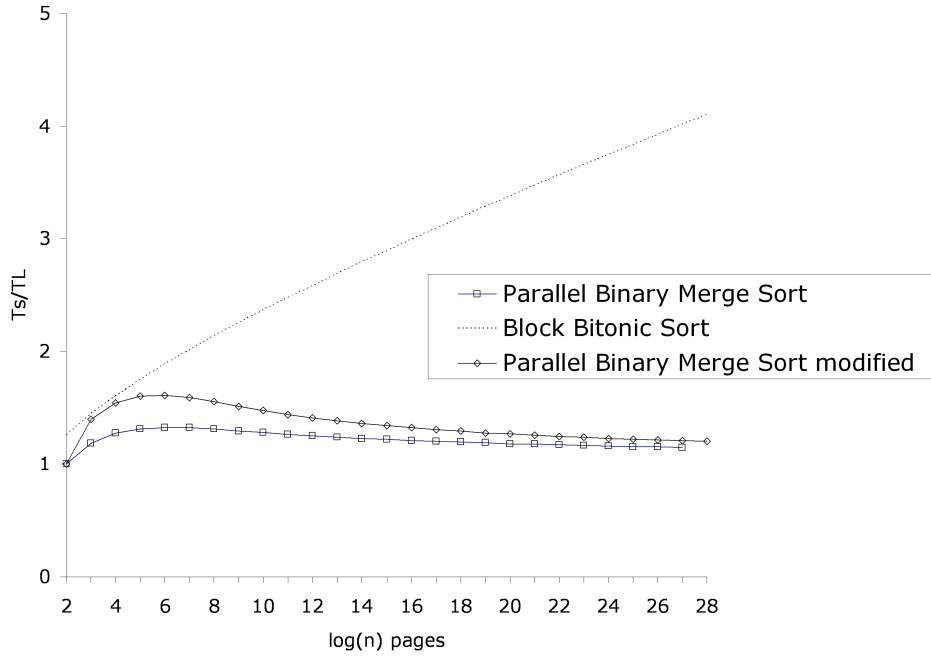


Abbildung 5.8: Sort Scale-up in a Simplified Grid Organization

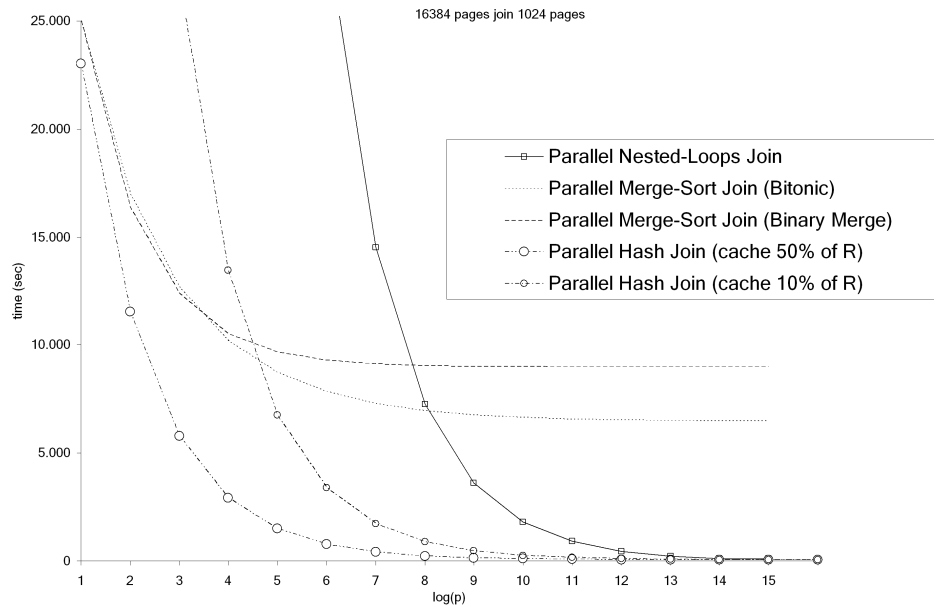


Abbildung 5.9: Join in a Generalized Multiprocessor Organization with $S=0.001$

see "Parallel Merge-Sort Join (Binary Merge Sort) modified" in Figure 5.3.

As in the multiprocessor organization we have analyzed the join algorithms in the simplified grid organization with a selectivity factor between 0.05 and 0.9. The effect is the

same as for the generalized multiprocessor organization, the difference between the merge-sort algorithm (Parallel Merge-Sort Join (Bitonic) and Parallel Merge-Sort Join (Binary Merge)) remains constant (if $S \geq 0.05$), see Figure 5.4. The reason is, that the merge cost have insignificant influence on the overall cost of the algorithm. On the other hand, in this case the Parallel Merge-Sort Join (Binary Merge) modified is always faster (in terms of C_p^2 costs) than the unmodified version. The Hashed Join algorithm in a Simplified Grid Organization performs similarly as in the multiprocessor organization. And also in the case of using nodes with a buffer greater than ten percent of R the Hashed Join outperforms all other analyzed algorithms.

5.2.2.1 Speedup and Scale-up of the Join Algorithms

The speedup in a generalized multiprocessor organization of the investigated join algorithms is depicted in Figure 5.10. The parallel nested-loop join has a linear speedup. The sharp bend at the end of the curve shows that the number of processors becomes equal to the number of pages of the larger relation. In contrast the parallel merge-sort based on the bitonic sort has a moderate speedup until the number of processors is greater than 2^7 . Beyond this number of processors, no more speedup can be reached. In a Simplified Grid Organization the speedup is similar to that in a generalized multiprocessor organization, see Figure 5.12. The modified version of the parallel merge-sort join has a better speedup than the unmodified version. A nearly linear speedup is also seen in the Hashing Join algorithm. In the case of using ten percent of the number of pages as buffer for the smaller relation R the speedup is more linear than using fifty percent of the number of pages. This circumstance can be used for the workflow orchestration of selecting nodes to choose the best performing algorithm, especially in a Simplified Grid Organization.

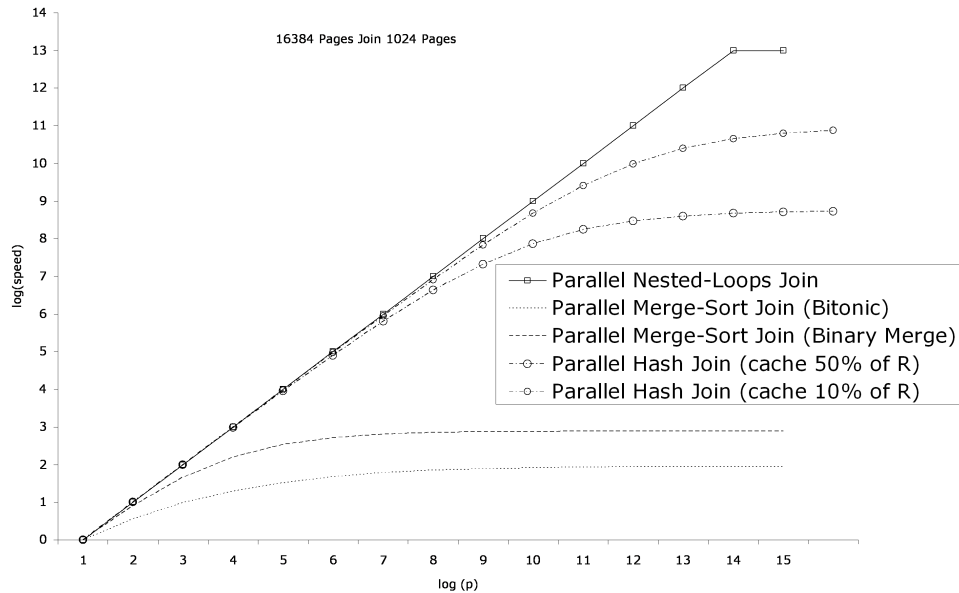


Abbildung 5.10: Join Speedup in a Generalized Multiprocessor Organization with $S=0.001$

The scale-up is depicted in Figure 5.11. The parallel nested-loop join shows the best scale-up effect. The Merge Sort algorithms (also the modified Version of the Merge Sort

based on the Binary Merge Sort) have the best scale-up. The Hashing Join algorithm has similar scale-up as Nested Loop.

The hump at $\log(n) = 11$ is the effect of the size of the relation n size $2^{10} = 1024$. Setting the selectivity factor much larger than 0.001 (like in the analysis of the overall C_p^2 cost evaluation) the effect is, that the scale-up of the three investigated merge-sort algorithms shows the same scale-up even with a lower number of processors. Like the speedup, the scale-up in a Simplified Grid Organization is similar to that in a generalized multiprocessor organization, see Figure 5.13. The modified version of the parallel merge-sort join is nearly identical with the unmodified version.

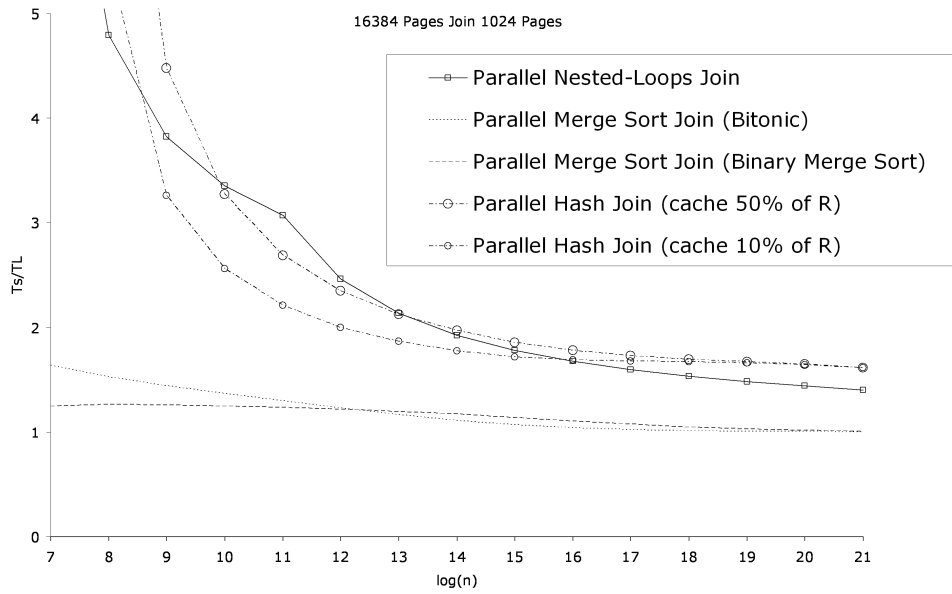


Abbildung 5.11: Join Scale-up in a Generalized Multiprocessor Organization with $S=0.001$

5.2.3 Aggregate Operations

The performance in a Generalized Multiprocessor Organization of the two aggregate algorithms is depicted in figure 5.14. We assume for both algorithms one source qualification (simple qualification) and 32 processors. The parameters of the analysis are listed in table 5.1. When the query contains a *src_qual*, Aggregate Algorithm B performs better except when the relation is very large. The performance of Aggregate Algorithm A is sensitive to the number of partitions. Both algorithms process the *by_qual* in the same way, the results shown are independent of the use of a *by_qual*.

We chose Aggregate Algorithm A for analysis within a Simplified Grid Organization. The reason is clear, the project operation is done by a parallel binary merge sort, and as seen in the performance analysis of the sort algorithms in a Simplified Grid Organization, the last processor is the bottleneck in the algorithm. We modified the workflow orchestration of Aggregate Algorithm A to exploit the highest bandwidth in the network accordingly. The result is that the algorithm is now faster than without modifying it, which can be seen in Figure 5.5.

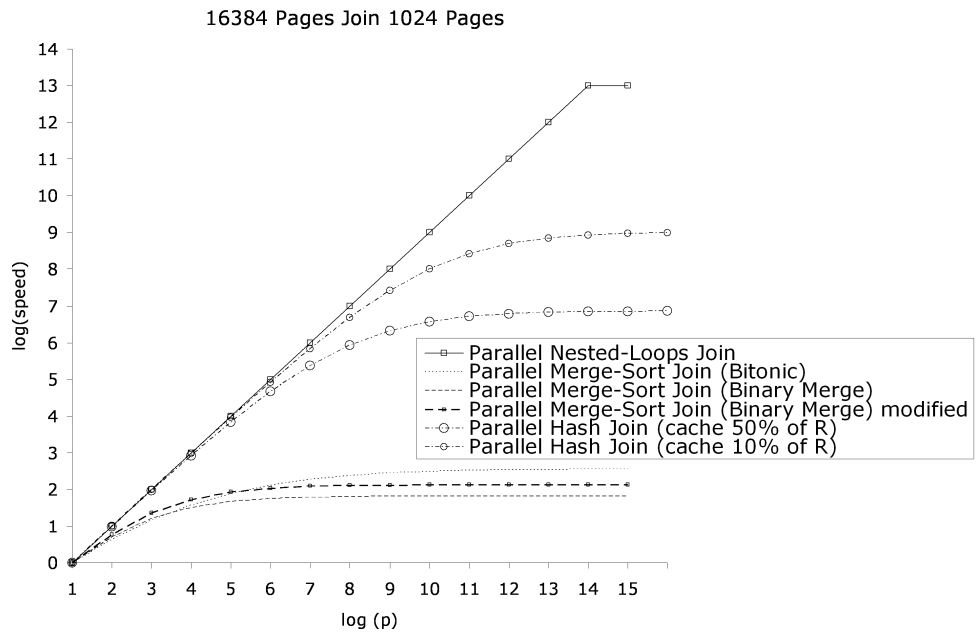


Abbildung 5.12: Join Speedup in a Simplified Grid Organization with $S=0.001$

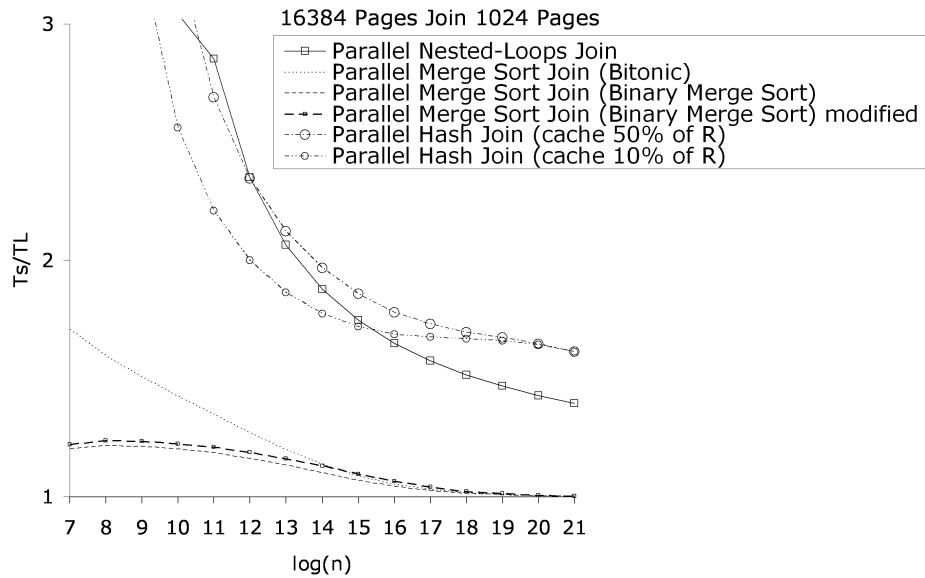


Abbildung 5.13: Join Scale-up in a Simplified Grid Organization with $S=0.001$

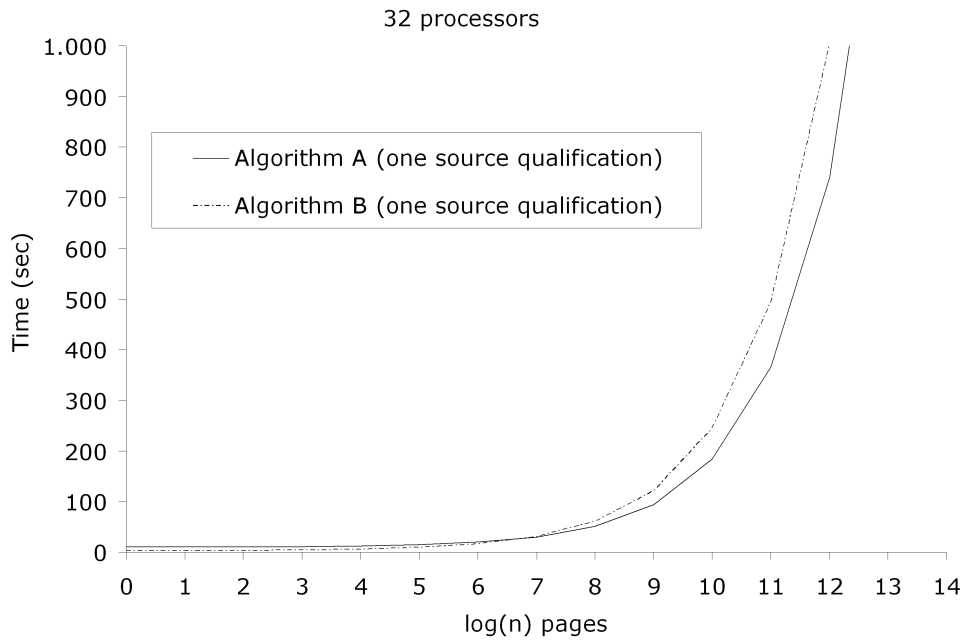


Abbildung 5.14: Aggregate in a Generalized Multiprocessor Organization

5.2.3.1 Speedup and Scale-up of the Aggregate Algorithms

The speedup in a generalized multiprocessor organization of the investigated aggregate algorithms is depicted in Figure 5.15. Note, Aggregate Algorithm A shows the better speedup. The investigated aggregate algorithms in a Simplified Grid Organization have similar speed-up behavior as the algorithms in a generalized multiprocessor organization. This is depicted in Figure 5.17. The modified version of the aggregate algorithm A shows a better speed-up than the unmodified version. The scale-up of the aggregate algorithms in a Generalized Multiprocessor Organization is depicted in Figure 5.16. The parallel Aggregate Algorithm A shows the best scale-up effect. But as seen in Figure 5.18, the scale-up of the modified Algorithm A in a Generalized Multiprocessor Organization and the one in a Simplified Grid Organization are nearly the same.

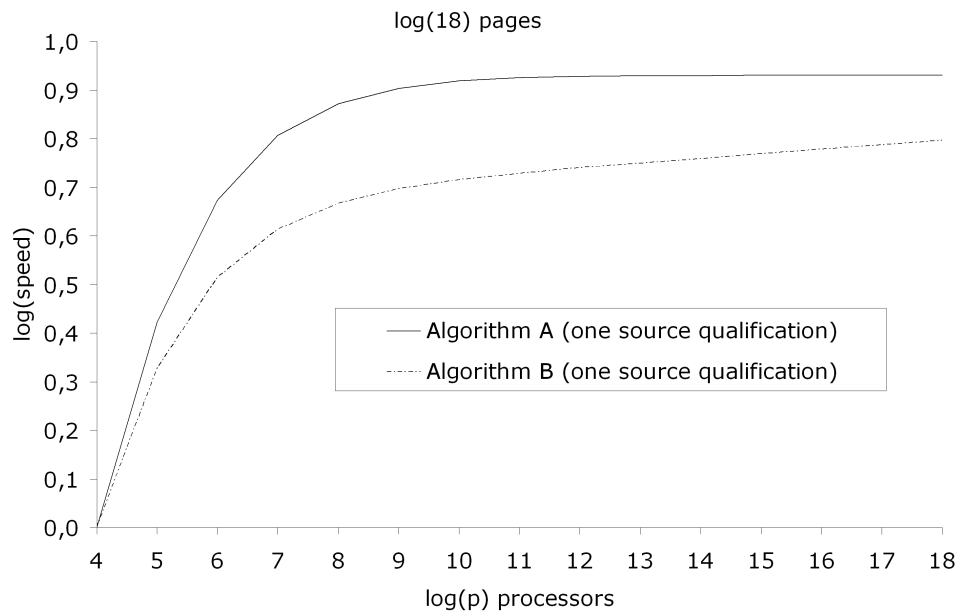


Abbildung 5.15: Aggregate Speedup in a Generalized Multiprocessor Organization

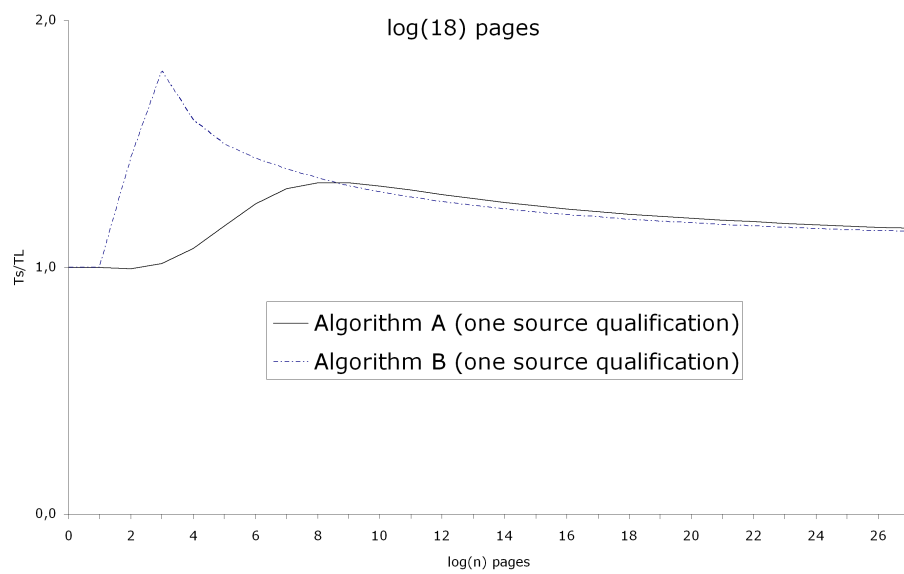


Abbildung 5.16: Aggregate Scale-up in a Generalized Multiprocessor Organization

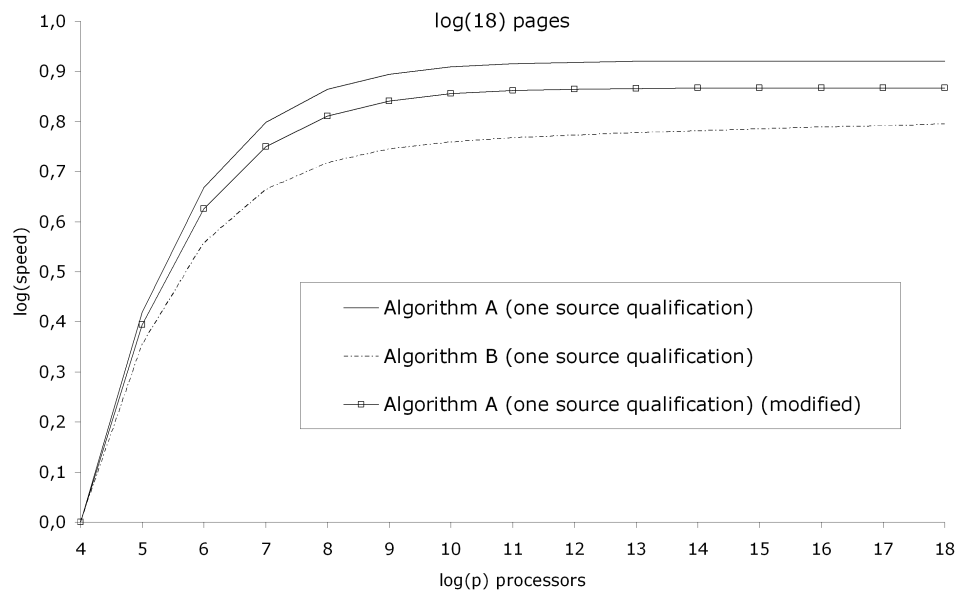


Abbildung 5.17: Aggregate Speedup in a Simplified Grid Organization

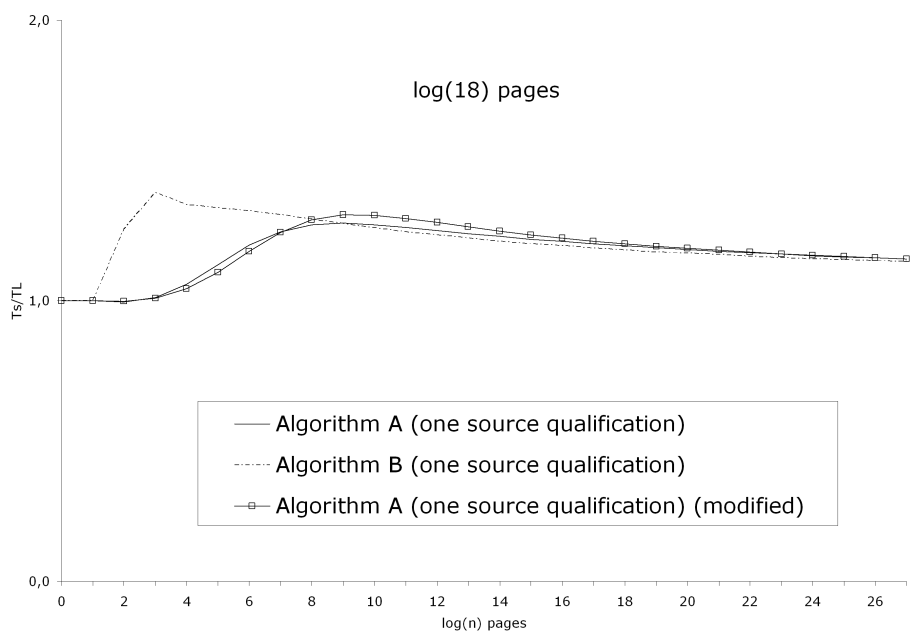


Abbildung 5.18: Aggregate Scale-up in a Simplified Grid Organization

6 Optimization of Workflow Orchestration

The general problem of query optimization is known to be NP-hard even for centralized database systems, see Ibaraki [64]. Ibaraki and Kameda prove that the problem of minimizing page fetches in multirelational joins by means of a nested loops is in general NP-complete.

As such, heuristic solutions are often needed to find good plans for query execution.

In this chapter we present an algorithm to find all perfect binary trees in a given undirected graph. First we define a graph representation of the Static Simplified Grid Organization.

6.1 Workflow Orchestration

6.1.1 Algorithm for the Workflow Orchestration

The cost of disk accesses are therefore much less influencing the overall performance than the network cost. Focusing on the sort algorithms the impact on the performance of the sort merge algorithms depends predominantly on the network bandwidth. Therefore the nodes with the best network-bandwidth should be grouped to perform the last (*postoptimal_{II}*) phase in the binary merge sort (see equation 5.1). One stage in this last phase consists of a number of sender and a (number of) receiver nodes. The algorithm of defining the layout of the workflow for the postoptimal phase can be described by choosing the nodes with the best network bandwidth starting from the final stage. If the bandwidth is the same for some nodes the ones with the best computational power have to be chosen then. This can be described by the algorithm 4:

Algorithm 4: Workflow Orchestration Algorithm for Grids

Input: Available Nodes and Algorithmic Sub Task

Output: Node Layout according to Algorithm Deployment)

- 1 Determine the network bandwidth and processing power for each processing node;
 - 2 Sort nodes according to network bandwidth;
 - 3 Nodes with equal network bandwidth in the sequence are sorted according to their processing power;
 - 4 Identify postoptimal phase as binary tree structure with node creating final run of length m as root;
 - 5 Starting from the beginning of sequence (i.e. best node first) map nodes level-wise from right to left beginning from root (root is level 0, successors of root are level 1, etc.);
-

6.1.2 A Graph Representation of the Static Simplified Grid Organization

For the representation of the Static Simplified Grid Organization we use the common definition of a weighted undirected graph. An example of such a graph is depicted in figure 6.1. For better understanding it is necessary to explain some definitions of graph theory.

A graph G is a pair (V, E) , where V is a set of vertices, and E is a set of edges between the vertices $E \subseteq \{\{u, v\} | u, v \in V\}$. An undirected graph is a graph whose edges are unordered

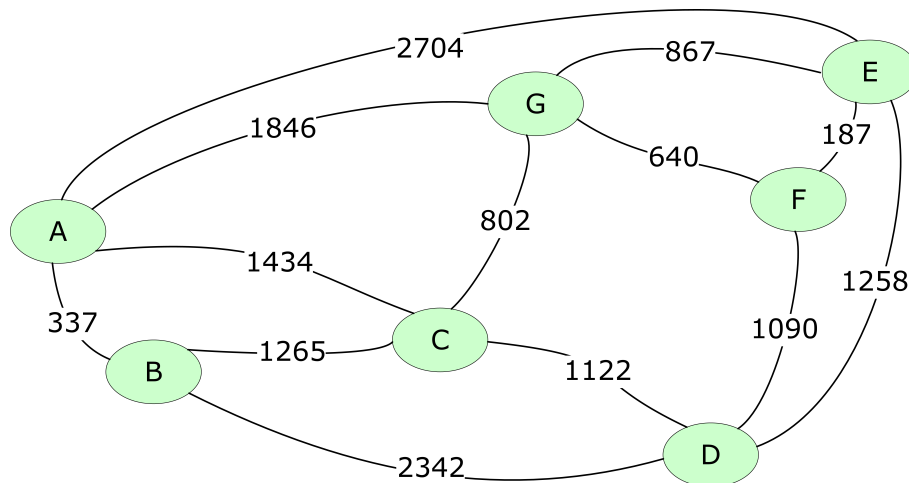


Abbildung 6.1: A Weighted Undirected Graph

pairs of vertices. That is, each edge connects two vertices. A weighted graph associates a label (weight) to every edge in the graph. Weights in our definition are restricted to integers. The weight of a path or the weight of a tree in a weighted graph is the sum of the weights of the selected edges. The notion **cost** can be used equally to **weight**. A **full binary tree** is a tree in which every node other than the leaves has two children. A **perfect binary tree** is a full binary tree in which all leaves are at the same depth or same level. A perfect binary tree has $2^{h+1} - 1$ nodes, where h is the height of the tree [49]. Sometimes perfect binary trees called **complete** trees. In figure 6.2 we see 5 examples of perfect binary trees, from height $h = 0, 1, 2, 3, 4$.

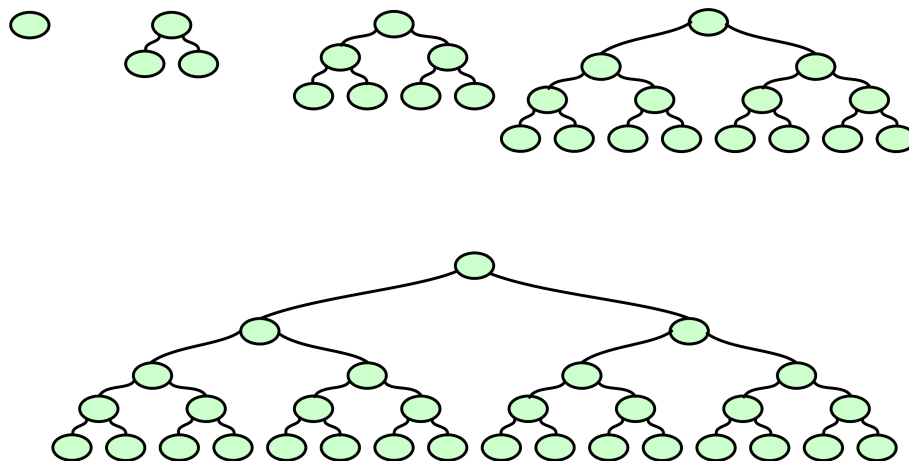


Abbildung 6.2: Perfect Binary Trees of height $h = 0, 1, 2, 3, 4$

6.1.3 A Binary Tree Search Algorithm

As explained in section above, to execute the parallel binary Merge Sort it is necessary to arrange nodes to build a **perfect binary tree**. It is easy to understand, that the maximum performance of the parallel binary Merge Sort can be reached by minimizing the connection weights (i.e. network bandwidth) between each level of the perfect binary tree.

We start by repeating the motivation and the basic strategy for our approach again. In homogeneous environments all nodes and networks are equal, therefore it does not matter

how the nodes are arranged in the binary tree. But in a heterogeneous environment, the arrangement of the nodes influences the performance (execution time) heavily, because the processing power and the bandwidth of the network connections between nodes can vary and influence the execution time of each step of the binary Merge Sort. The behavior of the parallel binary Merge Sort in a heterogeneous environment lead to arrange the nodes and their edges in the binary tree with the highest bandwidth on top (the root). That means we start the process by choosing the root node with two connections to other nodes with the highest bandwidth. With an underlying representation of the existing environment as a graph with weighted edges the optimal solution for the Binary Merge Sort can be found by performing a search of all possible perfect binary trees starting from every node and choose the binary tree with the best over all execution time. This is very time consuming for large networks (graphs) and large perfect binary trees to found. Therefore we propose an optimized algorithm to determine the optimal node arrangement as perfect binary tree to perform a parallel binary tree with the lowest overall execution time.

We define an algorithm, which is based on PRIM's algorithm [49] for finding a minimum spanning tree in an undirected weighted graph. The algorithm to find all binary trees with minimal weight is described in Algorithm 5.

Algorithm 5: Binary Tree Search Algorithm

Input: undirected graph $G(V, E)$ with weights of the edges, maxdepth of tree
Output: all possible perfect binary trees from the graph with depth 'maxdepth'

```

1 forall nodes in the graph do
2   create empty binary tree;
3   set actual node to root;
4   for level:=0 to maxdepth do
5     forall leafs in the current binary tree do
6       search for minimum edge from leaf that is not already in the tree;
7       if edge found then
8         add node to binary tree as left child;
9       search for minimum edge from leaf that is not already in the tree;
10      if edge found then
11        add node to binary tree as right child;
12  if binary tree not empty and binary tree is a perfect binary tree then
13    save binary tree in a list of trees;
14  delete binary tree;

```

6.2 Implementation of the Perfect Binary Tree Search Algorithm

We implemented the search algorithm to find the optimal solution in terms of minimal execution time of the parallel binary Merge Sort algorithm. Our goal was using realistic networks (graphs) as input. For this purpose we investigated Internet random topology generators, as described in section 6.2.1, to analyze the behavior of our search algorithm in realistic scenarios.

6.2.1 Generating Random Graphs

Studies on generating random graphs dates back to Erdős and Rényi [66] [67]. The process generating a random graph can be divided into two steps. Step one is generating a set of n vertices, the second step is adding edges between them at random. As we see later an important property of graphs are the node degree and their distribution in the graph. The node degrees in a network is the number of connections or edges the node has to other

nodes. The degree distribution is the probability distribution of these degrees. Networks like protein networks, citation networks, the Internet and some social networks have degree distributions that approximately follow a power law. The fraction $P(k)$ of nodes in the network having k connections to other nodes goes for large values of k as $P(k) \sim k^{-y}$ where y is a constant. Such networks are also called scale-free networks.

We used *tiers* [68] and *BRITE* [69] to generate input graphs. *tiers* generates a random graph based on the Minimum Spanning Tree algorithm [70] [49]. *Tiers* is based on a three-level hierarchy aimed at reproducing the differentiation between Wide-Area (WAN), Metropolitan-Area (MAN) and Local-Area networks (LAN) comprising the Internet. In the original version *tiers* does not generate random weights of the edges therefore we altered the source-code to generate random edge weights. Networks generated with *tiers* can be pictured by using *gnuplot* [71].

On the other hand *BRITE* implements the Waxman [72] and Barabási-Albert [73] generation model.

Waxman defines a probability model for interconnecting the nodes of the topology, which is given by the power law function: $P(u, v) = \alpha e^{-d/(\beta L)}$ where $0 < \alpha, \beta \leq 1$, d is the Euclidean distance from node u to node v , and L is the maximum distance between any two nodes. The nodes in the network are distributed at random across a Cartesian coordinate grid. A large value of α increases the number of connections, a large value of β increases the number of edges from each node.

BRITE implements also the Barabási-Albert model in which a network grows incrementally and the nodes interconnect with preference towards higher degree nodes. *BRITE* generated graphs can be viewed by the general purpose visualization tool *otter* [74]. Some example generated graphs, which built the basis for our analysis, are shown in Figure 6.4 and Figure 6.5. These two Figures should also demonstrate visually the enormous number of edges as input to the proposed algorithm.

6.2.2 Analysis of the Perfect Binary Tree Search Algorithm

We investigated the influence of the different random generation models and parameters on our binary tree search algorithm. The node degree of the generated graphs shows the most important influence on the number of possible perfect binary trees. With the generated binary trees we calculated the execution time of the parallel binary Merge Sort to find the minimum execution time of the algorithm using the particular binary trees. For the evaluation of our **Perfect Binary Tree Search Algorithm** we have implemented a complete test suite with a comfortable GUI (graphical user interface) and various features for importing generated graphs and exporting the results to a spreadsheet software. Figure 6.3 shows the test suite with a *BRITE* generated graph having 1000 nodes and a mean degree $m = 4$.

The results of our investigation can be summarized as follows:

1. Networks with higher degrees contain much more perfect binary trees and deeper perfect binary trees. See Table 6.1 and Figure 6.4 and Figure 6.5.
2. The minimum execution time shows always these trees where the last level delivers the minimum transfer time (that means the fastest connection).
3. The last processor (precisely the last three processors and the bandwidth between them) contributes more than 50 percent to the perfect execution time between 8 and 16 processors. Figure 6.6 shows this influence of the last three processors and their bandwidth in percentage of the overall execution time.
4. The splitting of the algorithm in two parts (part one and part two, as shown in equation 6.1) and computing the permutation of min and max values of C_p^2 gives the

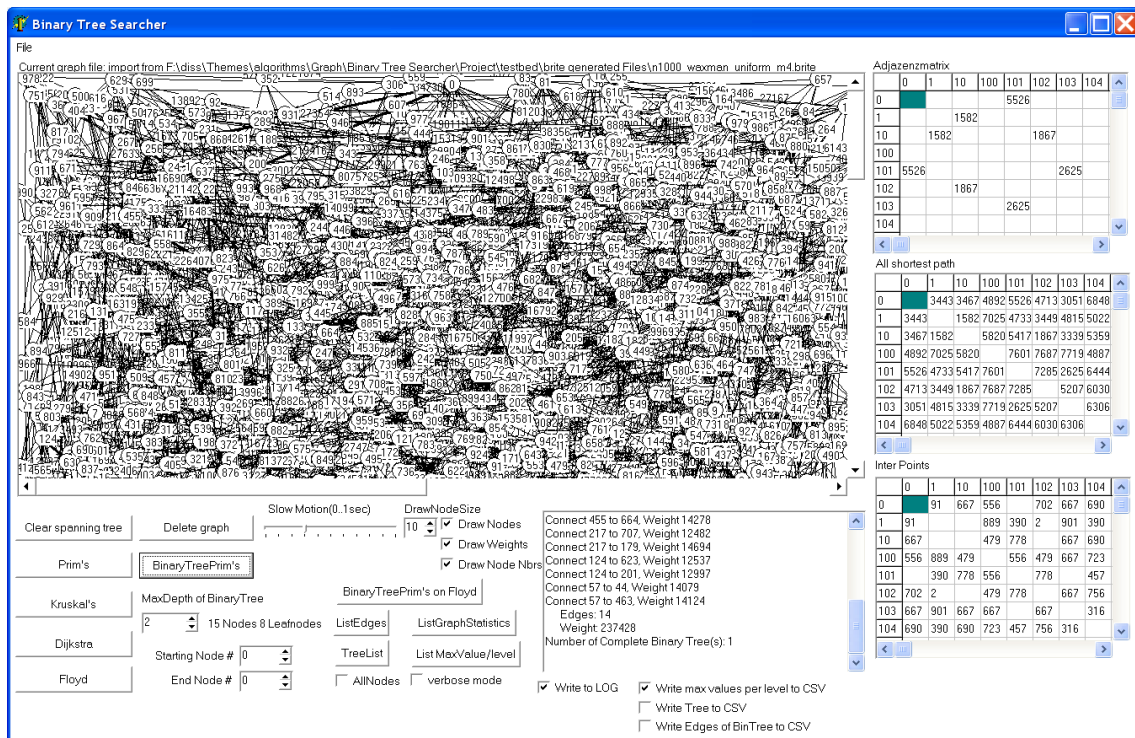


Abbildung 6.3: Perfect Binary Tree Searcher Test Suite

result, that the avg/min (average/minimum) combination is always the second-best solution for minimum execution time. Therefore this is the optimum solution. Note that a min/min configuration is not possible, except all edges show nearly the same weight. In this case the avg/min is also the optimum solution, because the average (avg) of minimum (min) is minimum (min). Figure 6.7 depicts this situation.

Note: Table 6.1 shows an extract of the results of our graph analysis.

$$\underbrace{\frac{n}{2p} \log\left(\frac{n}{2p}\right) + \frac{n}{2p}}_{partone} + \underbrace{\log p - 1 + \frac{n}{2}}_{parttwo} \quad (6.1)$$

Tabelle 6.1: Number of Binary Trees found in generated Graphs

name	nodes	edges	degree	16 leafs	32 leafs	64 leafs	128 leafs	256 leafs
n1000-waxman-uniform	1000	2000	2	44	-	-	-	-
n1000-waxmann-uniform	1000	2000	2	36	1	-	-	-
n1000-waxmann-alpha09-uniform	1000	2000	2	40	-	-	-	-
n1000-waxmann-uniform-m4	1000	4000	4	998	984	886	354	-
n2000-waxman-uniform	2000	4000	2	81	-	-	-	-
n4000-waxmann	4000	8000	2	170	2	-	-	-
...

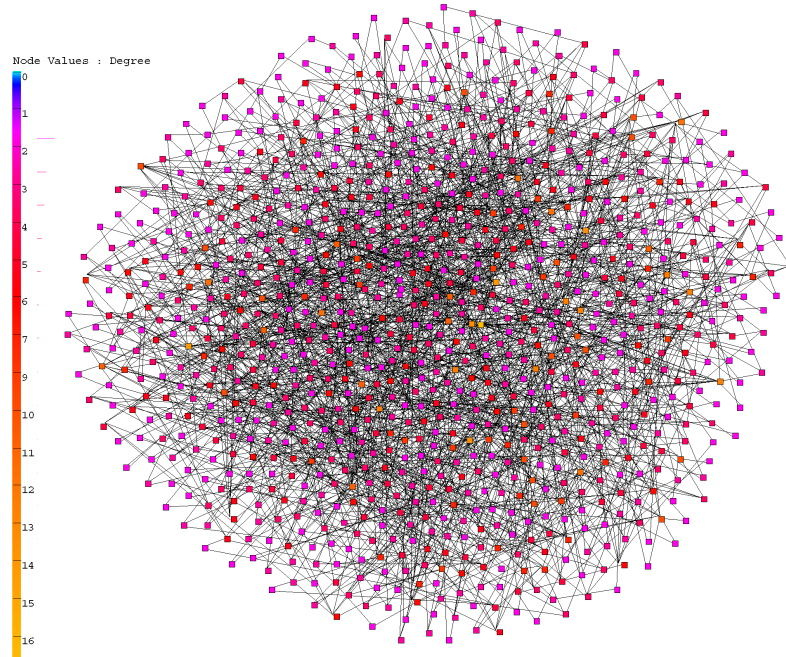


Abbildung 6.4: 1000 nodes with degree $m=2$

6.3 The Optimized Workflow Execution Process

Now we refine and speedup our algorithm for finding optimal perfect binary trees in a given graph in Algorithm 6.

Algorithm 6: Optimized Binary Tree Search Algorithm

Input: Environment representation as weighted undirected graph

Output: Optimal node arrangement as perfect binary tree

- 1 Search for all perfect binary trees with level 1 (root and two children);
 - 2 Sort the found binary trees according to their weight (sum of the two edges)(ascending);
 - 3 Choose the root from binary tree with the lowest weight (sum of the two edges);
 - 4 Perform a binary tree search with the found root as described in algorithm 2 (Binary Tree Search Algorithm) with the necessary depth for the binary Merge Sort;
 - 5 Output the perfect binary tree to perform the optimal parallel binary Merge Sort;
-

Based on the findings of section 6.2.2 it is not necessary to search for all perfect binary trees with the necessary full level depth. A search for all perfect binary trees with three nodes (trees of height 1) and only one search (from the starting node of the found optimal binary tree) with the necessary full level depth has to be done to find the optimum binary tree configuration to get a minimum execution time of the parallel binary Merge Sort.

The dramatic improvement can be seen in Figure 6.8. The diagram shows the relative runtime in numbers of child searches. A **child search** means, that given a starting node, two connected nodes that are not already in the tree should be found. One node for the left child of a node and one for the right child.

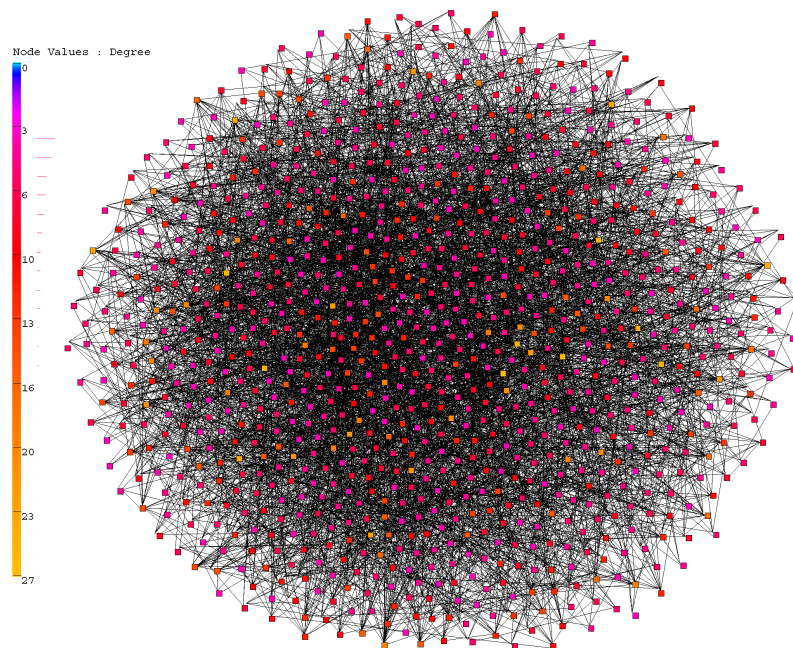


Abbildung 6.5: 1000 nodes with degree $m=4$

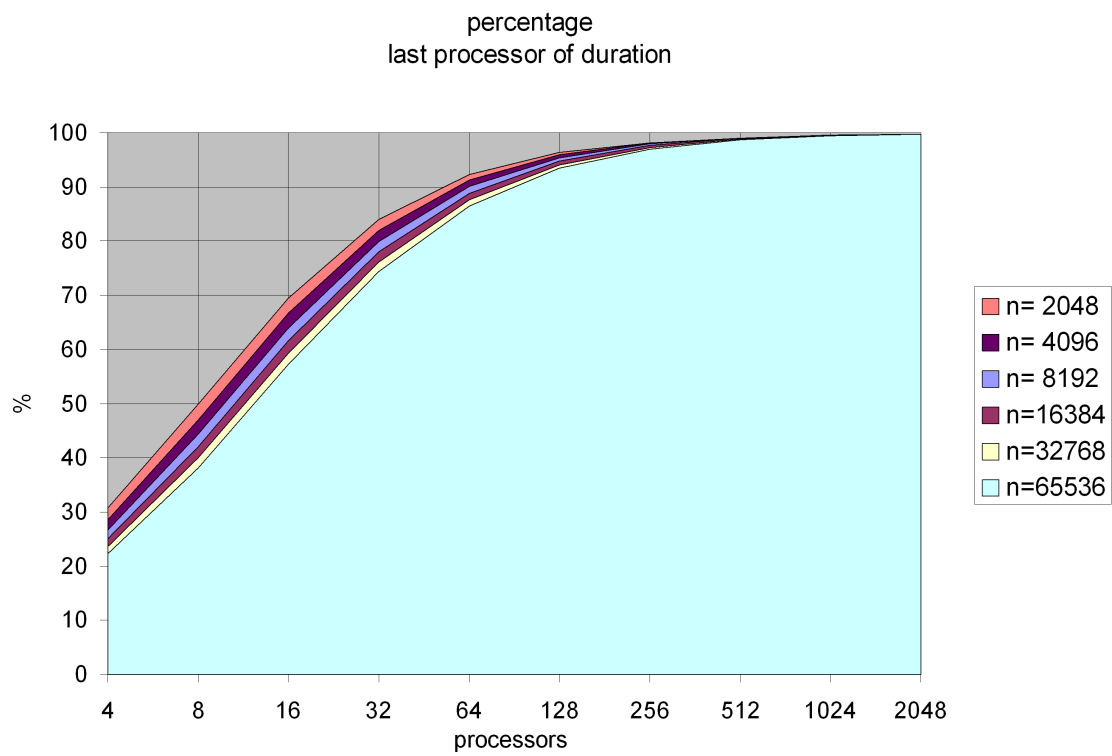


Abbildung 6.6: Contribution to execution time of last three processors

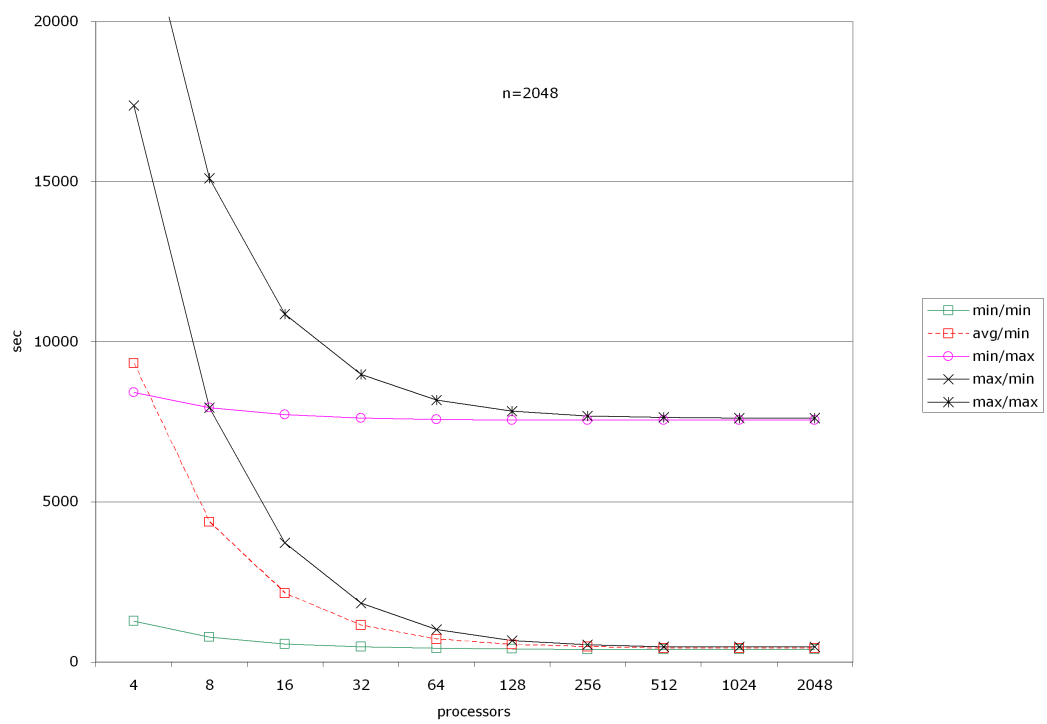


Abbildung 6.7: Performance influence of partone and parttwo combinations

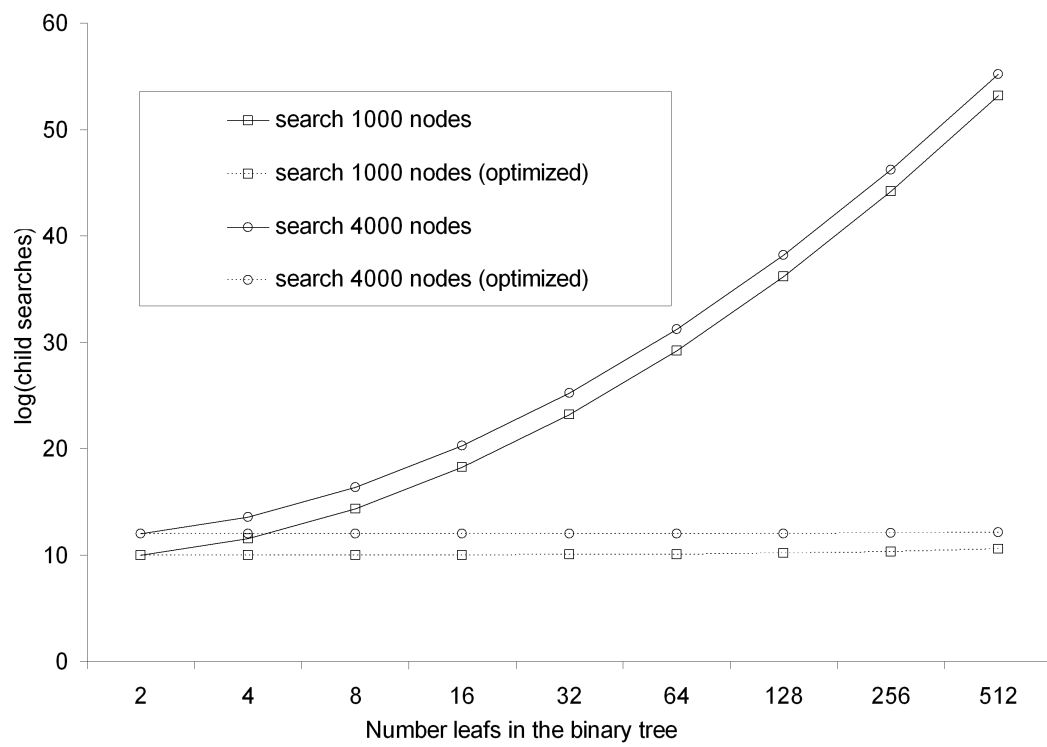


Abbildung 6.8: Comparison of algorithm search space between conventional to optimized algorithm

7 Implementation and Evaluation of the Static Heterogeneous Model

To prove our statements of the preceding chapter we implemented the discussed algorithms using SODA (Service Oriented Database Architecture), which is a novel execution framework for parallel and distributed database operations [10].

Soda builds up on Web-services, which can be plugged together very easily, almost in a LEGOTMlike manner. These services provide the business logic for distributed query execution, specifically supporting autonomous databases in heterogeneous environments. Operations used in this framework – like selection, projection, join, product or sort – can be orchestrated (used, added and removed) dynamically. The architecture is based on the WSRF standard and thus supports stateful Web Services.

Most recently we have seen the emergence of XML, XML storage in DBMS's. The Global Grid Forum (GGF) is producing specifications for both relational and XML databases in Grid environments to be located, accessed, and replicated [75].

7.1 Introduction to SODA

SODA, which means **Service Oriented Database Architecture**, is a proof-of-concept implementation of a middleware for distributed database environments [10]. This middleware provides small Web Services which can be combined, so that database queries which have to collect their data from different sites can be handled. Therefore, SODA makes it possible that database operations can be independent from the databases themselves. Furthermore SODA offers the possibility of adding, removing and using operations dynamically.

This middleware is based on two standards of distributed computing. The first one is the **Web Service** standard. A Web Service gives machines and software components the possibility to interact with each other while they use XML based messages for communication. The network protocol used for that XML-based communication is called SOAP. The second standard is called **SOA** (Service Oriented Architecture) and is actually a "description" or guidance how software should be implemented so that it can be used by other software components. The architecture of SODA is shown in Figure 7.1 and as can be seen it is based on four services, namely the Acquisition Web Service, the Transformation Web Service, the Storage Web Service and the Broker Web Service. All of them interact with each other using the same common interface. That means that the three operation web services and the broker web service can communicate with each other using a simple mechanism.

7.1.1 Components of Soda

The second layer and hence first Operation Web Service layer is the **Acquisition Web Service**, which is the one that interacts with the databases. The databases in turn are in the first layer of the Soda Layered Architecture. So the entirety of all Acquisition Web Services contains all data of the distributed database system. Usually one Acquisition Web Service is connected to one data source. The main task of this layer is to make the data

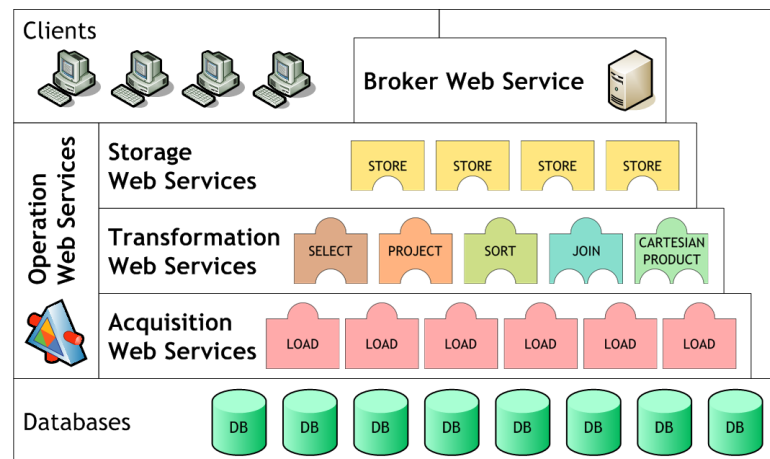


Abbildung 7.1: Soda Layered Architecture

accessible for all other Web Services. That is done by transforming the data in so-called **WebRowSet** documents, which can be accessed by the other services later. The operation which is performed by this layer is the following one:

- **LOAD (in 0, out 1)**: The operation is made up by a process which collects data from existing data sources. The sources can be databases or CSV, respectively other text files. The only requirement is that the Acquisition Service can gain access to the source. This is the only operation in the SODA framework where the database name and the database schema must be defined in the **ServiceParameters**.

Parameters: **TABLENAME** (exactly one)

The third layer is the **Transformation Web Service** layer. This operation web service is responsible for transforming the data. By converting one or more input streams to one or more output streams this layer is able to perform the following operations:

- **SELECT (in 1, out 1)**: This service filters data. It is the same what in a classical relational database a select-operation which compares an attribute with a literal or a number, is (see Chapter 3.3).

Parameters: **ATTRIBUTE** (exactly one), **OPERAND** (exactly one), **LITERAL** (exactly one)

- **PROJECT (in 1, out 1)**: This operation makes it possible to filter out attributes of a data document. The project operation explained in chapter 3.3 has the same purpose like this service.

Parameters: **ATTRIBUTE** (at least one)

- **SORT (in 1, out 1)**: With this operation it is allowed to order datasets while one or more sorting attributes are used. The sorting order can be either descending or ascending.

Parameters: **ATTRIBUTE** (at least one), **ORDER** (zero or more)

- **CARTESIAN PRODUCT (in 2, out 1)**: All rows from one dataset are combined with all rows from a second dataset and as explained in chapter 3.3 the Cartesian Product returns the product out of two datasets.

Parameters are not required

- **JOIN (in 2, out 1)**: Like the natural join explained in chapter 3.3, this service provides the join on two data documents or more specifically on two datasets which are the basis for the data documents.

Parameters: **ATTRIBUTE** (exactly two), **OPERAND** (exactly one)

The results from performed queries are stored within layer four, the **Storage Web Services**. These services just provide one operation which is:

- **STORE (in 1, out 0)**: a data document is stored permanently. Hence, there is no real operation performed here, just the result of a query execution is delivered and stored.

As the first four layers have been explained now, the function of the Broker Web Service must be understood and the way how SODA clients interact with the system. The **SODA Broker Service** is the central component of the framework and the only point of contact for clients [10]. So the requirements for being a SODA client are very low as they have just to communicate with the broker, but do not have to understand the underlying structure. **SODA clients** must therefore use the WSDL [76] service interface to communicate with the broker. The clients can still be implemented using different programming languages as long as they implement the service interface.

The responsibilities of the broker are eventually to coordinate, orchestrate and distribute incoming query requests. Moreover, the broker offers the possibility to register and unregister operation services, which means that this service can make operation services accessible. If there occur mistakes during the query execution, the broker's task is it to detect the failures and to recover them. The global database schema, which is made up by all database schemata of registered data sources is also held by the Broker Service.

7.1.2 Communication in SODA

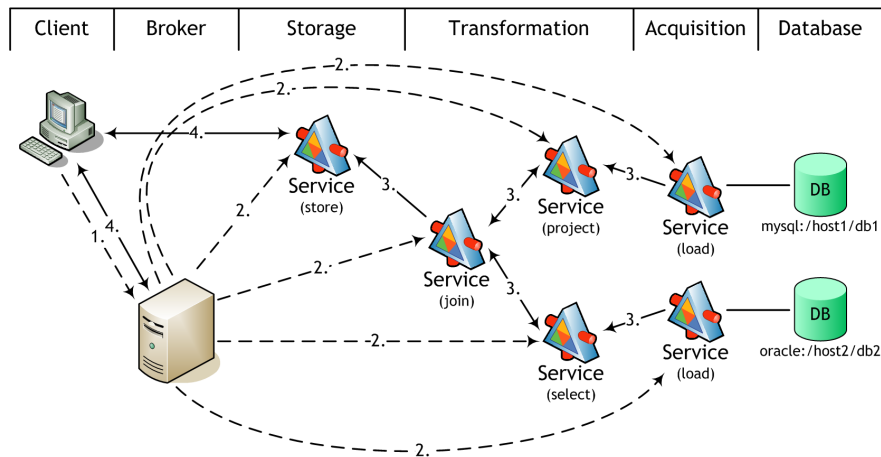


Abbildung 7.2: Soda Layer-Cross Communication

To explain the different services and their tasks better, the communication process will now be discussed. Furthermore the whole process is shown in Figure 7.2. The communication process consists of four steps, namely client request, task distribution, data exchange and result delivery.

Client Request An SQL statement is sent to the broker by a client. This statement can just request databases, that are registered.

Task Distribution This task is one of the broker's main responsibilities. The broker analyzes the query and creates a query execution tree. Based on the execution tree, the different tasks are distributed to the operation services. As already mentioned above,

the communication is based on XML, more specifically the authors of [10] propose an approach using a `RequestDocument` and a `ResponseDocument`.

The subtask which each single operation service has to carry out is defined in the `RequestDocument`. To make it possible that subtasks are uniquely identifiable and that they can be coupled together with a query task, every `RequestDocument` has a `ReqID` and a `SubReqID`. The `ReqID` is unique for every query task. So for every single client request, a new `ReqID` is made. The `SubReqID` is unique for every subtask of the whole query task. Furthermore the `RequestDocument` has information about the predecessors, a list of successors and a parameter section where additional settings for the execution can be provided.

Data Exchange As soon as the single services receive their task from the broker they start processing. Acquisition Services can send the requested data immediately to the next services as they do not get any further inputs. Indeed, the other services have to wait for inputs from their predecessors before they can start processing. The information needed is sent by the above mentioned `ResponseDocuments`. Those documents contain besides the results from predecessor, again the `ReqID` and the `SubReqID`. So the services can put related information together. Moreover, the `ResponseDocuments` contain information about possible errors that emerged. That information can then be used by the broker for error recovery. As some services might send their results in small pieces, there is a Pipeline element where metadata about the process is enclosed.

Result Delivery Eventually the data is stored in the Storage Web Services. The client can then request the data either from the broker or from the Storage Service itself.

7.1.3 Query Execution in SODA

Concerning the query execution, it should be mentioned that the execution of a NATURAL JOIN (see chapter 3.3) is not available at the moment [10]. The currently available query execution operations are the following:

- SELECT
- PROJECT
- SORT
- CARTESIAN PRODUCT
- JOIN

As the broker service is responsible for generating an execution tree before sending the `RequestDocuments`, it must rely on some rules for generating the execution tree. But first of all it is important that the broker knows all databases, tables and attributes so that a suggestive execution tree can be generated. If this prerequisite is met, one of the rules for generating the tree is that the amount of data transmitted between the services should be as small as possible. That means that operations which reduce the data most should be performed first. Such operations are for example projection and selection. Another rule is that if multiple tables are joined, the small tables should be used first. So the broker does not have just to know the tables themselves but also the size of all tables. Those rules and the execution tree generation in general are based on heuristic rules [10].

7.1.4 Performance and Performance Analysis in SODA

For speeding up the query execution process there are actually two different parallelization possibilities. These two possibilities are speedup and scale-up and they were already discussed in chapter 3.4.2.1 and 3.4.2.2. The speedup of a system describes how much the amount of response time is decreased by increasing the parallelism of a system. In other words, speedup describes the gained benefit if for example services are implemented **intra-parallel** on current multi-core processor architectures [10]. It is called linear speedup if the amount of time needed to process a query is inversely proportional to the resources allocated. Note, a linear speedup is hardly achievable. On the other hand there is the possibility to use **inter-parallelism**. That means that the concept of scale-up is used. The scale-up of a system shows how able the system is, to perform larger operations in the same amount of time if more resources are provided. Concerning the SODA there are two possibilities to gain inter-parallelism:

1. multiple services use different Acquisition Services to collect data parallel at different hosts.
2. multiple instances of the same service divide the data amount that has to be processed.

As already mentioned in the Data Exchange part of the SODA communication there might be services which send their data in packets and not the whole data as one document. Therefore, the pipelining approach is used and furthermore if intra-parallel systems are used there is the possibility of sending through an applicable number of pipes. Those pipes can then send the data with different speed and so intra-parallel systems can divide their datasets into blocks and operate on these blocks separately.

In Beran et.al. [10] a performance analysis for justifying the usage of SODA in a Web based environment was carried out. For defining the speedup of a query execution the change of time used for one specific query execution if computing nodes were added was analyzed. The approach for executing the query was the following: First the data was read from the disk, then the data was distributed for the Quicksort-Phase. In this phase a high benefit due to intra-parallelism can be achieved. After the Quicksort-phase the partly sorted data are collected again using a merge-sort. Finally the data was stored in the last step. The results are shown in Figure 7.3. As the Figure shows the speedup is not linear but still very remarkable. The disk processing cost is very low, but the processing and network costs are very high. Concerning the processing cost the reason for their size is the excessive packing and unpacking of the XML data format [10].

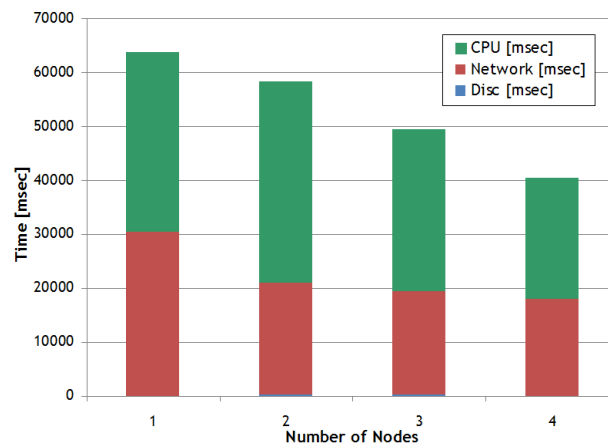


Abbildung 7.3: Speedup numbers

For analyzing the scale-up, the impact of increasing workload on the runtime was tested. The results of this test are shown in Figure 7.4 and here the scalability shows a slightly over-linear behavior which is again connected to the high processing cost which are related to the packing and unpacking of the XML format. These scale-up and speedup analyse show a very good runtime behavior and therefore a certain degree of scalability can be seen. The issue with the high-performance cost could be solved if the data exchange format between compatible nodes could be more efficient [10].

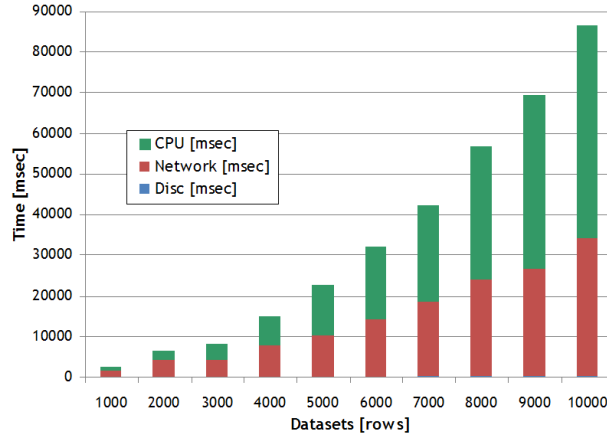


Abbildung 7.4: Scale-up numbers

In conclusion SODA is a very flexible architecture as operators can be easily added, removed or exchanged. Furthermore due to the layer architecture, the Acquisition layer is able to make various kinds of databases and datasheets available. At the moment just basic operations are available but these operations can be extended and moreover operators which benefit from inter/intra parallelism could be implemented.

7.2 Prototype

We focus in the following proof on the Binary Merge Sort algorithm, which is the central point of interest for the orchestration of the Grid workflow (see Algorithm 4). The performance analysis by the presented analytical model shows that a smart orchestration of the available nodes with heterogenous performance characteristics, results in an increased performance behaviour, as depicted in Figure 5.7 and Figure 5.2 by the speedup numbers for the modified parallel merge sort algorithm. This effect influences also the performance of all other parallel database operations on the merge sort algorithm dependent.

For our practical implementation we used a heterogenous hardware infrastructure consisting of a mix of three different blade systems (different number of cores and different clockrate of processors) each running a Scientific Linux. The configuration of the nodes for the Parallel Binary Merge Sort is seen in Figure 7.5 and the hardware configuration is listed in table 7.1. As described in the previous section, the workflow is executed in two configurations: using 4 nodes and 8 nodes. **Cha1**, **cha2** and **cha3** are the names of the physical nodes. The bar with the "throttling" tag shows where the network connection speed has been altered. We have done runs with different network speeds: 1GBit, 50MBit, 5MBit and 512kBit. We have also done multiple passes of the same configuration to avoid unexpected differences in the duration time of the workflow. This configuration reflects the unmodified implementation of the parallel binary merge sort. In Figure 7.6 the modified version is depicted. That means the last 3 nodes run with 1 GBit/s and the other nodes running slower.

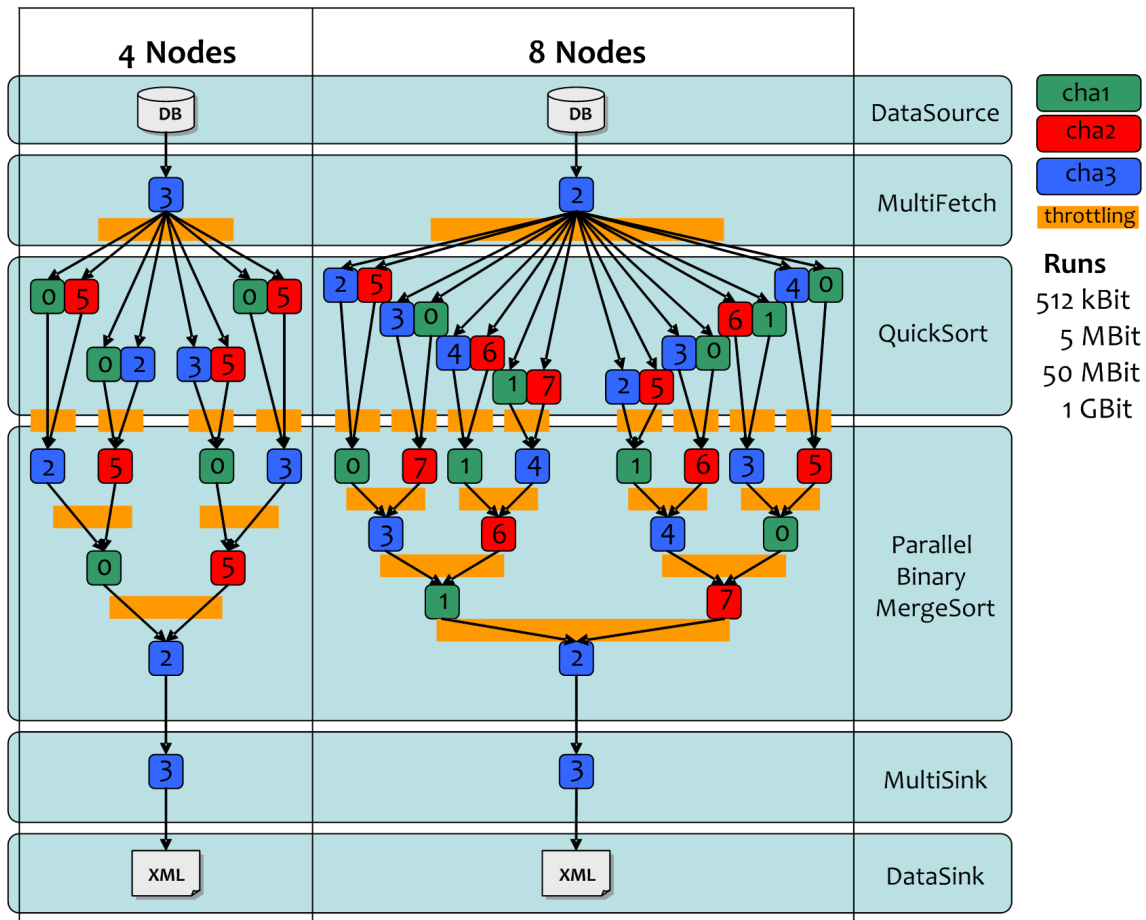


Abbildung 7.5: Parallel Binary Merge Sort Workflow Configuration

Tabelle 7.1: Blade Server Infrastructure for Performance Measurement

Name	Processor	Speed	Memory	Storage
cha1	1x Intel Core 2 Duo 2 cores (E 5130)	2.0 GHz	8.0 GB	300 GB
cha2	2x Intel Core 2 Duo 4 cores (E 5130)	2.0 GHz	8.0 GB	300 GB
cha3	1x Intel Core 2 Quad 4 cores (E 5335)	2.0 GHz	8.0 GB	300 GB

7.3 Performance Evaluation

According to our presented orchestration algorithm we placed, simply said, the postoptimal phase of the merge sort algorithm on the node with the highest performance and best bandwidth connection. We did several scenarios of orchestration by varying the different performance parameter values of the nodes, as pure processing power, disk performance and network bandwidth. The practical results justified our analysis that the most influencing factors are the processor performance and the network bandwidth. Figure 7.7 shows the actual execution times of the parallel merge sort algorithm for a conventional scenario and the modified one according to our orchestration algorithm. The expected performance improvement is easily recognizable and justifies our analytical findings. The benchmark configuration was 8 nodes all running with 512kBit network speed in the unmodified version and in the modified version the last 3 nodes running 1GBit network speed instead of

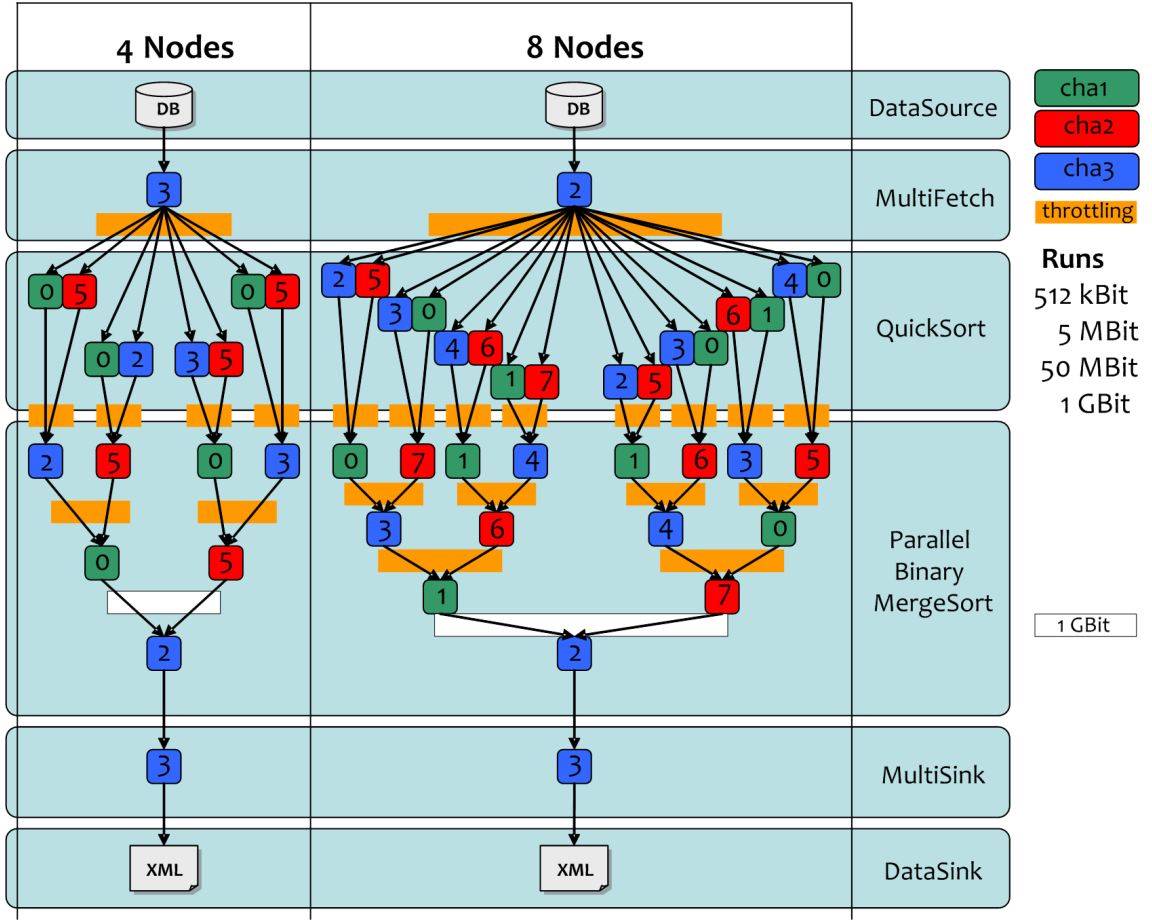


Abbildung 7.6: Modified Parallel Binary Merge Sort Workflow Configuration

512kBit. The regression coefficient between real measured time and calculated with our model is

$$r_{unmodified} = 0.985$$

for the unmodified algorithms, and respectively

$$r_{modified} = 0.983$$

for the modified algorithms.

That means the real measured time values are very close to our model and the behavior of the real measured values to that of the model are identical. Please note that the values with smaller amount of data have more deviation than the values for higher amount of data. The reason is, that in a service orientated architecture latencies can be occur and therefore for lower amount of data the model is not accurate enough.

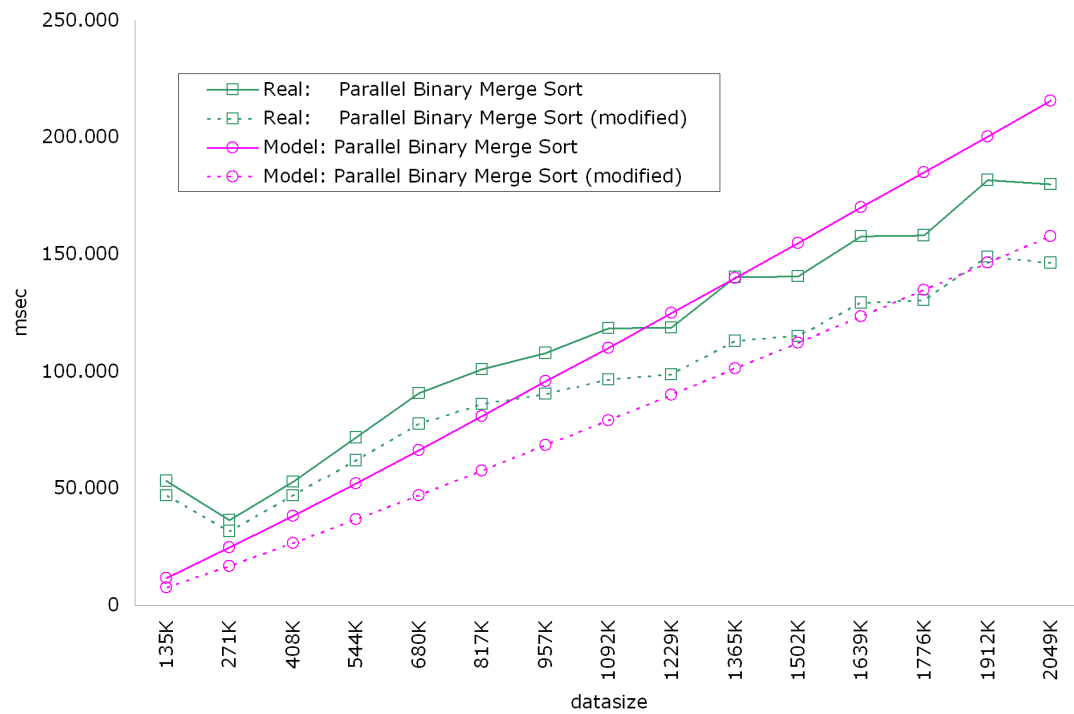


Abbildung 7.7: Real and Model performance behavior of Parallel Merge Sort, unmodified and modified

8 Extensions of the Static Heterogeneous Model

In this chapter we describe some ideas for possible extensions of our developed Static Heterogeneous Model. This extensions can be used to make it easier to implement a broker application and to take into account changes during execution of a workflow, because nodes can fail during execution and it is necessary to rearrange the node configuration dynamically. We have identified following extensions:

- Using Performance Indices to determine the optimal node configuration.
- Adding Reliability to the node characteristics.
- Dynamic Optimization of Workflows.

8.1 Performance Indices

Performance Indices for heterogeneous systems are well described in Kalinka et. al. [77] and load indices in [78]. Kalinka et. al. present two performance indices **PIV** - Performance Index Vector and a weighted index **WPIV** - Weighted Performance Index Vector. **PIV** and **WPIV** based on a Euclidian metric. These new developed performance indices use vectors to describe the load of the system or machine and take care of the characteristics of the load. In this section we give a short introduction to their definitions.

Kalinka et. al. define four basic resources to be analyzed in a machine **CPU**, **Memory**, **Disk** and **Network** and is obtained in equation 8.1.

$$ID = f(I_{CPU}, I_{Memory}, I_{Disk}, I_{Network}) \quad (8.1)$$

where ID is the performance index of a specific machine. For each of the given indices a weight W is also defined. These weights depend on the characteristics of the application to be scheduled.

The indices I_{CPU} , I_{Memory} , I_{Disk} and $I_{Network}$ are combinations of indices that describe applications to be more strictly bounded to their respective resource.

That means, I_{CPU} is a combination of the CPU indices describing applications to be more strictly CPU-Bound, I_{Memory} is a combination of the Memory indices describing applications to be more strictly Memory-Bound, and so on.

Each of the resource indices is calculated independently. The measured values cannot be directly combined and compared, and therefore a normalization should be used. Thus, each measured value is normalized separately. The specific index of CPU, Disk, Memory and Network resources has its value between 0 and 1. The normalization gives the ability to compare the indices among different machines and the values of each index can be added together.

Performance indices and load indices must be able to estimate the future for the decision putting a load on a specific machine. This estimation is based on current and past values. Kalinka et. al. define a performance index as the metric that can provide an image of the work capacity, or in other words, illustrate what can be expected. The load index can be defined as a non-numerical variable, zero if the resource is idle and positively when the load of the resource increases [79].

Wolffe et al. [80] propose the use of Load Capacity as a load index for heterogeneous environments. The Load Capacity is the effective use of the processor. The Load Capacity is defined:

$$(1 - CPUutilization) * RelativeCPU Speed \quad (8.2)$$

where *RelativeCPU Speed* is measured against some common reference processors.

In equation 8.1 it is possible using a specific load index for each resource, where each of them may be seen as a vector base. That means every resource can be represented as vector. e.g. $\langle 1, 0, 0, 0 \rangle$ represents an application that is 100% CPU bounded. $\langle 0, 1, 0, 0 \rangle$ is a 100% Memory application, $\langle 0, 0, 1, 0 \rangle$ 100% Disk and $\langle 0, 0, 0, 1 \rangle$ 100% Network application. The resources n that a machine can provide may be considered to form an n dimensional space. Where n is the number of resources. A point in this n dimensional space represent the current state of the machine. The point for an idle machine is located in $\langle 0, 0, 0, 0 \rangle$ and a completely overloaded machine's point is located in the opposite vertex $\langle 1, 1, 1, 1 \rangle$.

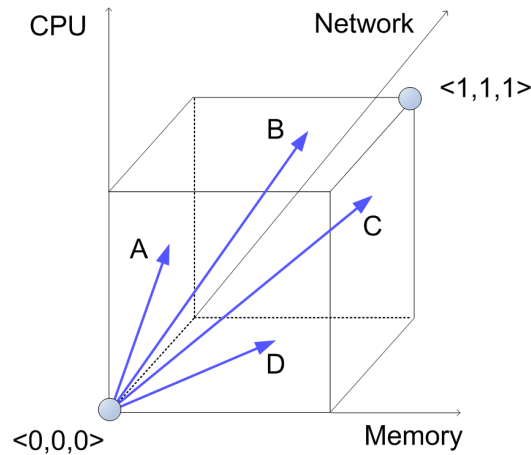


Abbildung 8.1: Three Dimensional Index

For instance, Figure 8.1 presents a three-dimensional space. We used three dimensions because for dimensions $n > 3$, the visualization became very complex. It shows the situation for four points of load of a particular machine. Vector A represents a machine with following load :

- great use of CPU
- no use of memory
- average use of network

Vector B shows a machine with:

- no use of CPU
- average use of memory
- average use of network

Similar, Vector C represent a situation for a machine with:

- great use of CPU
- great use of memory
- great use of network

The representation as vectors in an 3-dimensional space have two kinds of information.

1. length of the vector
2. angle between vector and x-axis

The length represents how much of each resource is used and the angle between shows the relative percentage of each resource used.

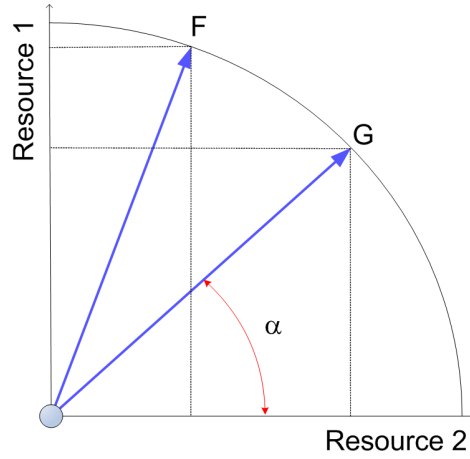


Abbildung 8.2: Two Machine Resource Vectors

As depicted in Figure 8.2 Load Balancing can be observed through the angle α of the vector. In general we can distinguish between following situations:

1. When $\alpha \approx 45^\circ$, both resources are equally loaded.
2. When $\alpha \gg 45^\circ$, it indicates that the Resource 1 is predominant.
3. When $\alpha \ll 45^\circ$, it indicates that the Resource 2 is the predominant one.

Machine G is balanced due situation 1 ($|45^\circ - \alpha| = 0$) concerning resource 1 and resource 2, machine F is less overloaded than G regarding resource 2, that is because close due situation 2. If $\alpha \approx 0$ and length tends to 1, then resource 2 is close to saturation and in the same way if $\alpha \approx 90^\circ$ and length tends to 1, then resource 1 is close to saturation. In these cases we notice overload conditions. Therefore for overload conditions we had to check the angle and the length of the vector.

8.1.1 One Dimensional Result Index

In Figure 8.3 we have depicted two different machines (**Machine 1** and **Machine 2**) with different load index but equally loaded. At both machines a new process (**Process**) is arriving. The new process uses only Resource 1, that means the process is Resource 1 bound. In Machine 1 Resource 1 is more loaded and in Machine 2 Resource 2 is more loaded. The new process can be allocated in Machine 1 and in Machine 2. It should be determined in which situation a better result is obtained. The length of the vector, the Euclidian distance, can be used as metric for this decision. Therefore Equation 8.1 can be written again as

$$ID = \sqrt{I_{CPU}^2 + I_{Memory}^2 + I_{Disk}^2 + I_{Network}^2} \quad (8.3)$$

Allocating the new process to Machine 1 and Machine 2 lead us to the result vectors Result 1 and Result 2. Result 1 is lower than Result 2 and therefore the process is allocated in

Machine 2. If we have two equally loaded machines and want to put a load on one of these these indices lead us to a better allocation of the load.

Kalinka et. al. proposed this metric as load identification by resource.

The example Figure 8.3 illustrates an example that uses processes with only 1 resource.

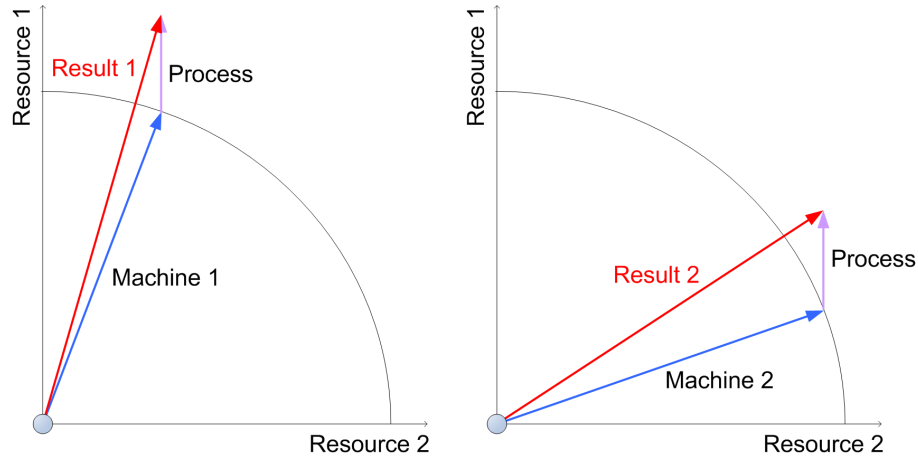


Abbildung 8.3: Process arrives on Machine 1 and Machine 2

8.1.2 Two Dimensional Result Index

Figure 8.4 shows a more complex example for applications (processes) that uses 2 resources. We use again two different machines Machine 1 and Machine 2 that are equally loaded. As in Figure 8.3 Resource 1 is more loaded in the Machine 1 while Resource 2 is more loaded in the Machine 2 in terms of use.

It should be determined in which machine a better result is obtained, and as in the previous example the resulting vector is used for that decision. The shorter result vector indicates which machine should be used to allocate the process. In this case the shorter result vector is in machine 2, therefore the process should be allocated there so that a better performance can be obtained.

The performance index presented from Kalinka et. al. bases itself on the Euclidian distance. The Euclidian distance is the distance between the origin point (0,0,0) where the machine is idle and the resulting vector of the two vectors: the load vector before receiving an new application and the load vector of the new application.

The machine with the shortest Euclidian result distance should be chosen to deploy the application or load.

Kalinka et. al. show also with modeling techniques and simulation that their proposed performance index **PIV** leads to a performance increase. They used the **AMIGO** (dyn**AM**ical flex**I**ble schedulin**G** enviro**N**ment) environment and several load indices execute tests were accomplished. **PIV** can present better results for the cases in which there are some knowledge from the kind of application or process.

8.1.3 Using Indices in a Static Heterogeneous Model

As seen in chapter 5 we have following node characteristics defined in our model for heterogeneous environments:

- Processing Power.
- Disk Speed.

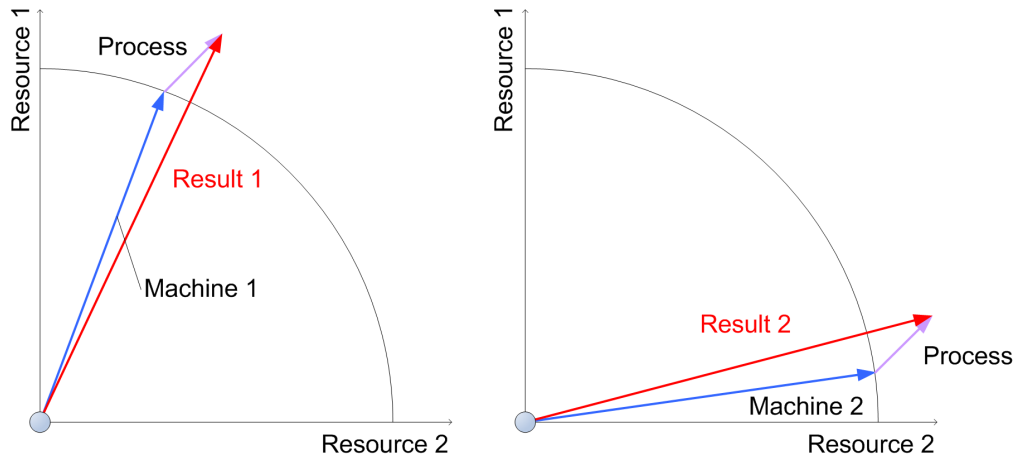


Abbildung 8.4: Process with 2 resources arrives on Machine 1 and Machine 2

- Network Bandwidth.

Now we can define a three-dimensional vector as described in the previous sections for the representation of the resources in our model. This three-dimensional vector can be used to find the optimal solution for the parallel Binary Merge Sort as described in section 6. The binary tree can be built upon the calculated performance index. It must be investigated if using performance indices to build the binary tree, the optimal solution could be found. This is a difficult and sophisticated problem, even possible np-hard, because all the combinations of nodes used in the build binary tree should be used.

8.2 Reliability Extension

An ad hoc extension for the consideration of reliability in our Static Heterogeneous Model can be done by using a fourth parameter in the model, the **reliability** of a node. The reliability can also be used in performance indices to choose the "best" node.

For the clarification of reliability we give a short introduction. One of the first definitions of reliability is that from Edward P. Coleman introduced in Techniques for reliability [81] 1957: "Reliability is the probability of a system performing its purpose adequately for the period of time intended under the environmental conditions encountered."

Reliability in computer systems plays an important role, a very early investigation of reliability of vacuum tubes for storing information has been introduced by E.B. Ferrell in his work *Reliability and its relation to suitability and predictability* 1953 [82]. Ferrell made live test inspections on vacuum tubes to control the manufacturing process of these tubes. His goal was to make the live time of the produced tubes predictable.

There exist also other definitions of reliability, here are some examples:

- Military standard definition of reliability : "The probability that an item will perform a required function without failure under stated conditions for a stated period of time."
- Engineering definition: "The probability that a component part, equipment, or system will satisfactorily perform its intended function under given circumstances, such as environmental conditions, limitations as to operating time, and frequency and thoroughness of maintenance for a specified period of time."
- Reliability definition from NASA (National Aeronautic and Space Administration): "Reliability is the probability of a device performing adequately for the period of time intended under the operating conditions encountered."

To define reliability it is necessary to define the following constraints:

1. Reliability is a probability. This means that failure is regarded as a random phenomenon.
2. Reliability is predicated on "intended function". That means operation without failure.
3. Reliability applies to a specified period of time, that means that a system has a probability to operate without failure within time t .
4. Reliability is restricted to operation under stated conditions.

Examples for reliability:

- Less than two hours of downtime in five years.
- A Mean Time Between Failures (MTBF) of at least 3000 hours.

As a mathematical description, see also Figure 8.5: The Reliability $R(t)$ is the probability of a system not failing during the period $[0, t]$. Where $F(t)$ is the failure distribution function, $R(t) = 1 - F(t)$ is the reliability and $f(t)$ is the failure density function.

$$R(t_1) = 1 - \int_0^{t_1} f(t)dt = \int_{t_1}^{\infty} f(t)dt$$

Note availability is not equal to reliability. Availability is the proportion of time a system is in a functioning condition and in its simplest form is as a ratio of the expected value of the uptime of a system to the aggregate of the expected values of up and down time.

$$A = \frac{E(Uptime)}{E(Uptime) + E(Downtime)}$$

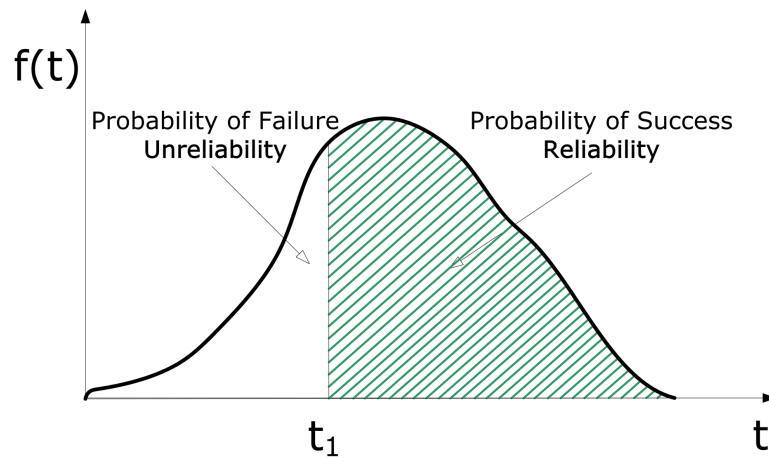


Abbildung 8.5: Time-to-Failure (Random-Function)

The probability distribution function curve can take many forms. Some of the different distributions are listed below.

Normal Distribution

Representing random events is the normal curve or gaussian curve.

$$f(t) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(t-\mu)^2/2\sigma^2}$$

where μ = the Mean and σ = the standard deviation.

Lognormal Distribution

The lognormal distribution is used for general reliability analysis. To test failures in material strengths and loading.

Weibull Distribution

The Weibull distribution function is a generalization of an exponential distribution and is a general purpose distribution. W. Weibull introduced this statistical distribution for wide applicability first 1939 for material strength and 1951 more general [83]. It can be used for:

- Yield strength of steel
- Size distribution of fly ash
- Fiber strength of cotton
- system reliability
- failure of electronic components

An example for using the Weibull distribution function is e.g. testing the archival performance of digital magnetic tapes [84].

$$f(t) = \left(\frac{\beta}{\eta}\right) \left(\frac{t-\gamma}{\eta}\right)^{\beta-1} e^{-\left(\frac{t}{\eta}\right)^\beta}$$

where β = shape parameter, γ = location parameter and η = scale parameter. The β is the shape parameter. β gives indications on the failure modes e.g. old age or wear out.

Exponential Distribution

The exponential distribution is a simple distribution which can be used if units have constant failure rate. The exponential distribution is simple but this simplicity can lead to use it in inappropriate situations. A good introduction for using the exponential distribution or the Weibull distribution is the very new study of K. Das [85]. Das studied the two distributions for machine reliability in a cellular manufacturing system and handled different failure characteristics for maximizing the system reliability.

$$f(t) = \lambda \exp^{-\lambda(t-\gamma)}$$

where λ = scaling factor or failure rate and γ = location factor.

8.2.1 Node Reliability

For our model extension we define that the reliability of the nodes and their representation as a graph is known by the broker as all other node characteristics, therefore we can use

this information to optimize the workflow. We can extend our model defined in chapter 5 with a fourth parameter, the reliability of a node. Please note that not the reliability of the Database System is used here, we use only the reliability of a the involved nodes. It is necessary to define the reliability of a processing node, because this is the element used in the execution of the workflow by arranging the nodes. Then we can extend the node characteristics defined in our model for heterogeneous environments with the reliability of a node:

- Processing Power.
- Disk Speed.
- Network Bandwidth.
- **Reliability.**

The node is a system containing electronic components and therefore a Weibull distribution with a "bath-tube" shaped curve is suitable. The "bath-tube" means at the beginning of the lifetime there are more failures, then the failure rate is relatively constant and at the end of the lifetime the failure rate increases again. This kind of distribution is typical to electronic systems.

As easily can be imagined arranging the nodes to build a perfect binary, it is necessary to have the most reliable nodes at the root of the tree. The reason is clear, if a node fails during execution of the parallel binary merge sort workflow the cost of this failure increases during execution of the phases. As seen in Figure 6.6 the last processor and their two predecessors must be the most reliable nodes.

As described in section 8.1 the reliability can be also used to calculate a performance index. Therefore Equation 8.3 can be rewritten as

$$ID = \sqrt{I_{CPU}^2 + I_{Memory}^2 + I_{Disk}^2 + I_{Network}^2 + I_{Reliability}^2} \quad (8.4)$$

Analogous the reliability extension as a parameter in an performance index to find the optimal solution must be investigated. Please note that the reliability extension influences the duration time of the parallel merge sort in that way, that we have to weight the most influencing parameter, the network bandwidth, with the reliability of the node. An example can be seen in table 8.1, where *Node4* would be chosen without the reliability extension, but if we used the reliability as additional parameter we would choose *Node1* as one of the last three nodes. Note using the reliability in this manner is only valid if the time of transferring data between the nodes is for all the three nodes the same, because the reliability varies from level to level of the perfect binary tree.

8.2.2 Database Reliability

In the previous section we introduce an approach for finding the optimal solution for the execution of the workflow in a parallel binary merge sort algorithm using the reliability of the involved nodes. In the field of database systems the term reliability is used to improve the reliability of the whole database system [86] or the reliability of queries [87]. Sadri [87] studies the problem of determining the reliability of answers to queries in a relational

Tabelle 8.1: *Node Reliability Extension*

Node Number	Network Bandwidth [Mbit/s]	Reliability	Weighted Network Bandwidth [MBit/s]
Node1	90	0.95	85.5
Node2	50	0.98	49.0
Node3	10	0.99	9.9
Node4	100	0.83	83.0
Node5	50	0.78	39.0

database system, where the information in the database comes from various sources with varying degrees of reliability. Sadri extends the relational model with an "information source vector". This also enables the database system to calculate the **reliability** of each tuple in the answer to a query as a function of the radiabilities of information sources.

8.2.3 Grid Reliability

In Dai et. al. [88] the two kinds of reliability in Grid systems have been investigated:

- Grid program reliability: grid program reliability is defined as the probability of successful execution of a given program running on multiple Virtual Nodes (VNs) and exchanging of information through Virtual Links (VLs) with the remote resources of other VNs, under the environment of grid computing system.
- Grid service reliability: grid service reliability is defined as the probability for all of the programs involved in the considered grid service to be executed successfully.

The failure process of either VN or VL can be modeled as a Poisson process, this assumption can be justified by the operational phase in which the software and hardware are not changed. This justification can be seen in e.g. Yang and Xie [89].

Different from software reliability, the grid program reliability actually involves the hardware reliability including the failures on VNs and VLs though this reflects the probability of successfully running the program.

Grid services reliability can be computed with the graphic theory using the concept of Minimal Resource Spanning Tree (MRST). This is similar to our graphical representation of the network using an adapted minimal spanning tree algorithm for finding the optimal workflow orchestration.

8.3 Performance Metrics for Grid Workflows

Performance metrics for Grid workflows are developed in Truong et. al. [90]. They introduce an ontology for describing performance data of Grid workflows and illustrate how the ontology can be utilized for monitoring and analyzing the performance of Grid workflows. They classify performance metrics according to five levels of abstraction, including, from lower to higher level, see Figure 8.6.

1. code region
2. invoked application
3. activity
4. workflow region
5. workflow

The metrics are categorized into: **execution time, counter, data movement, synchronization, ratio and temporal overhead.**

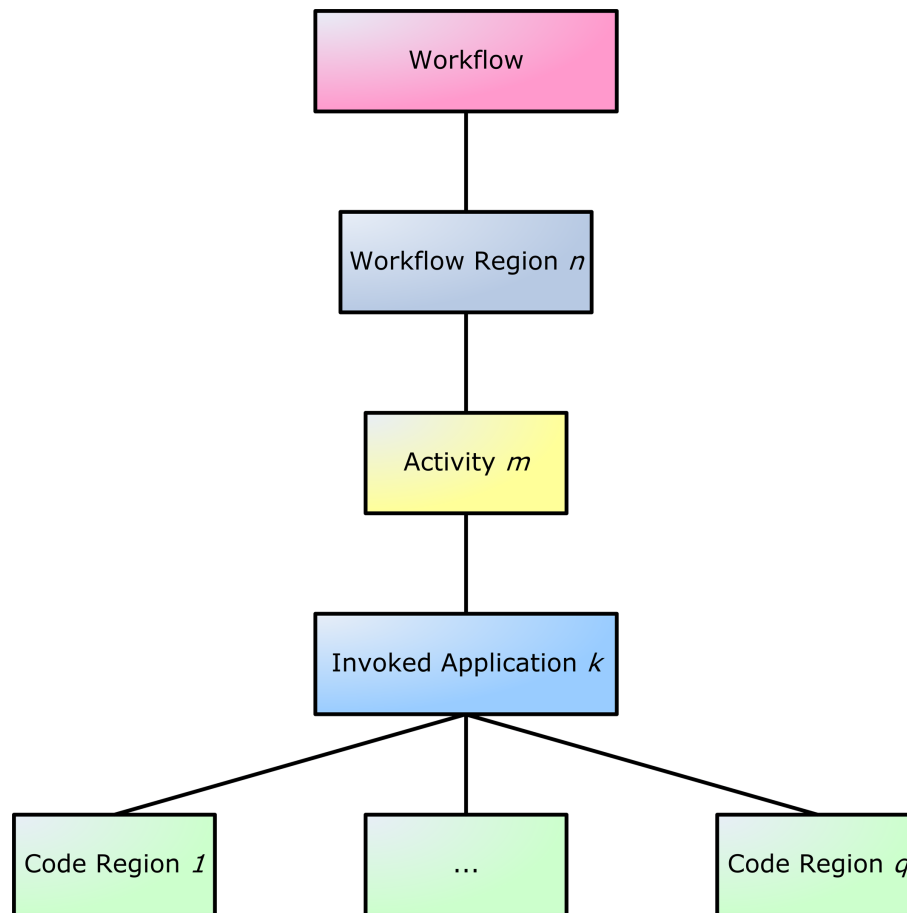


Abbildung 8.6: *Hierarchical structure view of a Workflow*

An example of Performance Metrics at Code Region is shown in table 8.2. Most metrics can be constructed from metrics at code region level, and most existing conventional performance tools provide these metrics. Existing workflow monitoring and analysis tools normally do not. The challenging issue is to integrate conventional performance monitoring tools into workflow monitoring tools.

Truong et.al. develop an ontology named **WfPerfOnto** (Ontology describing Performance data of Grid Workflows) for describing performance data of workflows. **WfPerfOnto** is based on OWL [91]. The Web Ontology Language OWL is a semantic markup language for publishing and sharing ontologies on the World Wide Web. OWL is developed as a

Tabelle 8.2: *Performance Metrics at Code Region Level*

Category	Metric
Execution time	ElapsedTime, UserCPUTime, SystemCPUTime, SerialTime, EncodingTime
Counter	L2-TCM, L2-TCA (hardware counters) NCalls, NSubs, RecvMsgCount, SendMsgCount
Synchronization	CondSynTime, ExclSynTime
Data Movement	TotalCommTime, TotalTransSize
Ratio	MeanElapsedTime, CommPerComp, MeanTransRate, MeanTranSize CachMissRatio, MFLOPS, etc.
Temporal overhead	temporal overhead of parallel code regions

vocabulary extension of RDF (the Resource Description Framework) and is derived from the DAML+OIL Web Ontology Language. The visualization of the described performance data of a workflow has been done in *Protege* [92]. Different monitoring and analysis tools can store/export performance data in/to ontological representation to use for high-level search and retrieval of performance data. The ontology **WfPerfOnto** can be used for:

- Knowledge base performance data of Grid workflows.
- Utilized by high-level tools such as schedulers, workflow composition tools.
- Re(discover) workflow patterns, interactions in workflows, to check correct execution.
- Distributed Performance Analysis.
- Performance analysis requests can be built based on **WfPerfOnto**.

8.4 Dynamic Optimization of Workflows

We have developed a static model for the execution of parallel database operations in heterogeneous environments, but in the real world there are not only static parameters. In this section we give some ideas to extend our model to consider dynamic aspects. Workflow optimization in Grid environments regarding dynamically changing resources and conditions are well described in [93].

Following events can happen during the execution of a workflow:

- One or more nodes fail.
- One or more nodes exceed their workload.
- Nodes with higher performance are available.
- The algorithm itself needs more nodes (resources).

In case 1, case 2 and case 3 new nodes have to be added and rearranged. Case 4 is for special applications like the calculation of fluid dynamics in astrophysics [94]. These applications are difficult to adapt in the current distributed processing model (such as the Grid) because of a lack of interface for them to directly communicate with the runtime

system and the delay of resource allocation. For this particular case an Application Agent (AA) embedded between the application and the underlying conventional Grid middleware is necessary, see [95]. For the execution of the presented parallel database operations no dynamic resource allocation is necessary, therefore we neglect this case in our extended model.

The simplest reaction to case 1, case 2 and case 3 is to abandon the whole workflow and start it again with the new nodes involved. This can be useful for small datasets, but for huge datasets this is not a clever strategy. A more sophisticated strategy should be used. Depending on the consumption of the cost of the workflow and the cost for the rearrangement of the nodes on the opposite should be compared. And therefore a restart of the whole workflow is cheaper in terms of duration time.

One of the premises for rearranging nodes is to have synchronizing points or checkpoints, where temporary results are written to disk for restarting reasons. Another requirement for rearranging nodes during the execution of a workflow is to have an Application Agent as introduced in [95] to have direct control to the resources as described in [96] for providing resource management services to parallel applications.

The dynamic extension of our model should have at least these features:

- Resource management services to parallel applications (or an Application Agent).
- Supporting checkpoints.
- A continuous progress check must be supported.

9 Conclusion

In this work we have shown that heterogeneous computing environments are suitable for the implementation of parallel database operations and can be optimized for such an environment. Only a few characteristic parameters are necessary to describe the performance of heterogeneous nodes, and based on these characteristics an optimal workflow execution algorithm can be found. We have developed a new optimization algorithm to find the optimal arrangement, in terms of processing time, of the involved nodes. The justification of this work has been done by an implementation of the unmodified and modified version of the algorithms. The results have shown the gain on performance when using the modified parallel database algorithms. We have also developed a heuristic solution for finding the optimal node configuration for the execution of specific parallel operators. This algorithm can be used for the implementation of a query optimizer in heterogeneous environments.

Our findings can be used for the implementation of database systems based on heterogeneous environments for applications in scientific computing, such as bioinformatics, fluid dynamics and high energy physics (HEP). Such applications are handling huge datasets and heterogeneous environments are cheaper than specialized parallel database machines. Therefore there is an urgent need for an optimized data management in heterogeneous environments.

Abbildungsverzeichnis

2.1	The relation CUSTOMER	14
2.2	The relation ORDER with the primary key: <i>OrderID</i>	15
2.3	Schema diagram for the relations CUSTOMER, ORDER and EMPLOYEE	16
2.4	Result of $\sigma_{Country="Germany"}(CUSTOMER)$	17
2.5	Result of $\pi_{City,Country}(CUSTOMER)$	18
2.6	Result of $\sigma_{CustomerID=0002}(CUSTOMER \times ORDER)$	21
2.7	Result of $\sigma_{Customer.CustomerID=Order.CustomerID}$	21
2.8	Result of projection and division operation	22
2.9	Result of $ShipCountry \mathcal{G}_{count(OrderID)}(ORDERS)$	23
3.1	Two-tier Architecture	25
3.2	Three-tier Architecture	27
3.3	Account Relation	29
3.4	Example Horizontal Fragmentation	30
3.5	Example Vertical Fragmentation	30
3.6	Pipeline- and Partitioned Parallelism	32
3.7	Shared-Nothing Architecture	32
3.8	Shared-Memory Architecture	33
3.9	Shared-Disk Architecture	33
3.10	Linear, sublinear and superlinear speedup	34
3.11	Linear, sublinear and superlinear scale-up	35
3.12	Generalized Multiprocessor Organization	36
3.13	Schema of the Evolution of the EGEE Grid Infrastructure	39
3.14	Static Simplified Grid Organization	40
3.15	Data Integration Architecture of PDBS	41
3.16	(a) the distribution of a global request in a usual DDBS. (b) the distribution of a request in a PDBS.	41
4.1	Revised Static Simplified Grid Organization	47
4.2	Percentage of C_p^2 cost in a Revised Static Simplified Grid Organization	48
4.3	Prepare and Suboptimal Phase	49
4.4	Optimal phase	50
4.5	(a) operator with two inputs and two outputs. (b) same comparator, just drawn as vertical line	51
4.6	(a) The comparison network, which is in fact a sorting network, at time 0. (b) comparison network at time 1. (c) comparison network at time 2. (d) comparison network at time 3	51
4.7	two comparison networks with depth 1, called half-cleaner.	52

4.8	There are four different cases where the midpoint of a bitonic zero-one sequence can fall. (a)-(b) the division occurs in the middle subsequence of 1's. (c)-(d) the division occurs in one of the "0" subsequences. The outputs consist always of two bitonic parts and one part is always clean. Furthermore every output element of the top half is always at least as small as every output element of the bottom half.	53
4.9	The bitonic sorting network[n] for $n = 8$ is shown. (a) The recursive construction of the sorting network. (b) The network showing the details and the progressively smaller half-cleaners that sort the input eventually.	54
4.10	Bitonic Sort	55
4.11	Perfect Shuffle	55
4.12	Phases of Query Processing	64
4.13	Operator Tree	64
5.1	Sort in a Generalized Multiprocessor Organization	68
5.2	Sort in a Simplified Grid Organization	69
5.3	Join in a Simplified Grid Organization with $S=0.001$	70
5.4	Join in a Simplified Grid Organization with $S=0.5$	70
5.5	Aggregate in a Simplified Grid Organization	71
5.6	Sort Speedup in a Generalized Multiprocessor Organization	71
5.7	Sort Speedup in a Simplified Grid Organization	72
5.8	Sort Scale-up in a Simplified Grid Organization	73
5.9	Join in a Generalized Multiprocessor Organization with $S=0.001$	73
5.10	Join Speedup in a Generalized Multiprocessor Organization with $S=0.001$	74
5.11	Join Scale-up in a Generalized Multiprocessor Organization with $S=0.001$	75
5.12	Join Speedup in a Simplified Grid Organization with $S=0.001$	76
5.13	Join Scale-up in a Simplified Grid Organization with $S=0.001$	76
5.14	Aggregate in a Generalized Multiprocessor Organization	77
5.15	Aggregate Speedup in a Generalized Multiprocessor Organization	78
5.16	Aggregate Scale-up in a Generalized Multiprocessor Organization	78
5.17	Aggregate Speedup in a Simplified Grid Organization	79
5.18	Aggregate Scale-up in a Simplified Grid Organization	79
6.1	A Weighted Undirected Graph	82
6.2	Perfect Binary Trees of height $h = 0, 1, 2, 3, 4$	82
6.3	Perfect Binary Tree Searcher Test Suite	85
6.4	1000 nodes with degree $m=2$	86
6.5	1000 nodes with degree $m=4$	87
6.6	Contribution to execution time of last three processors	87
6.7	Performance influence of partone and parttwo combinations	88
6.8	Comparison conventional to optimized	89
7.1	Soda Layered Architecture	92
7.2	Soda Layer-Cross Communication	93
7.3	Speedup numbers	95
7.4	Scale-up numbers	96
7.5	Parallel Binary Merge Sort Workflow Configuration	97
7.6	Modified Parallel Binary Merge Sort Workflow Configuration	98
7.7	Real and Model performance behavior of Parallel Merge Sort, unmodified and modified	99
8.1	Three Dimensional Index	102
8.2	Two Machine Resource Vectors	103

8.3	Process arrives on Machine 1 and Machine 2	104
8.4	Process with 2 resources arrives on Machine 1 and Machine 2	105
8.5	Time-to-Failure (Random-Function)	106
8.6	Hierarchical structure view of a Workflow	110

Tabellenverzeichnis

4.1	Common Disk Parameters	47
4.2	Average Cost of Network and Disk	48
4.3	Aggregate Functions	60
4.4	Employee Relation	60
5.1	Cost Values for Analysis	67
6.1	Number of Binary Trees found in generated Graphs	85
7.1	Blade Server Infrastructure for Performance Measurement	97
8.1	Node Reliability Extension	109
8.2	Performance Metrics at Code Region Level	111

Statements of Thesis

1. *"Heterogeneous computing environments are suitable for parallel database operations."*
2. *"The heterogeneity of computing environments can be used to optimize the execution of parallel database workflows on it."*
3. *"The network bandwidth has the most influence on the performance of the execution of parallel database operations in heterogeneous environments."*
4. *"Classic performance evaluations on general multiprocessor environments are reversed or invalidated with modified versions of the same algorithms on heterogeneous environments."*
5. *"Amdahl's law built the basis for optimization on heterogeneous environments."*
6. *"The optimization of the parallel sort operation has the highest performance gain in heterogeneous environments of all other operations."*
7. *"Finding the perfect binary tree to perform the optimal parallel sort operation can be done heuristic."*
8. *"Every man is the architect of his own fortune."* and *"Man has no power over his destiny."* is no contradiction.
9. *"Often it is better to wait until things happen, rather than to snatch them."*
10. *"The fortune of your life depends on the kind of your thoughts."*

CURRICULUM VITAE Werner Mach

General Informations

Date of birth	April 6, 1960 Amstetten (Lower Austria)
Citizenship	Austria
Status	Married to Sylvia Mach

Education	1966-1970 Primary School in Amstetten (Lower Austria) 1970-1974 Middle School in Steyr (Upper Austria) 1974-1979 School for higher technical education (HTBLA) Vienna (Austria) 1984-1989 Computer Science and Business Informatics at the University of Vienna 1989 Master degree Magister der Sozial- u. Wirtschaftswissenschaften (Mag.rer.soc.oec.) equivalent to Master of Business Information systems Since Nov. 2006 PhD study at the University of Vienna under the guidance of Prof. Erich Schikuta
------------------	--

Languages	German native English written and oral
------------------	---

Professional Career

Studies in Computer Science and Business Informatics at the University of Vienna (UV), graduation with a Master degree while working as software engineer.

Software development projects like database-applications, CAD and distributed database systems.

Head of the computer department in a world-wide acting electronics company.

Project leader of IT infrastructure projects in Austria and Germany. Many of these projects are trend-setting inside the company, like corporate environment -, networking- and information retrieval concepts.

Project manager of an electronic enforcement application.

Head of the development department for intelligent traffic systems and electronic design.

Head of project management and industrial engineering department.

Vienna, June 2009

Literaturverzeichnis

- [1] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.
- [2] George A. Gravvanis, John P. Morrison, and Heinz Stockinger. Special section: Defining the grid, experiences and future trends. *Future Generation Comp. Syst.*, 25(4):399–400, 2009.
- [3] Zsolt Németh and Vaidy Sunderam. Characterizing Grids: Attributes, Definitions, and Formalisms. 1(1):9–23, 2003.
- [4] D.J. DeWitt and J. Gray. Parallel Database Systems: The Future of Database Processing or a Passing Fad? *SIGMOD record*, 19(4), 1990.
- [5] Heinz Stockinger, Marco Pagni, Lorenzo Cerutti, and Laurent Falquet. Grid approach to embarrassingly parallel cpu-intensive bioinformatics problems. page 58, 2006.
- [6] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*, 1991.
- [7] H. Pirahesh, C. Mohan, J. Cheng, T.S. Liu, and P. Selinger. Parallelism in Relational Database Systems: Architectural Issues and Design Approaches. In *Proc. Of the IEEE Conf. On Distributed and Parallel Database Systems*. IEEE Computer Society Press, 1990.
- [8] M. Stonebraker, P.M. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: a new architecture for distributed data. *Data Engineering, 1994. Proceedings. 10th International Conference*, pages 54–65, Feb 1994.
- [9] Edward Moreno. Hash Join Algorithms on SMPs Clusters: Effects of Netcaches on Its Scalability and Performance. *Journal of Information Science and Engineering*, 18(1-10), 2002.
- [10] Peter Paul Beran, Gernot Habel, and Erich Schikuta. SODA A Distributed Data Management Framework for the Internet of Services. In *GCC '08: Proceedings of the 2008 Seventh International Conference on Grid and Cooperative Computing*, pages 292–300, Washington, DC, USA, 2008. IEEE Computer Society.
- [11] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *IEEE/ACM International Workshop on Grid Computing Grid*, 2000.

- [12] M. Antonioletti, M.P. Atkinson, R. Baxter, A. Borley, N.P. Chue Hong, B. Collins, N. Hardman, A. Hume, A. Knox, M. Jackson, A. Krause, S. Laws, J. Magowan, N.W. Paton, D. Pearson, T. Sugden, P. Watson, and M. Westhead. The Design and Implementation of Grid Database Services in OGSA-DAI. *Concurrency and Computation: Practice and Experience*, 17(2-4):357–376, February 2005.
- [13] Anastasios Gounaris, Rizos Sakellariou, Norman W. Paton, and Alvaro A. A. Fernandes. Resource Scheduling for Parallel Query Processing on Computational Grids. In *5th International Workshop on Grid Computing (GRID 2004)*, pages 396–401, 2004.
- [14] Khin Mar Soe, Than Nwe Aung, Aye Aye Nwe, Thinn Thu Naing, and Nilar Thein. A Framework for Parallel Query Processing on Grid-Based Architecture. In *ICEIS 2005, Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 203–208, 2005.
- [15] B.R. Iyer and D.M. Dias. Issues in Parallel Sorting for Database Systems. In *Proc. Int. Conf. on Data Engineering*, pages 246–255, 1990.
- [16] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, 1984.
- [17] Dina Bitton, David J. DeWitt, David K. Hsaio, and Jaishankar Menon. A taxonomy of parallel sorting. *ACM Comput. Surv.*, 16(3):287–318, 1984.
- [18] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. Benchmarking Database Systems A Systematic Approach. pages 8–19, 1983.
- [19] S. Sudarshan Abraham Silberschatz, Henry F. Korth. Database System Concepts, 5th Edition. 2005.
- [20] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
- [21] E. F. Codd. The Significance of the SQL/Data System Announcement. *Computer-world*, 15(7):27–30, 1981.
- [22] James Steuert and Jay Goldman. The relational data management system: A perspective. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 295–320, New York, NY, USA, 1974. ACM.
- [23] Michael Stonebraker and Eugene Wong. Access control in a relational data base management system by query modification. In *ACM 74: Proceedings of the 1974 annual conference*, pages 180–186, New York, NY, USA, 1974. ACM.
- [24] Michael Stonebraker. Retrospection on a database system. *ACM Trans. Database Syst.*, 5(2):225–240, 1980.
- [25] E. F. Codd. Extending the Database Relational Model to Capture More Meaning. *Commun. ACM*, 4(4):397–434, 1979.
- [26] Shamkant B. Navathe Ramez Elmasri. *Fundamentals of Database Systems*. The

- Benjamin/Cummings Publishing Company, Inc., 1989.
- [27] ISO. www.iso.org. Website, 2009.
 - [28] Ceri Stefano. *Distributed Databases - Principles and Systems*. McGraw-Hill Book Co, 1985.
 - [29] Michael Stonebraker. The Case for Shared Nothing. *Database Engineering*, 9:4–9, 1986.
 - [30] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.
 - [31] Enabling Grids for E-Science. <http://www.eu-egee.org/>. Website, 2009.
 - [32] European Datagrid Project DataGRID. <http://eu-datagrid.web.cern.ch/eu-datagrid/>. Website.
 - [33] Ian Bird. Operating the LCG and EGEE production Grids for HEP. Proceedings of the CHEP’04 Conference, 2004.
 - [34] The Large Hadron Collider LHC. <http://public.web.cern.ch/public/en/LHC/LHC-en.html>. Website.
 - [35] Lightweight Middleware for Grid Computing gLite. <http://glite.web.cern.ch/glite/>. Website.
 - [36] Fabrizio Gagliardi, Bob Jones, Francois Grey, Marc-Elia Bgin, and Matti Heikkuri-nen. Building an Infrastructure for Scientific Grid Computing: Status and Goals of the EGEE Project . *Philosophical Transactions: Mathematical, Physical and Engineering Sciences, Scientific Grid Computing*, 363:1729–1742, 2005.
 - [37] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 2001.
 - [38] Jeffrey M. Nick Steven Tuecke Ian Foster, Carl Kesselman. The Physiology of the Grid. 2002.
 - [39] Aris M. Oukel Kai-Uwe Sattler Angela Bonifati, Panos K. Chrysanthis. Distributed Databases and Peer-to-Peer Databases: Past and Present. *SIGMOD Record*, 37(1):5–11, 2008.
 - [40] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.
 - [41] Renee J. Miller Wnag-Chiew Tan Ariel Fuxman, Phokion G. Kolaitis. Peer Data Exchange. *ACM Transactions on Database Systems*, 31(4):1454–1498, 2006.
 - [42] Andrei Lopatenko Ilya Zaihrayeu Enrico Franconi, Gabriel Kuper. Queries and Updates in the coDB Peer to Peer Database System. In *Proceedings of the 30th VLDB Conference*, pages 1277–1280, 2004.

- [43] J. M. Hellerstein. Toward Network Data Independence. *SIGMOD Record*, 32(3):34–40, 2003.
- [44] D. Suciu I. Tatrinov A. Y. Halevy, Z.G. Ives. Schema Mediation in Peer Data Managment Systems. In *Proc. of ICDE*, 2003.
- [45] Dina Bitton, Haran Boral, David J. DeWitt, and W. Kevin Wilkinson. Parallel algorithms for the execution of relational database operations. *ACM Trans. Database Syst.*, 8(3):324–353, 1983.
- [46] David J. DeWitt. DIRECT - a multiprocessor organization for supporting relational data base management systems. pages 182–189, 1978.
- [47] Haran Boral and David J. DeWitt. Design considerations for data-flow database machines. In *SIGMOD '80: Proceedings of the 1980 ACM SIGMOD international conference on Management of data*, pages 94–104, New York, NY, USA, 1980. ACM Press.
- [48] Team DSLReports. www.dslreports.com. Website.
- [49] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. McGraw-Hill Science/Engineering/Math, July 2001.
- [50] Jae-Dong Lee and Kenneth E. Batcher. Minimizing Communication in the Bitonic Sort. *IEEE Trans. Parallel Distrib. Syst.*, 11(5):459–474, 2000.
- [51] Jai Menon. A Study of Sort Algorithms for Multiprocessor Database Machines. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 197–206, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.
- [52] K. E. Batcher. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, New York, NY, USA, 1968. ACM.
- [53] H. S. Stone. Parallel Processing with the Perfect Shuffle. *IEEE Trans. Comput.*, 20(2):153–161, 1971.
- [54] Peter Kirkovits and Erich Schikuta. Parallel Join Algorithms on Clusters. *Issue of Calculateurs Parallèles journal*, 2001.
- [55] T. P. Martin, P. Å.; Larson, and V. Deshpande. Parallel Hash-Based Join Algorithms for a Shared-Everything Environment. volume 6, pages 750–763, Piscataway, NJ, USA, 1994. IEEE Educational Activities Department.
- [56] Priti Mishra and Margaret H. Eich. Join processing in relational databases. volume 24, pages 63–113, New York, NY, USA, 1992. ACM Press.
- [57] M. Negri and G. Pelagatti. Distributive join: a new algorithm for joining relations. *ACM Trans. Database Syst.*, 16(4):655–669, 1991.
- [58] Leonard D. Shapiro. Join processing in database systems with large main memories.

- ACM Trans. Database Syst.*, 11(3):239–264, 1986.
- [59] X. Wang and W. S. Luk. Parallel join algorithms on a network of workstations. In *DPDS '88: Proceedings of the first international symposium on Databases in parallel and distributed systems*, pages 87–96, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.
 - [60] Patrick Valduriez and Georges Gardarin. Join and Semijoin Algorithms for a Multiprocessor Database Machine. *ACM Trans. Database Syst.*, 9(1):133–161, 1984.
 - [61] Shahram Ghandeharizadeh and David J. DeWitt. Hybrid-range partitioning strategy: a new declustering strategy for multiprocessor databases machines. In *Proceedings of the sixteenth international conference on Very large databases*, pages 481–492, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.
 - [62] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, 1993.
 - [63] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.
 - [64] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing N-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.
 - [65] Marguerite C. Murphy and Ming-Chien Shan. Execution Plan Balancing. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 698–706, Washington, DC, USA, 1991. IEEE Computer Society.
 - [66] P. Erdos and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, 5:17–61, 1960.
 - [67] P. Erdos and A. Rényi. On the strength of connectedness of random graphs. *Acta Math. Acad. Sci. Hungar. Sci.*, 12:261–267, 1961.
 - [68] Matthew B. Doar and Ascom Nexion. A Better Model for Generating Test Networks. pages 86–93, 1996.
 - [69] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. BRITE: An Approach to Universal Topology Generation. 2001.
 - [70] Robert Sedgewick. *Algorithms, 2nd Edition*. Addison-Wesley, 1988.
 - [71] Team Gnuplot. www.gnuplot.info. Website.
 - [72] B. M. Waxman. Routing of multipoint connections. *Selected Areas in Communications, IEEE Journal on*, 6(9):1617–1622, 1988.
 - [73] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286:509, 1999.
 - [74] B. Huffaker, E. Nemeth, and K. Claffy. Otter: A General-purpose Network Visualization Tool. *INET*, 1999.
 - [75] Susan Malaika, Andrew Eisenberg, and Jim Melton. Standards for databases on the

- grid. *SIGMOD Rec.*, 32(3):92–100, 2003.
- [76] Web Services Description Language (WSDL). <http://www.w3.org/TR/wsdl>. Website.
 - [77] Kalinka R. L. J. C. Branco, Marcos J. Santana, Regina H. C. Santana, Sarita M. Bruschi, Clia L. O. Kawabata, and Edward D. M. Ordonez. PIV and WPIV: Two New Performance Indices For Heterogeneous Systems Evaluation. *INFOCOMP Journal of Computer Science*, 5(4):64–73, 2006.
 - [78] Kalinka Regina Lucas Jaquie Castelo Regina and Edward David Moreno Ordonez. Load Indices - Past, Present and Future. *ichit*, 2:206–214, 2006.
 - [79] Domenico Ferrari and Songnian Zhou. An Empirical Investigation of Load Indices for Load Balancing Applications. Technical Report UCB/CSD-87-353, EECS Department, University of California, Berkeley, May 1987.
 - [80] G. S. Wolffe, S. H. Hosseini, and K. Vairavan. An Experimental Study of Workload Indices for Non-dedicated, Heterogeneous Systems. In *Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, p.470, 1997.
 - [81] Edward P. Coleman. Introductory remarks. In *IRE-AIEE-ACM '57 (Western): Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 9–9, New York, NY, USA, 1957. ACM.
 - [82] E. B. Ferrell. Reliability and its relation to suitability and predictability. In *AIEE-IRE '53 (Eastern): Papers and discussions presented at the Dec. 8-10, 1953, eastern joint AIEE-IRE computer conference*, pages 113–116, New York, NY, USA, 1953. ACM.
 - [83] W. Weibull. A Statistical Distribution Function of Wide Applicability. *J. Appl. Mech.*, 18:293, 1951.
 - [84] William B. Poland, Jr., Gilbert E. Prine, and Thomas L. Jones. Archival performance of NASA GFSC digital magnetic tape. In *AFIPS '73: Proceedings of the June 4-8, 1973, national computer conference and exposition*, pages m68–m73, New York, NY, USA, 1973. ACM.
 - [85] K. Das. A comparative study of exponential distribution vs Weibull distribution in machine reliability analysis in a CMS design. *Comput. Ind. Eng.*, 54(1):12–33, 2008.
 - [86] Vassos Hadzilacos. A theory of reliability in database systems. *J. ACM*, 35(1):121–145, 1988.
 - [87] F. Sadri. Reliability of Answers to Queries in Relational Databases. *IEEE Transactions on Knowledge and Data Engineering*, 3(2):245–251, 1991.
 - [88] Y. S. Dai, M. Xie, and K. L. Poh. Reliability of grid service systems. *Comput. Ind. Eng.*, 50(1):130–147, 2006.
 - [89] B. Yang and M. Xie. A study of operational and testing reliability in software reliability analysis. Reliability Engineering and System Safety. *Reliability Engineering and System Safety*, 70:323–329, 2000.

- [90] Hong-Linh Truong, Schahram Dustdar, and Thomas Fahringer. Performance metrics and ontologies for Grid workflows. *Future Gener. Comput. Syst.*, 23(6):760–772, 2007.
- [91] OWL Web Ontology Language Reference. <http://www.w3.org/TR/owl-ref/>. Website.
- [92] Protege. <http://protege.stanford.edu/>. Website.
- [93] Erich Schikuta, Helmut Wanek, and Irfan Ul Haq. Grid workflow optimization regarding dynamically changing resources and conditions. *Concurr. Comput. : Pract. Exper.*, 20(15):1837–1849, 2008.
- [94] Hy Trac and Ue-Li Pen. A Primer on Eulerian Computational Fluid Dynamics for Astrophysics. *Publications of the Astronomical Society of the Pacific*, 115:303, 2003.
- [95] Hao Liu, Amril Nazir, and Soren-Aksel Sorensen. Supporting Dynamic Parallel Application Execution in the Grid. *Semantics, Knowledge and Grid, International Conference on*, 0:420–423, 2008.
- [96] J. Pruyne and M. Livny. Providing Resource Management Services to Parallel Applications. 1995.
- [97] Werner Mach and Erich Schikuta. Performance Analysis of Parallel Database Sort Operations in a Heterogenous Grid Environment. In *Sixth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar) in conjunction with the 2007 IEEE International Conference on Cluster Computing (Cluster07)*, Austin Texas, 9 2007. IEEE Computer Society Press.
- [98] Werner Mach and Erich Schikuta. Analysis of Parallel Binary Merge Sort in a Grid Environment. In *ISCA 20th International Conference on Parallel and Distributed Computing Systems*, pages 187–192, Las Vegas, Nevada, 9 2007.
- [99] Werner Mach and Erich Schikuta. Parallel Database Sort and Join Operations Revisited on Grids. In *High Performance Computation Conference (HPCC)*, Houston, Texas, 9 2007.
- [100] Werner Mach and Erich Schikuta. Parallel Database Join Operations in Heterogenous Grids. In *The 8th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, Adelaide, Australia, 9 2007. IEEE Computer Society Press.
- [101] Werner Mach and Erich Schikuta. Optimized Workflow Orchestration of Parallel Database Aggregate Operations on a Heterogenous Grid. In *The 37th International Conference on Parallel Processing (ICPP-08)*, Portland, Ohio, USA, 9 2008. IEEE Computer Society.
- [102] Werner Mach and Erich Schikuta. Parallel Algorithms for the Execution of Relational Database Operations Revisited On Grids. *Int. J. High Perform. Comput. Appl.*, 23(2):152–170, 2009.

