Institut für Informatik
der Technischen Universität München

Providing efficient, extensible and adaptive intra-query parallelism for advanced applications

Clara Nippl

Institut für Informatik
der Technischen Universität München

Providing efficient, extensible and adaptive intra-query parallelism for advanced applications

Clara Nippl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. M. Paul

Prüfer der Dissertation:

1. Univ.-Prof. R. Bayer, Ph.D.
2. Univ.-Prof. Dr. B. Mitschang, Universität Stuttgart

Die Dissertation wurde am 21.01.2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 20.07.2000 angenommen.

Abstract

Parallel execution offers a solution to the problem of reducing the response time of object-relational queries against large databases. A database management system answers a query by first finding a procedural plan to execute the query and subsequently executing the plan to produce the query result. In this thesis we address all significant levels of the query processing architecture in order to provide a comprehensive approach to the problem of efficient intra-query parallelism.

Thereby, we develop optimization and parallelization algorithms using models that incorporate the sources of parallelism as well as obstacles to achieve speedup. To reduce its inherent complexity, we have split parallelization into several phases, each phase concentrating on particular aspects of parallel query execution. This rule- and cost-based approach guarantees both extensibility as well as effectiveness. Adaptability to diverse application domains and architectural characteristics are provided by means of appropriate parameter settings.

The proposed strategies have been implemented and evaluated within the parallel object-relational DBMS prototype MIDAS. The results show that the presented approach is particularly suitable for the parallelization of large and complex queries, as can be found in upcoming applications such as data warehouses, digital libraries or stream analysis.

Acknowledgments

I express my gratitude to the people and organizations that made this thesis possible.

Many thanks to my advisor, Professor Dr.-Ing. habil. Bernhard Mitschang. I owe him a great deal for his help and guidance from the beginning of the research to the end of this thesis. Thanks for his friendly supervision, visionary instructions and useful comments during the preparation of common publications.

I am grateful to Professor Rudolf Bayer for the analysis of my results and helpful and incisive comments.

I acknowledge the help of my colleagues Giannis Bozas, Michael Jaedicke, Angelika Reiser and Stephan Zimmermann with whom I worked together in the MIDAS project. The research area of Giannis was lock and buffer management. Michael focused on parallelization of user-defined functionality. Stephan worked on transaction management, process architecture, communication as well as benchmarking and performance. This group effort yielded a valuable experimental platform that served also for the performance analysis of this work. I thank Angelika also for her helpful comments for this thesis.

Special thanks to Professor Leonard Shapiro form the Portland State University. He helped me understand the realities of top-down query optimization. I gratefully acknowledge the fruitful discussions with him via mail and meetings. Thanks to Goetz Graefe as well for providing us the Cascades code that served as a basis for our TOPAZ parallelizer and Model-M optimizer.

Thanks to the students that have helped developing and implementing many of the concepts presented in this thesis: Franz Brandmayr, Rolf Druegh, Michael Fleischhauer, Matthias Hilbig, Kay Kruegel-Barvels, Sabine Perathoner, Jean-Jacques Raye, and Steffen Rost. This work also benefits from the effort of all other students that worked in the MIDAS project.

The following friends and colleagues were a source of invaluable discussions and diversions: Paula Furtado, Andreas Müller, Jürgen Sellentin, Aiko Frank, Volker Markl.

I also gratefully acknowledge the valuable comments of the anonymous referees of diverse papers, which have improved this work as well. Furthermore, I would like to thank the Deutsche Forschungsgemeinschaft for supporting this research. The MIDAS project was funded from grants from SFB342, B2.

Last but not least, this thesis would not have been possible without the support and understanding of my family. I thank them all for their affection and continuous care. I owe a debt to my husband Zoltan and son David for putting up with my long hours of work and for their support, love and encouragement.

Table of Contents

Chapter 1
Introduction

1.1 Parallel Databases

In recent years, the amount of data handled by database management systems (DBMSs) has increased steadily. Indeed, it is no longer unusual for a DBMS to manage data in the range from hundreds of gigabytes to terabytes. This massive increase is coupled with a growing need for databases to exhibit more sophisticated functionality such as the support of object-relational extensions. In many cases, these new requirements have rendered traditional sequential DBMSs unable to provide the necessary system performance. This applies especially for information systems that service large numbers of concurrent users.

In order to be able to query these large data volumes in a reasonable amount of time, it is necessary to provide a powerful, high-performance parallel database processing. Scalability, flexibility, and manageability are equally important to achieve the required performance.

Enabling parallel query processing for database systems that feature large volumes of data and concurrent access by large numbers of users entails several new issues that must be solved. Although parallel databases have been an active research area in the past years, most of the work concentrated on isolated problems, mostly in combination with a pre-determined processing scenario, like e.g. the join ordering problem for a specific system architecture. Thus, most of the proposed solutions have left several open questions, especially w.r.t. applicability and interoperability with other system components.

Moreover, recent years highlighted the trend towards extended support for complex data types as well as for user-defined functionality. However, most related work on parallel database systems does not account also for these forthcoming extensions from the simple traditional data supported in the mainstream relational database products.

In this thesis we address the problem of providing efficient intra-query parallelism to queries coming from complex applications, such as OLAP, data-mining, digital libraries etc. Thereby, our main contribution is to provide a comprehensive and integrative approach to this topic. We will address the main components of the query processing system and present the most important aspects w.r.t. intra-query parallelism related to each component.

Special interest is dedicated to the interoperability among the different system components in order to generate efficient parallel plans. The strategies in each module are based on the insights

that result from the realization and validation of other components as well, thus yielding a seamless system architecture adapted to next-generation application development.

An additional focus of this work is to combine parallelism with the necessary extensibility and flexibility that is needed to support object-relational queries as well [JM99]. Therefore, the design of each component follows a generic approach that is independent of any concrete data types, operator functionality, data organization or physical architectural decisions. The customizing to specific application scenarios is done through different mechanisms, such as parameters, (optimization) rules, user-defined operator functionality etc.

All of the proposed strategies are implemented and evaluated within the parallel object-relational DBMS prototype MIDAS. The aim of the MIDAS project [BJ+96] is to build a testbed PDBMS that is well suited to serve as a platform for the exploration of various parallel database technology and its integration into the system architecture. As explained before, the main focus of this thesis is to provide efficient intra-query parallelism for complex applications. Other areas of interest in the MIDAS project include transaction management and system architecture optimization [Zi99], extended database functionality [Ja99], as well as buffer and lock management [Boz98]. The contribution of the entire MIDAS team to the results presented in this thesis are gratefully appreciated.

1.2 Motivation

Query optimization refers to the important process in relational database systems that selects an optimal execution plan for a query from a set of feasible plans, based on some predefined optimization objectives such as minimizing the response time and maximizing throughput. Because of the importance and complexity of query optimization, a large amount of effort has been spent on developing efficient optimization algorithms for various systems, including centralized, distributed and parallel database systems. Most of these works concentrate on the optimization of joins. As shown in [LVZ93] the number x_n of possible plan shapes for a query of n relations is:

$x_n=(2n - 2)!/(n -1)!$.

With this equation, the number of distinct plans for e.g. *6,7,8, 9* and *10* relations are, respectively *30240, 665280, 17297280, 518918432* and *17643225600*.

In a uniprocessor environment, besides choosing the join order, a query execution plan needs to specify the join method as well. Given m join methods supported by the DBMS, the total number of query evaluation plans becomes:

$y_n=m^{n-1}x_n$.

In a parallel system, the optimization problem is even more complicated, with new dimensions and additional parameters introduced by parallelism. First, the number of feasible plans increases dramatically because of the availability of more resources such as processors and memory. Second, proper scheduling and resource allocation are critical to parallel DBMS. Thus, besides determining a good join order and the join strategies used, it is also important to deter-

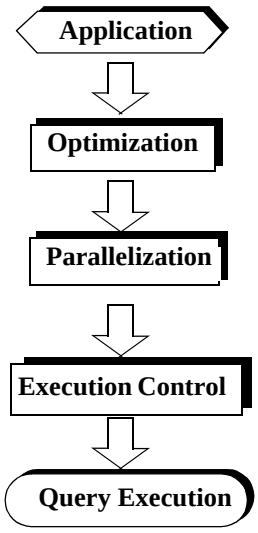


Fig 1.1: Simplified database query processing model

mine the number of operations to perform in parallel as well as the scheduling and allocation of resources to the concurrent operations. Hence, optimization of a query is even more expensive and complicated in a parallel DBMS.

With such a huge search space, it is computationally expensive to perform an exhaustive search to obtain the optimal execution plan. The challenge is to develop efficient heuristics that are not only able to prune the search space effectively but also to ensure that the plan obtained from the restricted space is optimal or near-optimal.

As already mentioned, most previous work concentrated on join operators only. However, upcoming applications stress the demand for new functionalities as well, such as aggregations, all quantification, etc., as well as for object-relational extensions, that are also important factors in determining the optimality of a plan. As shown in [Ju99], the consideration of a single additional operator within query optimization, namely grouping, determines already a dramatic increase of the search space size and optimization performance.

Furthermore, in order to reduce search complexity, most related work concentrated only on specific forms of intra-query parallelism or plan shapes. However, any simplifying assumption w.r.t. the search space, like e.g. considering only linear execution plans, incurs the risk that the optimal or near-optimal solutions might be excluded.

As stated in the previous section, our aim is to elaborate a comprehensive approach to query optimization and parallelization, taking into account all forms of parallelism. To reduce complexity, we have split plan generation into subsequent phases, each phase concentrating on particular aspects of parallel query execution. This strategy is also reflected within the query processing architecture. For an easier understanding, we have depicted a simplified model of this architecture in Fig. 1.1.

In our approach, the search space is refined gradually. In the first phase, the *optimizer* generates a plan where only selected aspects of the subsequent parallelization are taken into account. A detailed parallelization follows within the *parallelizer.* This component takes the costs of the participating operators into account to determine the best segmentation strategy and to derive a

set of parameters for run-time aspects, like e.g. degrees of parallelism, resource consumption etc. Finally, a query *execution control* component takes the current run-time environment into account to derive the final values for the parameters as well as the best scheduling policy.

This comprehensive multi-phase strategy based on modularization and parametrization is in contrast to other approaches that incorporate either detailed knowledge, as in the so-called one-phase approach, or no knowledge (cf. the two-phase approach) of the parallel environment within query optimization. A detailed evaluation of these strategies and processing architectures w.r.t. optimization, parallelization and scheduling will be given in the subsequent chapters.

1.3 Overview

The thesis is organized as follows. In Chapter 2, we start with some basic concepts and notations based on our MIDAS prototype. Thereby, we restrict ourselves to concepts of direct relevance to this work.

In the following, we concentrate on the efficient realization of intra-query parallelism throughout different database modules. Necessary interoperability issues are best understood by proceeding in a bottom-up manner within query processing cf. Fig. 1.1.

In Chapter 3 we focus on the execution level. Thereby, we present the *data river* paradigm. This paradigm is used for the management of intermediate query result sets that are produced as well as consumed by operators in a parallel database engine. We point out some aspects related to this paradigm that have a serious impact on query processing and efficiency. In addition we present an implementation based on a stringent modularization concept in combination with a set of parameters that on one hand provide necessary flexibility and on the other hand contribute to significant performance improvements. Furthermore, based on a thorough performance analysis we present a comprehensive set of parameter combinations that are recommended for specific situations covering the variety of communication patterns typically found in parallel database engines.

In Chapter 4 we concentrate on the generation of efficient parallel plans that can make best use of the concepts presented before. Currently the key problems of query optimization are extensibility imposed by object-relational technology, as well as query complexity caused by upcoming applications, such as OLAP. We propose a generic approach to parallelization, called *TOPAZ*. Different forms of parallelism are exploited to obtain maximum speedup combined with lowest resource consumption. The necessary abstractions w.r.t. operator characteristics and system architecture are provided by rules that are used by a cost-based, top-down search engine. A multi-phase pruning based on a global analysis of the plan guides the search process, thus considerably reducing complexity and achieving optimization performance. Since TOPAZ solely relies on the widespread concepts of iterators and data rivers common to most (parallel) execution models, it fits as an enabling technology into most state-of-the-art object-relational systems. This is additionally enforced by the fact that special strategies have been devised

within the parallelizer to overcome specific problems that result from the usage of the data river paradigm, such as e.g. deadlocks.

Chapter 5 provides a detailed description of the *TOPAZ cost model*. Furthermore, it illustrates the consideration of resources within the cost calculation and presents a pruning package that has been elaborated to effectively reduce the search complexity of the parallel search space.

Since the input of TOPAZ represents a sequential physical plan, we further concentrate in Chapter 6 on necessary modifications that affect the *optimizer* component. The input of TOPAZ is delivered by a cost-based optimizer, called *Model-M*. This component uses the same top-down search strategy as the TOPAZ parallelizer, but explores different search space regions. Prior work has shown that traditional two-phase parallelization, i.e. parallelizing the best sequential plan, produces suboptimal results. Hence, our approach employs a so-called *quasi-parallel cost model* that accounts for parallelism already in the optimization, i.e. in the sequential plan generation phase. Thus, the Model-M optimizer perfectly fits into the overall parallelization scheme, as it contributes towards a phase-wise search space exploration and refinement. The validation of this overall generic parallelization scheme shows that in the average linear speed-ups are obtained without producing significant additional overhead.

An exception from the bottom-up presentation within the query processing system is merely provided by the *query execution control (QEC)* module. In Chapter 7, this is the last database system component to be presented in this thesis. This is because the QEC module combines the tasks of scheduling, load balancing and resource allocation by interfacing the run-time environment with other system components. Hence, it collects its input from diverse sources presented before. In order to provide the necessary scalability, this module is based on a distributed design. A phase-oriented strategy yields both robustness as well as adaptability to non-uniform hardware configurations and different run-time aspects, such as data skew etc.

After presenting the extensions performed on the query processing system, in Chapter 8 we move our focus towards the validation of the proposed approaches. Therefore, we consider a class of applications having as a common feature the need to process *sets* of items that satisfy a specific requirement. This class contains various applications such as data mining, pattern recognition, financial time series and profile matching in digital libraries. By evaluating this class, we aim to investigate if the proposed approaches towards efficient and extensible intra-query parallelism are capable of facing new requirements in today's complex, rapidly changing world as well. Thereby, we first introduce a new operator, called *StreamJoin*, as an efficient strategy to solve the itemset generation problem of the given application class. We show how this operator can effectively be integrated within the processing system. For a better illustration, different application-specific query execution plans are provided as well. Furthermore, we validate through measurements results that the necessary functionality extensions are also able to make profit of the inherent intra-query parallelization scheme of the MIDAS system that results from the implementation of the strategies in this thesis.

Thus, the class of applications considered in Chapter 8, together with the class of OLAP applications, for which we have chosen the TPC-D benchmark [TPC95] as a representative to effectuate most of our performance analysis in the previous chapters, convincingly demonstrate that the concepts presented in this thesis have yielded an extensible, parallel and solid technology

that is capable of facing the requirements of next-generation complex query types as well. Finally, in Chapter 9 we summarize our contributions and discuss some open problems.

Chapter 2
Basic Concepts and Notations

This chapter gives a short introduction to MIDAS, our testbed parallel database system and thereby introduces the basic concepts and terminology used in this thesis.

2.1 Introduction

The main focus of the MIDAS project is to provide efficient support for complex applications, such as *on-line analytical processing (OLAP)*, *decision support systems (DSS)*, *data mining* etc.

MIDAS (**M**un**I**ch Parallel **DA**tabase **S**ystem) is a prototype parallel object-relational database system (ORDBMS) running on a hybrid architecture. This architecture comprises several *processing sites* (also called *nodes* or *hosts*) that communicate via a high-speed interconnection, e.g. fast ethernet. The processing sites can be either simple workstations or shared-memory computers (also called SMP for symmetric multi-processing). The starting point of the MIDAS project was a commercially available sequential relational DBMS, the TransBase SQL-DBMS [Trans95]. Parallel database technology has been integrated gradually into this system either by component exchange or by system extension. For portability reasons the whole system has been embedded into the Parallel Virtual Machine environment (PVM) [Ge94, BFZ97].

The MIDAS system supports different kinds of parallelism. *Inter-transaction parallelism* allows for a parallel processing of concurrent transactions, thus increasing transaction throughput [Zi99]. *Intra-transaction parallelism* focuses on parallel query processing within a single transaction, in this way especially reducing the response time of complex transactions. Herein, the main parallel processing feature in MIDAS is *intra-query parallelism*. This kind of parallelism comprises inter- and intra-operator parallelism as well as data and pipeline parallelism. Intra-query parallelism is needed to achieve acceptable response times for complex and data-intensive operations as occurring e.g. in decision-support and data warehousing systems, geographic information systems or multimedia database systems. Since database systems are no single-user systems, such a parallel query processing must be supported for multiple concurrently running queries and possibly in parallel to OLTP transactions. Please note that this thesis deals with intra-query parallelism only.

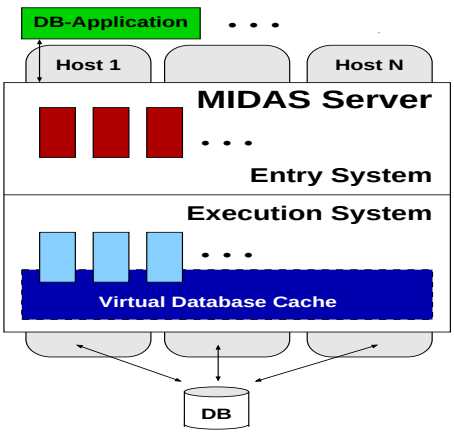


Fig 2.2: The MIDAS prototype

MIDAS realizes a client/server architecture as depicted in Fig. 2.2. The MIDAS clients are database applications. They are sequential programs performing transactions on the *MIDAS Server*. A MIDAS client can run on any computer having access to the MIDAS Server.

The Server is composed of two layers, the MIDAS *entry system* that corresponds to the logical database processor and the MIDAS *execution system* that corresponds to the physical database processor. The components of the server communicate via PVM.

The architectural components of MIDAS will be described in detail in the next section. Section 2.3 deals with the MIDAS query processing architecture. Finally, Section 2.4 presents the MIDAS execution model.

2.2 MIDAS System Architecture

The **MIDAS entry system** supports inter-transaction parallelism, i.e. parallel execution of different MIDAS client applications. For that purpose, the entry system consists of a varying number of *application servers* and an *administration server* that provide a mechanism such that an arbitrary and varying number of clients can issue their queries to the MIDAS Server in parallel. There is only one administration server process for each DBMS instance. It assigns each new client application exclusively to one application server. Hence, the client transmits all further queries directly to the corresponding application server. A query is compiled, optimized, parallelized, and finally transformed into a *parallel query execution plan (PQEP)* which can be performed by the execution system in parallel. In addition, the application server initiates, schedules, and controls the parallel execution of these PQEPs. Finally, this component is also in charge of transmitting the PQEP evaluation results back to the application.

The **MIDAS execution system** is responsible for intra-query parallelism. This is achieved by the capability of the execution system to work on different parts of one PQEP simultaneously. The execution system is designed as a set of *execution units (EU)* and a set of *segment servers* and *segment slaves*. Furthermore, the execution system comprises the *virtual database cache (VDBC).* This component provides storage and transaction

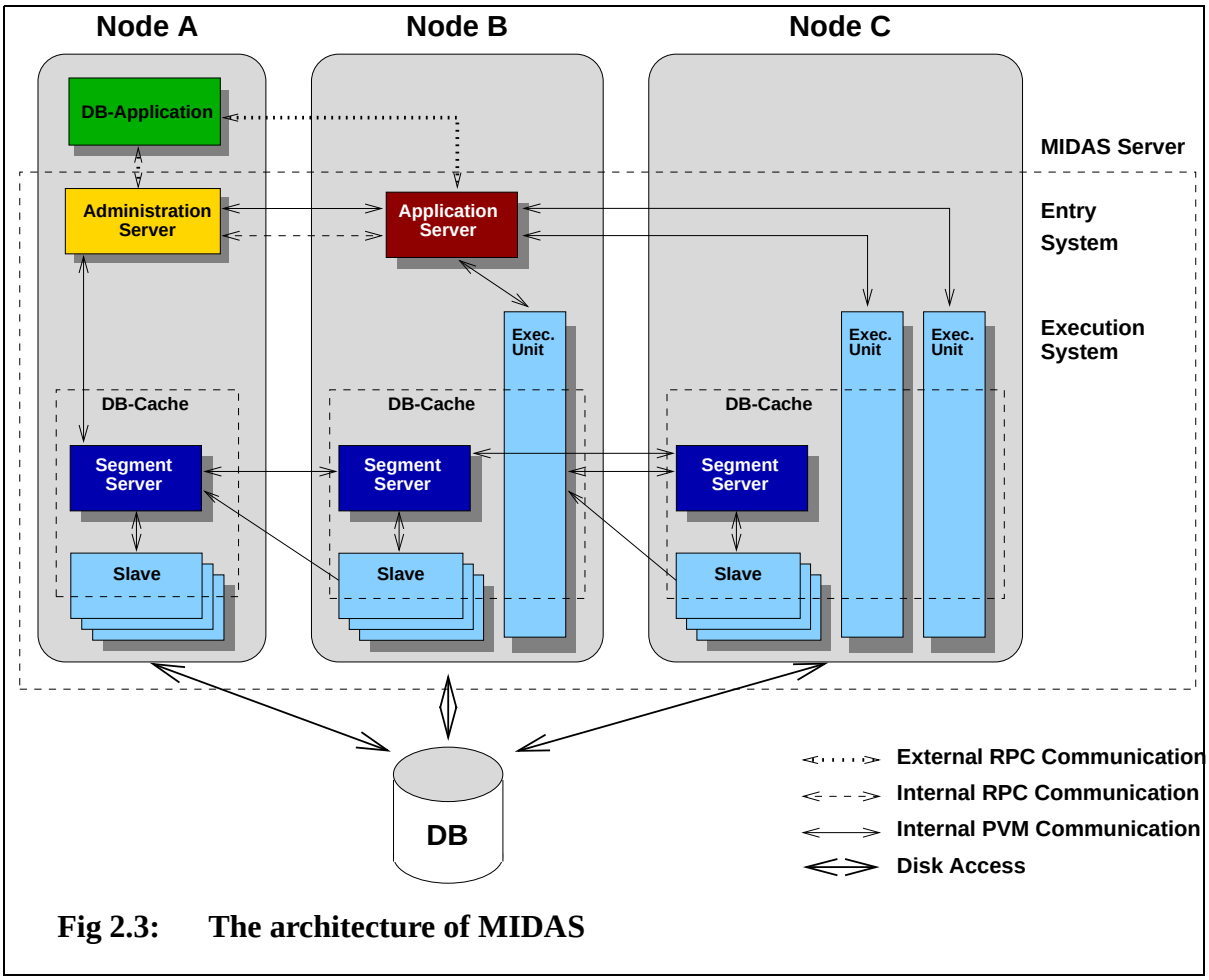


Fig 2.3: The architecture of MIDAS

support as well as efficient access to the shared physical database for all concurrent execution units.

A query is processed exploiting multiple execution units, i.e. multiple independent processes, as depicted in Fig. 2.3. Please note that MIDAS cannot use threads to lower the inter-process communication costs, because PVM is not thread-safe. The application servers are in charge of dynamically starting and removing execution units as needed. Obviously, the actual number of execution units depends on the number of PQEPs concurrently performed by the MIDAS execution system and the chosen degree (or even degrees) of parallelism for each PQEP. Each execution unit works on one portion of a parallel execution plan as determined and assigned by the corresponding application server. In the following, we will call such PQEP portions *subplans*. The execution units that process subplans of the same query communicate via the VDBC component by exchanging buffer frames. The corresponding mechanism will be detailed in Section 3.3.1. For efficiency reasons and in order to overcome communication overhead, each execution unit has full access to the physical database.

The virtual database cache is managed by the segment server and segment slave processes. Each processing site of the MIDAS parallel ORDBMS employs exactly one segment server. This segment server handles all requests for buffer pages that come from other hosts. Thus, it is also in charge of cache coherency and concurrency control [LB97, Boz98]. All requests that involve time-consuming operations, like I/O operations, or that can be blocking, e.g. due to deadlock problems, are delegated to a segment slave process.

2.3 Query Processing in MIDAS

One goal of the MIDAS project is the design of an adaptive cost-based query execution integrating scheduling and load balancing. In order to reach this goal, we have to solve two problems: first, at compile-time, the generation of a parallel QEP and second, at run-time, the adaptive, parallel execution of this plan.

The parallel plan is generated in several steps, shown in Fig. 2.4. The query processing architecture of MIDAS first incorporates the *parser* that compiles (SQL-)queries. For each statement the parser produces an initial QEP. A QEP is internally represented as a C-structure. However, it is possible to transform QEPs into a textual (and corresponding visual) format called *gentree*. Such a gentree can also be used as an input for the parser via a special interface. An application can directly execute a gentree via the call level interface. In the following sections, we will mostly use the gentree representation to visualize QEPs.

The initial QEP is further pre-optimized by some passes of the *TransBase pre-optimizer*. This optimizer applies heuristic transformations in several passes to an operator plan. While this component did the complete optimization in TransBase, it is currently replaced by a new *optimizer* called *Model_M* (*M* for MIDAS). However, some passes of the TransBase optimizer are still used as a pre-optimization step, for example to generate a normalized representation.

The pre-optimized execution plan constitutes the input of the Model-M optimizer. The goal of this component is to produce a sequential QEP with a high potential for parallelization. The corresponding concept will be detailed in Chapter 6. The resulting sequential plan is handed to the *parallelizer*, called TOPAZ. Both, optimizer and parallelizer, are implemented based on the *Cascades Optimizer Framework* [Gr95, Bi97] representing a cost-based and rule-driven approach.

The TOPAZ parallelizer, treated in detail in Chapter 4, analyzes the sequential QEP for its potential w.r.t. parallelism. To reduce complexity, it splits parallelization into subsequent phases, each phase concentrating on particular aspects of parallel query execution. TOPAZ uses

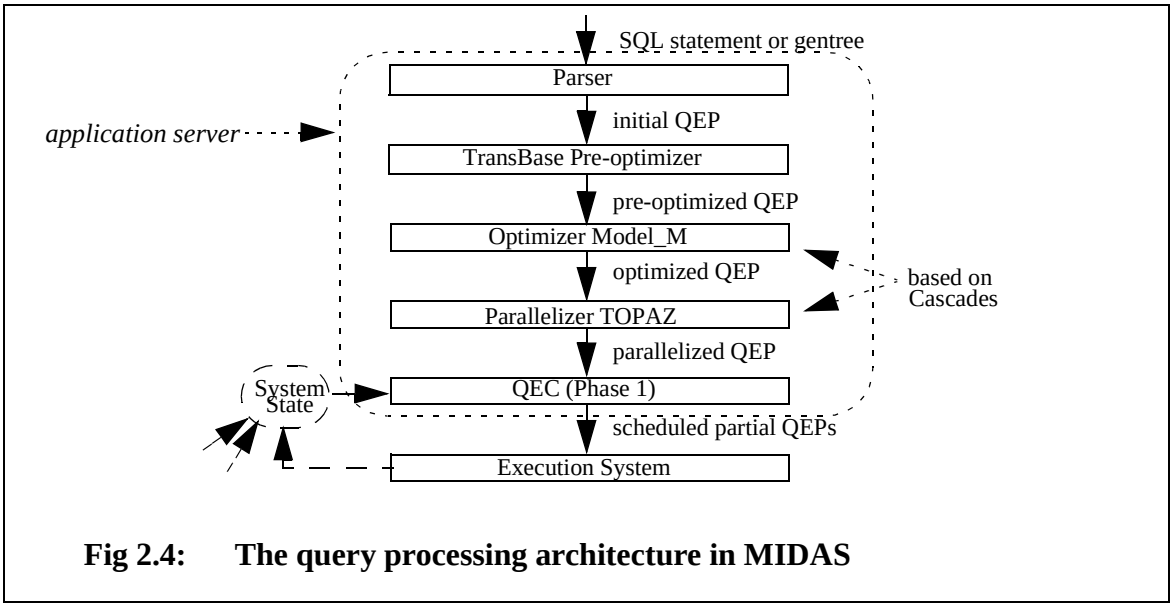


Fig 2.4: The query processing architecture in MIDAS

a novel parallel cost model. This model comprises besides CPU-costs also communication costs, memory usage, and disk accesses and it supports inter-operator as well as intra-operator parallelism by means of pipelining and data parallelism. The search complexity is additionally reduced by a pruning package that is designed to consider also global constraints in the course of parallelization. To be able to perform the parallelization step at compile-time, the decisions of the parallelizer must not depend on the current system state. This loose coupling is realized in MIDAS through parameters. They determine the *degrees of parallelism (DOP)* as well as other run-time aspects like memory allocation, buffer management etc.

This parameterized PQEP allows the run-time component, called *Query Execution Control* (*QEC*, cf. Fig. 2.4), to derive individual PQEPs with parameter values that are adjusted to the run-time system state. Thus, QEC comprises the run-time responsibilities of load balancing, scheduling, and resource allocation. It performs a fine-tuning of the PQEP and schedules different portions of the PQEP to different execution units. As presented in more detail in Chapter 4, this is done in two phases. The decisions take the current system state and the provided cost formulas into account. This allows to choose the parameters in such a way that the query execution resources match both resource availability and contention in the best possible ways. Therefore, load information is gathered from different system components (like buffer manager, lock manager, and operating system) at all processing sites and further combined and condensed in order to get the global system view (cf. Fig. 2.4).

The compiler, optimizer and parallelizer components are part of the MIDAS entry system and are realized within the application server. The mediator role of the QEC module between the MIDAS entry and execution system is also reflected by its architectural design. Thus, the first phase is executed within the application server, while the second phase is part of the execution system.

2.4 The MIDAS Execution Model

The MIDAS operators are designed according to the *iterator* (or *Open-Next-Close*) processing model [Gr95, GB+96]. Iterators are self-contained software objects that accept streams of rows from one or several data sets, apply a predefined behavior and manipulate the data according to the specifications of the behavior. Thus, regardless of what type the iterator is and what behavior is associated with it, all iterators follow the same model. Some of the MIDAS operators, used by the execution system, are listed in Appendix B.1.

In this model that is used also in object-relational DBMSs [GB+96], queries are structured by bundling together the appropriate operators (iterators) into a QEP (Fig. 2.5a). In a sequential DBMS, each QEP is processed by a single execution unit. In the course of parallelization, this QEP is broken down into several subplans or *blocks* [TD93] (Fig. 2.5b) that define the granularity or unit of parallelism [HS93]. A block can be considered as a single operator whose processing cost is the sum of the processing costs of the constituting operators. It is vital to perform a cost-based analysis to identify the optimal granularity for parallelism, i.e. number of blocks for a given query, number of operators within a block, and degree of parallelism assigned to a

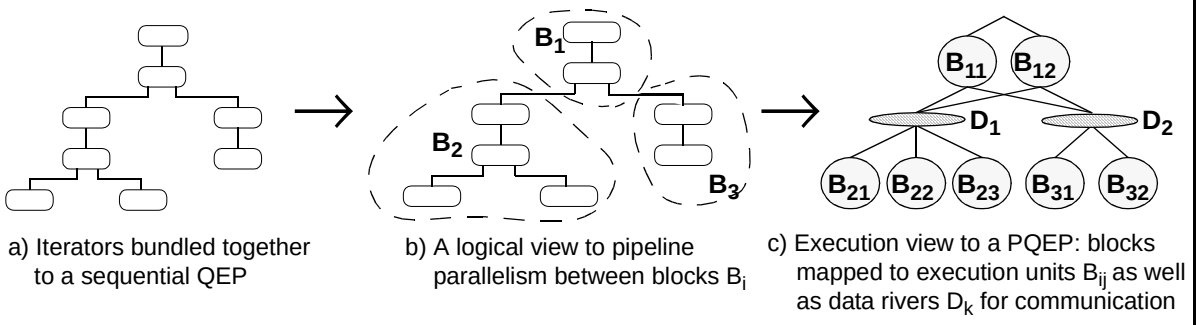


Fig 2.5: From a sequential QEP to parallel execution using blocks and data rivers

block [NM98a]. As already mentioned, this is achieved in MIDAS by the TOPAZ parallelizer.

As determined by the parallelizer and at run-time adjusted by the QEC, a block is assigned to one or several execution units, according to its degree of parallelism. The flow of tuples among the various execution units is organized by the concept of *data rivers* (D_1 and D_2 in Fig. 2.5c). This mapping, presented in more detail in Chapter 3, supports data parallelism within a block as well as pipeline parallelism in between blocks.

More information on the MIDAS prototype can be also found in [Zi99, Boz98, Ja99, Br97, Fl97, Hi98, KB98, Pe98].

Chapter 3

Efficiently Exploiting Data Rivers for Intra-Query Parallelism

This chapter concentrates on obtaining efficient intra-query parallelism on the execution level. Therefore, we evaluate the data river paradigm for the management of intermediate query result sets that are produced as well as consumed by operators in the parallel database engine.

3.1 Introduction

The *data river* paradigm [Gr95, DG92] has been devised to achieve inter- and intra-operator parallelism between producing and consuming instances by means of special communication operators. Meanwhile this concept is used in many (object-)relational PDBMSs under various synonyms, e.g. the *send/receive* operators in DB2 UDB [JMP97, Ibm98], the *split/merge* operators in Gamma [DG92], or the *Exchange* operator in Informix Dynamic Server and Volcano [Gr94, Zo97, Inf98]. However, while validating our implementation of this concept, we have identified a set of parameters that significantly impact the performance of parallel queries, in terms of speedup as well as resource consumption. These parameters concern dataflow control, the granularity for data transport, as well as specific measures to minimize overhead and thus increase efficiency. Based on this insight we developed a clean modularization of the data river paradigm that provides this set of parameters and that resulted into an extensible approach covering the whole spectrum for communication found and needed in parallel query processing. To our knowledge, no other publicly available report deals with these aspects and its consequences at this level of parallelism and detail.

The rest of the chapter is organized as follows. First, we develop the primitives of the data river paradigm. Section 3.3 details our design and implementation of data rivers, coming up with a modularization and parameterization approach. As a first summary, a set of parameter combinations is investigated that is recommended for specific situations covering the necessary spectrum for communication. Section 3.4 presents possible deadlock scenarios when exploiting the

data river paradigm. Efficient embedding of the data river paradigm into query processing is discussed in Sections 3.5 and 3.6. Finally, Section 3.7 covers related work; a summary is given in Section 3.8.

3.2 Anatomy of the Data River Concept

As already mentioned in the previous section, the data river concept, based on split and merge of intermediate result tuple sets, is adopted also by many (object-)relational DBMS as it complies best with the iterator concept for the operators. If multiple producers add records to the same river that is consumed by several consumers, the river consists of several parallel *data streams*. In this way, parallelism is transparent for operators or operator instances, as they still operate sequentially on these data streams that constitute the data river.

In Fig. 3.1 we detail the data river concept (data streams are visualized as black bars) to distinguish the following basic communication patterns that comprehensively support intra-query parallelism:

- **Pipelining:** This is the simplest way of intra-query communication. In Fig. 3.1a left, the output of producer B_2 is consumed by consumer B_1. In this case, the data river consists of a single data stream. In the more general case, when consumer and producer have the same degree of parallelism N, the corresponding data river consists of N distinct streams (Fig. 3.1a right).
- **Replication:** In this case, all consumer instances read the same input. In Fig. 3.1b left, a single block produces a data stream, that is read by all consumer instances. In the more general case (Fig. 3.1b right), N instances produce N data streams that are read by all consumer instances. Thus, for replication N data streams are necessary, N being the degree of parallelism of the producer block.
- **Partitioning:** In Fig. 3.1c, the tuples produced by B_2 are partitioned according to a given criterion and written into the corresponding data stream that is read by an instance (B_{11} to B_{1M}) of consumer B_1. In this case M data streams are needed, where M is the degree of parallelism of the consumer block.

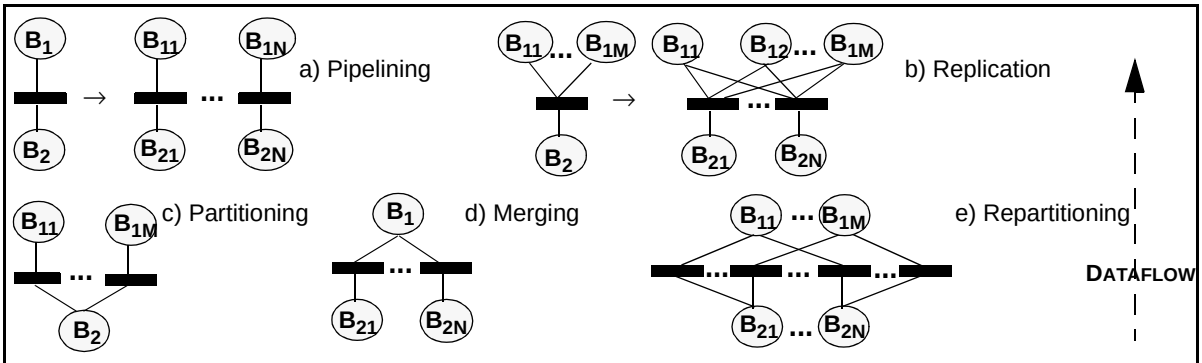


Fig 3.1: Communication patterns realized by using data rivers

- **Merging:** To produce a final result, each producer instance fills a separate data stream with its intermediate results (see Fig. 3.1d). These are read by the consumer which merges the tuples coming from the different data streams according to the requirements for further processing (like sort ordering etc.). Please note that another possibility would be to have a single data stream, that is jointly filled by all producer instances. The disadvantage of this communication pattern is that the merging has to be done on the producer side. Hence, any requirements for data stream properties at the consumer side have to be taken into consideration already at the producer side. Moreover, the producer instances cannot operate independently from each other. Thus the independency criterion, i.e. being self-contained processing objects, would not be satisfied for operators and operator instances.
- **Repartitioning:** Different degrees of parallelism or different partitioning strategies on both producer and consumer side imply a repartitioning of the data as shown in Fig. 3.1e. For a *N:M* redistribution, as each of the *N* producers write into *M* partitions, each mapped to a data stream, *N x M* data streams are necessary to build this type of data river.

3.3 Implementation Concepts for Data Rivers

In the following we describe the design and implementation of the data river paradigm as developed in the PDBMS MIDAS. Please note that in the logical concept presented in Section 3.2, there is no dependency on any specific execution architecture. In this section, we concentrate on an implementation concept for data rivers that preserves this independency to a large extent. Thereby, we identify the following basic aspects of the data river approach: data flow, data partitioning and data merging. Each aspect is implemented by a separate module and a set of parameters to offer maximum adaptability to specific situations. In addition, extensibility measures to the data rivers concept are also valuable for forthcoming hybrid heterogeneous architectures [NZT96, HFV96].

In the following we concentrate on the modularization approach and show the impact of the defined parameters on performance. Thereby, a thorough performance analysis is conducted and a comprehensive set of parameter combinations is identified that are recommended for specific situations.

3.3.1 Communication Segments as a Concept to Implement Data Rivers

In order to organize the flow of intermediate and result tuples, the parallelizer equips each block instance with two communication operators: `send` and `receive`. These follow the same iterator concept as all the other operators, thus hiding all particularities and implementation aspects of the data river concept. Each block instance, excluding the top one, has as root a *send* operator, which transmits the resulting tuples to one or more parent blocks. Many DBMSs use a special communication subsystem for the implementation of the data river between the *send* and *receive* operators [BF+95, Zo97]. In contrast, we decided to view a data stream as a special temporary

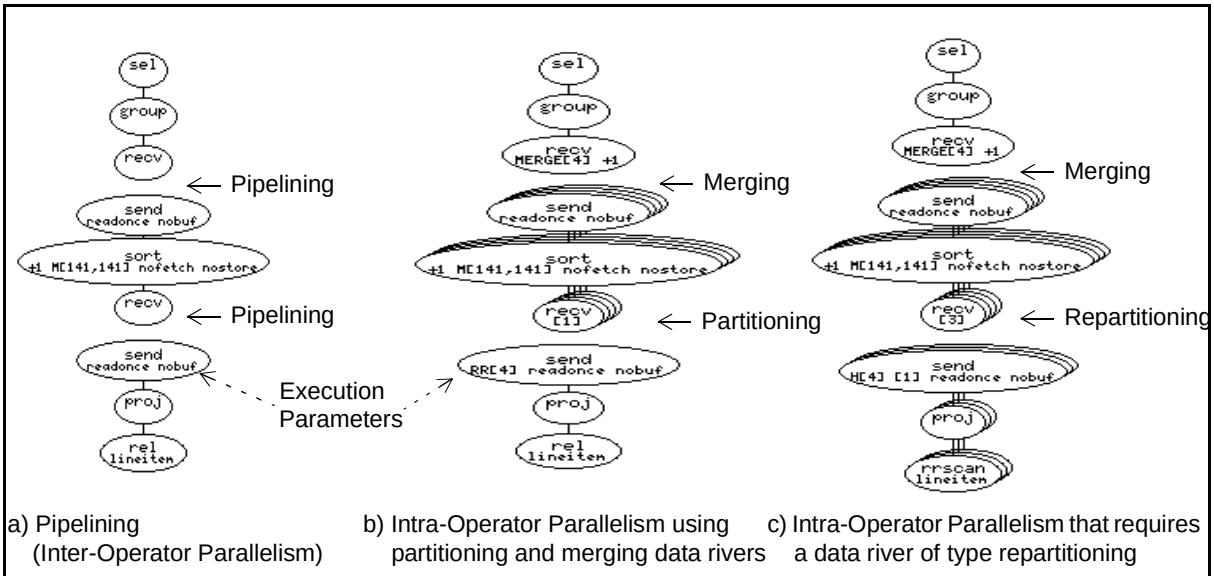
relation mapped to so-called *communication segments (CS)* that are handled by the storage system and by the database buffer management (VDBC). CSs are temporary segments that are only visible inside a transaction and that are deleted at the end of the query or transaction. The same kind of segments are used for instance by the *sort* operator to store initial runs. Thus, *send* respectively *receive* are implemented to write to (resp. read from) CSs. With this concept of mapping each data stream to a CS, MIDAS uses the same model for permanent and intermediate data sets. Additionally, the communication between consumers and producers takes place in an efficient buffered manner.

MIDAS supports different data organizations within a CS. In addition to the very general notion of a tuple stream flowing from producer to consumer execution units, one can use appropriate data structures to improve the execution of both producer and consumer processing. For example, sorting (the task of the producer) as well as sorted access (a consumer requirement) can be considerably simplified by a B-tree data organization of the CS representing the data stream; in addition, such a data structure allows for direct and repeated access. Thus, by storing intermediate results in 1 or several CSs, MIDAS allows the reuse of these results, a prerequisite to optimization for common subexpressions as well as to multi-query optimization [Qi96, Se88].

With the concept of data streams mapped to CSs, we could realize all communication patterns necessary for intra-query parallelism as described in Section 3.2. However, in order to make a certain communication pattern executable, several settings for data partitioning, data merge, or data flow have to be provided. This will be discussed in the following subsections and exemplified by the QEP examples given in Fig. 3.2. Please note that in this representation, the operators bracketed by *send* and *receive* (*recv)* nodes are bundled together to a block. Blocks can have different degrees of parallelism depicted as a sequence of overlapping operators. For simplification purposes, we omit the data rivers and the associated data streams at the borders of a block, but we express the bundling of operators within a block by connection lines. Most operators (e.g. the *sort*), show certain parameters that describe the operator execution in more detail, as e.g. memory allocation and management, sort parametrization etc. These parameters can also be adapted to the system state at run-time by the QEC component.

3.3.2 Control of Dataflow

In MIDAS, the dataflow granularity between execution units that communicate via CSs is a *page* or alternatively a *subpage*, i.e. a fraction of a page, since MIDAS uses subpages for coherency control, locking, as well as logging/recovery to reduce overhead [Li94]. Flow control is achieved by a page-based locking scheme. For instance, consuming block instance(s) are stalled until at least one page of their input data streams, i.e. CSs, is filled up by the corresponding producer. A local reader can access the pages directly through the database buffer, so there is no need for memory copies. Otherwise, the transmission of the page is achieved through (PVM) messages and by memory to memory transfer. Each CS has a certain quota of buffer pages assigned to the producer. If this is exhausted, one of the following dataflow control strategies is used, depending on the *send* operator's parameters that are set by the parallelizer or dynamically reset by the QEC:



a) Pipelining (Inter-Operator Parallelism)

b) Intra-Operator Parallelism using partitioning and merging data rivers

c) Intra-Operator Parallelism that requires a data river of type repartitioning

Fig 3.2: Forms of intra-query parallelism in MIDAS

- WRITEOUT: The "newest" page is written to disk. The rationale behind this decision is that the older pages, that are still buffered, are the next to be read by the consumer.
- NOBUF: The original LRU replacement algorithm of the buffer management is used, hence no explicit quota has to be specified. However, in contrast to the WRITEOUT strategy, the CS pages may be replaced due to any page request to the database buffer. According to the LRU strategy, in this case the "older" pages are affected, although they are the next to be read by the consumer(s).
- WAIT: In this case the producer is stalled until the consumer(s) has/have read pages and request(s) more input.

Based on these definitions one can easily observe that WRITEOUT is just a NOBUF strategy restricted by a fixed buffer quota and a FIFO replacement strategy; WAIT refers to a fixed buffer quota without any replacement, but with a hold request on the producer if the quota is exceeded. Obviously only the NOBUF and WRITEOUT strategies lead to materialization, hence we call these *send* operators at some places *materializing* nodes. If there is only one consumer instance (i.e. no replication or multi-query optimization), a page can be deleted as soon as it is read. This is achieved by an additional parameter, called *readonce*. Hence, if this parameter is used in combination with WAIT, I/O can be avoided entirely.

3.3.3 Control of Data Partitioning

By streaming the output of one operator into the input of the other operator, the two can work concurrently giving *`inter-operator parallelism`* or pipelining. This is achieved by simply placing a *send-receive* pair on any edge of a QEP. Thus, the QEP in Fig. 3.2a has been split up in 3 blocks that communicate in a pipelining manner via CSs as described above.

`Intra-operator parallelism` in MIDAS is based on data partitioning. The splitting and merging of data streams is performed by the *send* and *receive* operators as well. For each

partition a separate CS is created and filled by the *send* operator with corresponding tuples. In this way, each operator (or block) instance processes one partition of the data. The strategies implemented are `round-robin, hash, range,` and `user-defined partitioning` [JM99]. The particular technique used depends on the type of the operator that has to be parallelized. Thus the partitioning often has to keep track of the attribute values, like in the case of hash- or range-based partitioning. An example is given in Fig. 3.2b. Here, the *sort* operator has a degree of parallelism of 4, each instance processing a single partition of the initial data stream. As in the case of sorting a value-based partitioning is not necessary, a round-robin strategy has been chosen, indicated in Fig. 3.2b by the parameter *RR[4]* set for the *send* operator. The reason for this choice is that round-robin partitioning is cheap and prohibits data skew.

In Fig. 3.2c, an example for repartitioning is presented, where a change from DOP=3 to DOP=4 has to be accomplished. As described in Section 3.2, for a *N:M* redistribution the data river consists of *N x M* data streams, each mapped to a separate CS instance. Thus, in this example 3 x 4 = 12 CSs are used. For illustration purposes, in this example the *send* operator performs a hash partitioning on the (first) sorting attribute, indicated by parameter *H[4] [1]* of the *send* operator.

3.3.4 Control of Data Merging

In order to combine several parallel data streams created by the *send* operator as described before, the *receive* operator has to read from multiple CSs as well. The communication between *send* and *receive* can be based on polling or on a notification technique. In MIDAS, both alternatives are implemented, the first one being the default mode and the second one being triggered by the NOTIFY option. The decision on which mode to select is made by the QEC component, according to the relative costs of the consumer and producer instances. If the consumer is slower than the producers a polling technique is beneficial since most probably the first request for data can already be answered. In the opposite case, the NOTIFY alternative is favorable.

With respect to the order in which the data streams are read, the following alternatives are possible:

- MERGE: In this case, the CSs containing locally sorted data streams are merged into a final sorted stream (see Fig. 3.2b and Fig. 3.2c).
- SEQ: Here one entire data stream is consumed before the next is worked on. As a side effect, a sorted output can be produced directly from range partitioned and locally sorted data streams.
- ASYNCH: In this mode, the tuples are read in their order of arrival. The output is unsorted. In contrast to the other modes presented, this one does not prescribe how the streams are merged. As soon as any page is available it is subject to consumption. Hence, we call this a *data-driven* consumption while the other two policies are called *demand-driven*.

3.3.5 Performance Measurements

With the concept presented, we could parallelize in MIDAS traditional and application-specific

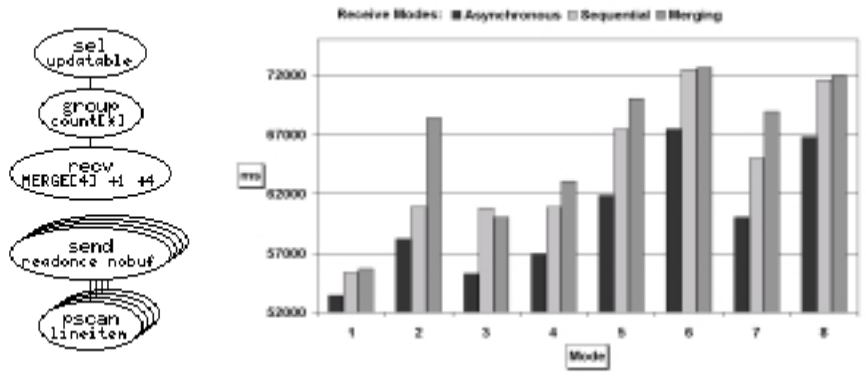
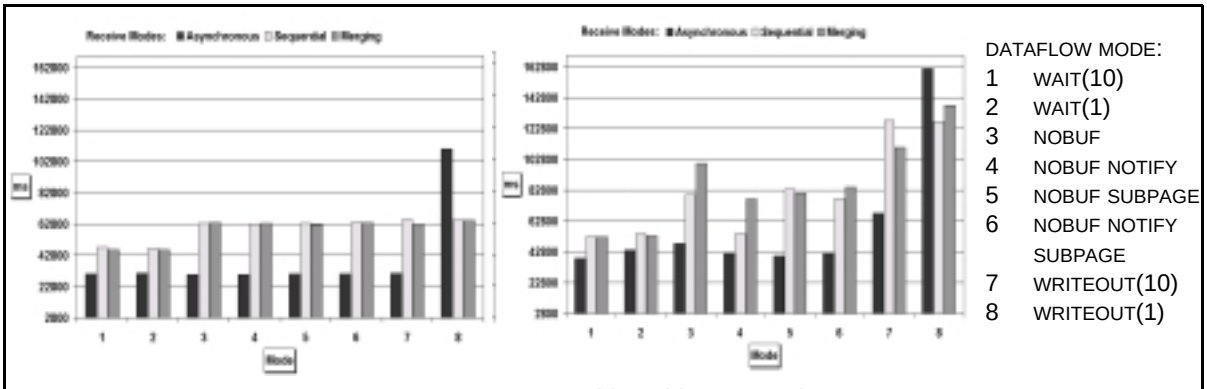


Fig 3.3: Example query and performance using different dataflow parameters (cluster)

operators [NJM97]. This has been extended to user-defined functions and table operators as well [JM98, Ja99]. In addition, parallel I/O is supported by new parallel scan operators that exploit specific storage structures and data fragmentation as well as physical properties of the resulting data streams (e.g. sorting, partitioning etc.).

As shown until now, apart from the decision on where to place a *send-receive* pair, several parameters related to the data river paradigm can influence the performance of a parallel execution plan. We have investigated these aspects by using a 100 MB TPC-D database running on a cluster of 4 Sun-ULTRA1 workstations with 143 MHz Ultra-SPARC processors and, for comparison purposes, also on a Sun-SPARC20 SMP, having 4 processors, each 100 MHz, and 4 disks.

As shown in Fig. 3.3, we have used a simple query consisting of the parallel scan of the LINEITEM table, performed by the *pscan* operator, followed by a grouping. In this scenario, the consumer subplan is slower than the producers. Since the LINEITEM table is physically partitioned onto 4 disks, the parallelizer has chosen to set the degree of parallelism of the scan operator to 4 as well. We have executed this query repeatedly while modifying the *receive* modes. For each *receive* mode, the dataflow parameters of the *send* operator have been modified as well, as listed in the figure. Note that in the case of an ASYNCH *receive* the output is not sorted; we have only listed it here for comparison purposes. The numbers in parenthesis show the buffer quota in pages for the WAIT and WRITEOUT strategies. The SUBPAGE option indicates that instead



a) Single-user mode b) Multi-user mode

Fig 3.4: Performance using different dataflow parameters (SMP)

Table 3.1: Average response times for TPC-D query set (ms)

	writeout (1)	writeout (10)	nobuf	wait (1)	wait (10)	wait(10 2 users	nobuf 2 users
SMP (4 x 100 MHz)	38713	38069	36615	35284	34824	61711	60123
Cluster (4 x 143 MHz)	23337	22472	22623	23794	23034	43724	41275

of a whole page, only a fraction of it (here 1/4) is used as dataflow granule. This leads to a finer granularity for the management of intermediate results, but also to more overhead due to the increased number of messages. In Fig. 3.4a the same test series are performed on the SMP machine, in Fig. 3.4b additionally using a noise factor as a constant load on the same system environment (i. e. disks and processors) simulating a multi-user scenario.

In real-life queries, the plans are more sophisticated and usually split into more blocks using also replication. In order to analyze these scenarios as well, we have executed 16 different parallel TPC-D queries on the above mentioned platforms. In this test series, all aspects related to intra-query parallelism, i.e. degrees of parallelism, *receive* mode parameters etc. are kept constant while changing the dataflow parameters. The average response times for the whole query set can be found in Tab. 3.1.

The results in Fig. 3.3 and Fig. 3.4 show that the ASYNCH *receive* mode produces the best performance, except for the WRITEOUT strategy on the SMP platform, when only one buffer page is allocated for intermediate results. This is due to the fact that all producer instances, in this scenario being faster than the consumer instance, have to spool their intermediate results to disk as soon as a page is filled up, thus causing a high disk contention. However, although the ASYNCH *receive* mode shows good results for all the other dataflow parameters analyzed, it is only usable if the final result doesn't have to fulfill any specific physical properties, as e.g. sorting. This property is only guaranteed by the other two (demand-driven) *receive* modes. Here, an advantage of the SEQ *receive* strategy over the MERGE one can be observed. This aspect is especially obvious for the workstation cluster as the MERGE strategy generally involves a slightly increased communication overhead. However, the SEQ case requires a range-based partitioning on the sorting attribute. This constraint is not necessary in the case of the MERGE strategy. Hence the *sort* operator can be bundled into a block also with operators that require a different partitioning strategy than a range partitioning on the sort attribute (the details on block construction can be found in Section 3.5). Although several systems [Gr95] use only one (usually hash-based) partitioning strategy, the above discussion also confirms that various strategies should be used for different problems.

As for dataflow control, these tests show that a synchronized communication between subplans, using the WAIT option, is always favorable, especially if enough buffer pages are available. For this strategy, the difference between demand- and data-driven *receive* modes is also minimal. So why not use this dataflow strategy, combined with a demand-driven *receive*, without being obliged to find solutions for overflowing buffers as in the case of an ASYNCH *receive* and disk contention as in the case of materializing *send* operators? The answer is that although the demand-driven approach has the least synchronization overhead and produces correct outputs

w.r.t. physical properties, it sometimes can produce deadlocks (see Section 3.4). One solution to this problem is the usage of an ASYNCH *receive* in parts of the PQEP where this is possible, or the usage of materializing *send* operators, like WRITEOUT or NOBUF. Another important remark is that in the simple example query we used for this test, the intermediate results have been read by a single consumer, making efficient garbage collection possible (*readonce* option). In this case the WAIT strategy eliminates completely the need for disk access and thus leads to a good performance. If replication or multi-query optimization is used, more consumers use the same intermediate results, hence these have to be materialized. In this case, the use of the WAIT option doesn't bring any remarkable benefits (see Tab. 3.1).

As expected in the example given in Fig. 3.3 (and Fig. 3.4) showing a slow consumer, the NOTIFY strategy has a negative influence on performance, due to the increased number of messages. Hence in such cases the (default) strategy based on polling should be used for communication between the *send* and *receive* operators. Using a finer granularity for communication (i.e. the SUBPAGE option) is beneficial e.g. on the SMP platform in a multi-user environment (Fig. 3.4b); when the intermediate results have to be communicated over the network, larger units are preferable (Fig. 3.3). From the two materializing *send* strategies, NOBUF seems to be a good compromise when WAIT cannot be used because of deadlocks. However, the tests in Fig. 3.3 and Fig. 3.4 have been performed in an environment with a relatively large database buffer (1500 pages) that could be exploited by the NOBUF strategy. Column 2 and 3 of Tab. 3.1 show that the WRITEOUT strategy leads to a comparable performance even though having less buffer pages. This is due to the FIFO replacement strategy.

3.3.6 Recommendations for the Parametrization of Data Rivers

The results of our performance analysis can be described by a comprehensive set of parameter combinations that are recommended for specific situations. These recommendations, summarized in Tab. 3.2, are first taken into account by the parallelizer. As already mentioned, this component takes the cost model and system statistics into account to find out which forms of parallelism provide optimal response time combined with minimal resource utilization. However, the outcome of this phase is still a preliminary result, as certain parameters can be further modified by the QEC at runtime. This is important, because the results in this section show that different parameters or parameter combinations are favorable, depending on the runtime environment, such as the platform where a block is scheduled, available database buffer, current number of users etc. With this concept of parametric PQEPs we achieve the necessary flexibility to be able to adapt in the best possible ways to these runtime situations.

Tab. 3.2 is structured as follows. The first column separates situations for which the other columns determine the corresponding parameters. First, we discuss the scenarios where no special physical properties are requested for the final stream. After this we present various situations that require a sorted result. We continue with the parameter settings for replication. Furthermore, it is worthwhile to identify the situations where the cost of the producer is higher than that of the consumer as well as multi-user contexts in SMP environments.

Focusing first on the *receive* modes, we can conclude from Tab. 3.2 that the ASYNCH mode is

Table 3.2 Parameter combinations recommended for specific situations

Situations		Receive Mode Parameter	Dataflow Mode Parameter	Data Partitioning Parameter
Resulting stream unsorted	• No special physical properties of resulting or intermediate data streams	ASYNCH	Buffer pages for communication • sufficient available: WAIT • else: NOBUF	ROUND-ROBIN
	• No special physical properties of resulting stream • Intermediate streams require partitioning on certain attributes	ASYNCH	Buffer pages for communication • sufficient available: WAIT • else: NOBUF	HASH
Resulting stream sorted	• No deadlock possible • Partitioning on sort attribute possible	SEQ	WAIT	RANGE (on sort attribute)
	• No deadlock possible • Partitioning required on attributes R, different from sort attribute	MERGE (on sort attribute)	WAIT	HASH (on attributes R)
	• Deadlock possible • Partitioning required on attributes R, different from sort attribute	MERGE (on sort attribute)	Buffer pages for communication • sufficient available: WRITEOUT • else: NOBUF	HASH (on attributes R)
	• Deadlock possible • Partitioning on sort attribute possible	SEQ	Buffer pages for communication • sufficient available: WRITEOUT • else: NOBUF	RANGE (on sort attribute)
Replication		ASYNCH	Buffer pages for communication • sufficient available: WRITEOUT • else: NOBUF	REPLICA-TION
Cost of producer higher than that of consumer		-	in addition NOTIFY option	-
SMP, multi-user environment		-	in addition SUBPAGE option	-

favorable for almost all situations. Exceptions are provided only by scenarios that require a sorted output.

The dataflow mode parameter in Tab. 3.2 is set by the QEC component depending in some situations on the available number of buffer pages. If there are enough pages available, a dataflow mode that shows good performance in combination with a sufficiently large buffer quota should be chosen. Such modes are e.g. WAIT for deadlock-free plans or WRITEOUT for PQEPs bearing a possible cyclic data dependency. Otherwise, if the number of available buffer pages at the beginning of the execution is insufficient, the NOBUF strategy is advisable, as this can make use also of pages that are freed at runtime, possibly also by other queries.

As for the partitioning parameters, we recommend a round-robin strategy for all cases where this is possible, as this partitioning does not result into any data skew. In all other situations, the partitioning has to keep track of the actual parallelization strategy.

Our practical experience showed that a buffer quota is necessary in any case. The size of the quota has to be adapted according to general system state, i.e workload, and in accordance to the difference of the processing rate between consumer and producer blocks. Exactly for that reason it is important that the parallelizer builds blocks having similar processing rates. Our experiments showed that if both producer and consumer processing rates are in the same range, a buffer quota of less than 10 proves to be sufficient.

In the following sections, we will concentrate on more complex parallel query execution plans that employ the data river paradigm. Thereby, we introduce and discuss indispensable measures related to this concept that have a serious impact on applicability and efficiency.

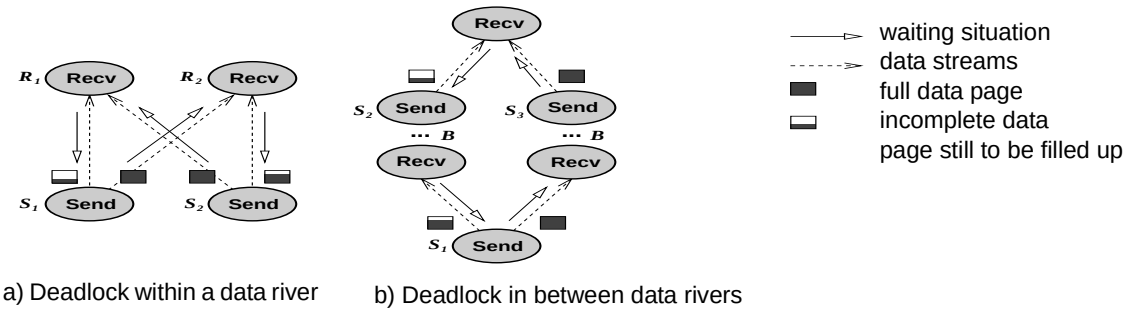
3.4 Deadlock Situations Caused by Intra-Operator Parallelism

In MIDAS, as well as in other PDBMS (e.g. DB2 UDB [BV98]), the intermediate result tuples to be communicated between execution units are bundled together to granules containing multiple rows. This is to reduce communication overhead. The dataflow granule can be e.g. a page, as in MIDAS. As shown in Section 3.3.5, a synchronized communication for intermediate results, using the WAIT option for the *send* operator combined with a demand-driven *receive* mode, e.g. MERGE or SEQ, is the most favorable parameter combination, if certain physical properties (e.g. sorting) of the final data stream are important. In this case, flow control has to stall producer instances to prevent buffer overflow, as well as consumer instances until pages are filled up. These waiting situations can produce deadlocks when intra-operator parallelism is used. The reason is that intra-operator parallelism abolishes the tree structure of a QEP and thus can generate cycles.

In the following, we will present some deadlock scenarios, as well as solutions to resolve cyclic data dependencies. These solutions are based on dedicating selected *send* nodes along a data cycle as *materializing*. To avoid speedup degradation due to disk contention, this has to be done in a cost-based manner, taking into account intermediate result cardinalities and operator costs. Please note that even a materializing *send* node results to I/O only if there is not sufficient buffer available. The corresponding strategy that is incorporated into the parallelizer will be detailed in Section 4.6.

3.4.1 Deadlock Within a Data River

Situation: A demand-driven merging order within the *receive* operator is necessary if certain physical properties (e.g. sorting) of the final data stream are important. Suppose that in Fig. 3.5a R_1 and R_2 are two instances of such a *receive* operator. For simplicity reasons, assume that the corresponding *send* operator has also two instances S_1 and S_2 performing a data repartitioning.



a) Deadlock within a data river b) Deadlock in between data rivers

Fig 3.5: Deadlock scenarios for unary operations

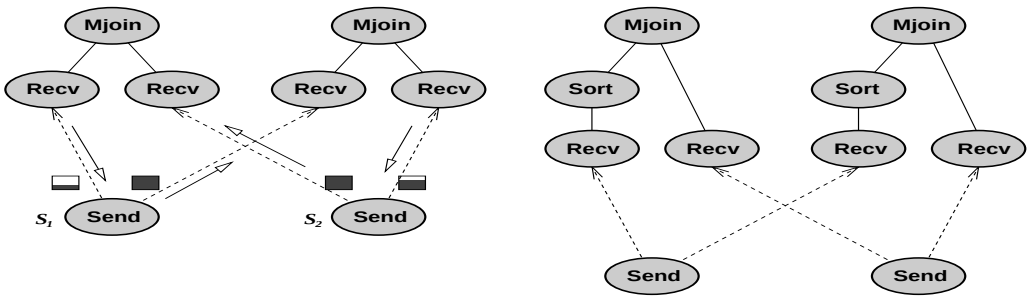
Thus, according to Section 3.3.3, there are *2 x 2 = 4* data streams involved, depicted in Fig. 3.5 as dotted lines. We further assume that the available buffer per data stream is one page. Due to data skew, at a given point in the query execution, some of the buffer pages are already filled up when others are still incomplete (cf. Fig. 3.5a). However, if the buffer corresponding to the current output tuple of the producer is filled, it is stalled by the flow control, unless a materializing *send* operator is used. In the example from Fig. 3.5a S_2 has already filled up the buffer corresponding to R_1. Suppose that according to the partitioning function the subsequent tuple is assigned to the same partition. In this case, it should be added to the buffer corresponding to R_1 as well. However, since this buffer is full, S_2 cannot continue with its evaluation. On the other hand, in a demand-driven *receive* mode the reader also blocks until the sender has filled up a full page or has finished. In Fig. 3.5 the continuous arrows depict waiting situations from the arrow bottom to its head. As shown in Fig. 3.5a, due to a pre-defined merging strategy, instance R_1 waits for S_1 instead of processing the data stream corresponding to S_2. Hence, a cyclic data dependency comes to existence.

Solution: As shown by the arrows in Fig. 3.5a for instance a 2:2 repartitioning can produce a deadlock, if R_1 and R_2 are *receive* nodes for which the merging strategy of the data streams is demand-driven and the *send* nodes realize the WAIT strategy. Generally, in order to avoid deadlocks with *N:M* repartitioning using a demand-driven *receive* strategy, the usage of non-blocking *send* nodes that materialize the intermediate results in case of an overflow is necessary. Hereby it is sufficient to modify only one of the *send* nodes S_1 or S_2 to materializing.

3.4.2 Deadlocks in Between Data Rivers

Situation: In Fig. 3.5b a partitioning is performed, and later on the data is merged by a central *receive* node. Here, global data dependencies have occurred, producing a cycle. It is important to note that the problem doesn't occur if in the intermediate block instances (marked with *B* in Fig. 3.5b) there is at least one so-called `blocking boundary`. These refer to particular locations within query execution, where the complete intermediate result table has to be derived before the next operation can start. A frequently used blocking operator is e.g. the *sort* node. Blocking boundaries prevent from cycles, because at the time when the topmost *receive* operator starts merging, the *send* operator S_1 has already processed all of its input.

Solution: In order to avoid this type of deadlocks, also materializing *send* nodes are necessary. Here, changing the mode of S_1 or S_2 and S_3 breaks up the data dependency. However, if it



a) Deadlock caused by a merge join b) Sort nodes prevent from deadlock situation

Fig 3.6: Deadlocks situations for binary operators

becomes necessary to introduce materializing *send* nodes, it is less expensive to change the mode of replicating nodes, as multiple readers usually impose a materialization of the intermediate results anyway. If the overall parallel plan doesn't contain any replicating nodes, *send* operators that delimit inner subplan instances (e.g. S_2 or S_3) should be modified, because they usually operate on smaller data sizes as compared to a central partitioning *send* node (e.g. S_1).

3.4.3 Deadlocks Caused by Binary Operators

Situation: Binary operators with a predetermined processing order, as e.g. the *merge join* in Fig. 3.6a, are able to produce similar global dependencies as a central merging *receive* node discussed before. In this situation the deadlock is also among data rivers, but in contrast to Section 3.4.2, these are on the same level in the QEP and being consumed simultaneously by the inputs of the *merge join* operator. Binary operators that process one input stream entirely after the other (e.g. the *hash join*) don't raise such problems. A situation like in Fig. 3.6a doesn't occur either if the sorting of at least one input is necessary (Fig. 3.6b). As shown above, the *sort*, being a blocking operator, implicitly resolves the data dependency. In general, deadlocks caused by binary operators are possible, if both inputs are partitioned or replicated by non-materializing *send* nodes and none of the inputs is processed either by a blocking operator (e.g. *sort*) or entirely before or after the other.

Solution: In the depicted situation it is sufficient to modify only one of the two nodes S_1 or S_2 to a materializing *send* node. Generally the node selection is done according to intermediate result sizes.

3.5 Reducing the Number and Size of Data Rivers

Prior work on parallel execution and cost models as well as scheduling relies on the assumption that the QEP is *coarse-grain*, i.e. the parallelization overhead for each operator exceeds only up to a specific factor the total amount of work performed during the execution of the operator [GI97], [GGS96], [DG92]. However, this strategy based on one-operator granules for parallelism is suboptimal for practical database execution plans. This is due to the fact that a data

Table 3.3 Influence of block construction on performance

(average over 16 TPC-D queries is reported)	Small Blocks	Combined Blocks
Avg. Execution Time (ms)	41686	23002
Avg. Speedup (vs. Sequential Execution)	2,4	4,34
Avg. Number of Execution Units	13,625	7,5

river is necessary for each operator in the QEP, even if specific operators don't contribute significantly to overall performance improvements. This leads to a loss of efficiency w.r.t. communication costs and resource utilization.

Our response to this problem is *cost-based block building*. This strategy accounts for operator costs, selectivities, and intermediate result sizes to construct *coarse-grain blocks*, i.e. to perform several operators within a single execution unit. The goal of this block-building is to achieve mutually adjusted processing rates among communicating, i.e. neighboring blocks. If the rate at which tuples are sent to an execution unit is much higher than the rate at which tuples can be processed, the communication buffer can overflow, forcing the sender to block. On the other hand, if the rate at which tuples are received is much too low, the corresponding execution unit will frequently be idle and will waste non-preemptive system resources, as e.g. memory. Hence, mutually adjusted processing rates are prerequisite to efficient pipelining [MD95]. Additionally, through block construction intermediate result materialization and inter-process communication between operators, i.e. data rivers, can be avoided. This implies savings in main memory or even I/O costs.

Building blocks by taking the sizes of intermediate results into consideration and performing the required block partitioning where it is most favorable, is an important instrument to reduce the size and number of data rivers and implicitly reducing communication overhead as well.

To validate our approach, we have processed 16 different parallel TPC-D queries on the same workstation cluster as in Section 3.3.5. In one test series small blocks have been used, in some cases comprising only one operator, in the other series these blocks have been combined, adapting the degree of parallelism of the final block accordingly. The results, summarized in Tab. 3.3, show that cost-based block building combined with adapted degrees of parallelism throughout the PQEP contributes significantly to obtain optimal speedups, in the same time reducing resource utilization.

3.6 Reducing the Number of Data Streams and Execution Units

In several situations, the insertion of a data river is not imposed by cost-based decisions, i.e. to reduce response time by means of parallelization. For instance, if the partitioning strategies of two low-cost operators are different, a data river has to be inserted merely for repartitioning purposes, although the costs don't justify a parallel processing of the operators. To reduce the increased resource consumption in this case, i.e. number of execution units and the communi-

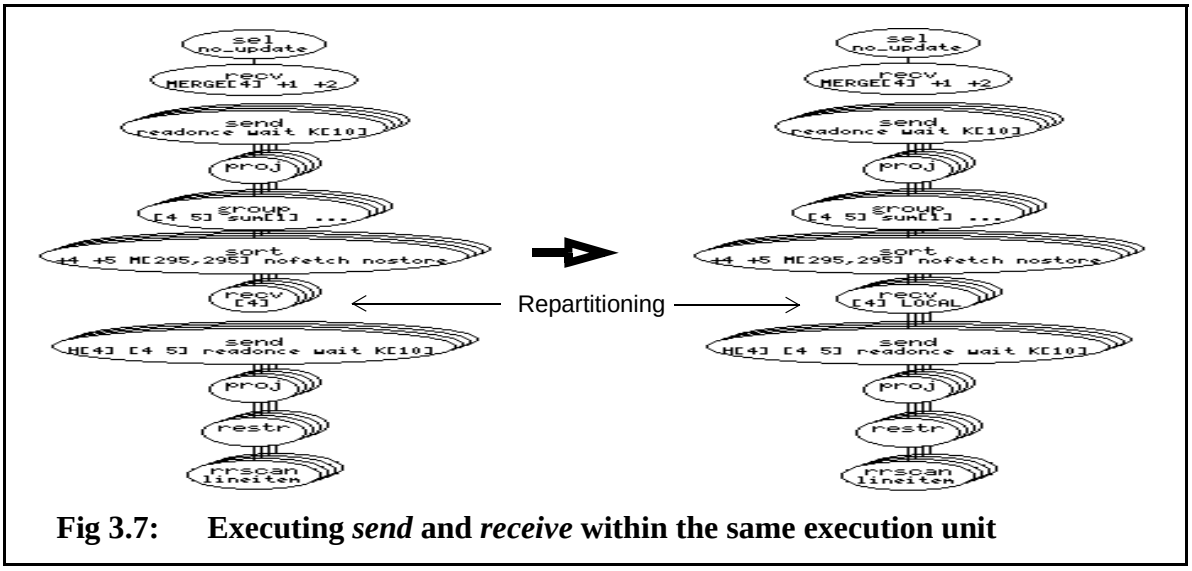


Fig 3.7: Executing *send* and *receive* within the same execution unit

cation costs, we have implemented the possibility to execute the *send* and *receive* operators within the same execution unit. Hence, the need for a data stream between *send* and *receive* operator instances that are processed within the same execution unit disappears. With this strategy, we deliberately dispense with parallel execution at some points which don't contribute to overall performance gains, thus saving resources. In this way a good trade-off is achieved, especially for intra-query parallelism in a multi-user environment. We have adapted the *send/receive* nodes to deal with this special situation as well. Please note that this is in contrast to Section 3.5, here we have to deal with repartitioning that generally prohibits block combination.

As an example, consider TPC-D query Q1 (Fig. 3.7), parallelized for 4 processors and 4 disks. Here, a degree of parallelism of 4 is used throughout the PQEP. However, the grouping requires an attribute-based partitioning, in this case hash, indicated by the parameter (*H[4] [4 5]*). Thus, a repartitioning is necessary, realized by a data river of 4 x 4 = 16 data streams, leading to altogether 9 execution units. By bundling together *send* and *receive*, the number of execution units is reduced to 5. In addition, 4 data streams get substituted by communication within an execution unit, as visualized by the lines connecting inner-block operators. Hence the data river is now constituted only of 16 - 4 = 12 data streams.

The validation of this approach, using all applicable TPC-D queries and the same test scenario as before, can be found in Tab. 3.4. The results clearly show that this strategy leads to an increased efficiency, especially in a multi-user environment.

Table 3.4 Benefit of execution unit combination

(average over applicable TPC-D queries)	Separate Execution Units	Combined Execution Units
Avg. Number of Execution Units	6,6	4,7
SMP avg. Execution Time (ms) - Single User	38985	35565
SMP avg. Execution Time (ms) - 2 Users	60136	52699
Cluster avg. Execution Time (ms) - Single user[a]	30171	26765

a. While comparing the results please note that the workstations in the cluster have more processing power than the SMP.

3.7 Related Work

Important research activities have been done in the area of parallel query execution. However, most previous work uses simplifying assumptions or concentrates on specific architectures and operator types that don't cover the requirement catalog coming from real-life application scenarios. In addition, language extensions or extensions to the database engine itself, as supported by object-relational DBMS, are hard to accomplish. Thus parallelization in many cases is restricted to join orderings in combination with restricted forms of parallelism [Has95] or is based on heuristics, as e.g. in XPRS [HS93]. Here, the exchange of intermediate results is done through temporary tables, but there is no discussion on how to organize these tables, which is one focus of this chapter. Other approaches are highly specialized for specific, e.g. main-memory [BDV96] architectures or certain operator types, like e.g. hash joins [SD90, LC+93, ZZS93, WFA95].

The most similar approach to the MIDAS execution system in a shared-memory environment is Volcano [Gr94]. Here, intermediate results are managed by a buffering control mechanism based on semaphores. Volcano adopts only a data-driven dataflow between the subplans, while having a demand-driven approach within a subplan. Processes needed for the execution of subplans are forked dynamically through the *Exchange* operator. Hence, this results also into an architecture-specific solution. To reduce the number of processes, the *Exchange* operator was extended to run within a process' operator tree (similar to our approach described in Section 3.6), thus making inter-process communication demand-driven, too. The author claims this makes flow control obsolete. However, we cannot support this statement, as in the course of redistribution, the *Exchange* operator within a process also provides input for other processes, that can overflow. As shown in 4.1, this leads to a high deadlock probability, an issue that is not discussed in the Volcano paper. Furthermore, no experimental results are presented, which makes the developed techniques hard to be analyzed or to be compared to other approaches.

The Gamma prototype [DG92] uses the *split* and *merge* operators in a similar way as MIDAS uses the *send* and *receive* operators. Apart from this, the query execution system of Gamma is quite different from ours, as it doesn't allow multiple operators to be processed within the same process. In this model [Gh90], there is a generic process template that can receive and send data and can execute exactly one operator at any point of time. Later work [BF+95] has shown that this mechanism generates too many processes. In addition, network I/O is the only means of obtaining input and delivering output for operators, implying high interprocess communication for queries containing multiple operators.

The assumption that each operator is parallelized independently, the output of an operator being always repartitioned to serve as input to the next one is the basis of more recent models as well [GI97, GGS96]. However, our results in Section 3.5 show that the optimal degree of parallelism of a set of operators differs from the optimal degree of parallelism of each separate operator, as in this way repartitioning can be avoided and larger blocks can be constructed, thus saving resources.

As for commercial databases, Informix Dynamic Server [Zo97, Inf98] uses some Volcano concepts for the query execution system. A special operator is used to pipe intermediate results from one set of operators to initiate another set of operators. A special dataflow manager is used

to provide flow control and avoid spoolings to disk and probably also deadlocks, an issue not discussed there.

In DB2 UDB [Ibm98, JMP97, BF+95], the intermediate results are managed by so-called table queues that connect subsections of a plan. Only one (hash-based) redistribution strategy is used. But the major difference to MIDAS is that there is no synchronization between the sender and the receiver. The disadvantage of such a pure data-driven approach is that a special communication subsystem has to be implemented, that guarantees the correct order of arrival and the materialization/dropping of messages. Our measurements in Section 3.3.5 also show that the communication costs for such an approach are higher than in a synchronized communication. That is why in MIDAS we combine both methods, using the demand driven approach whenever possible and materializing only to avoid deadlocks. Another difference to MIDAS is that the degree of parallelism of an operator is mainly determined by the partitioning of its inputs, while in MIDAS this is calculated according to the actual cost of the operator or the block the operator belongs to.

The Nonstop SQL/MX Database [Tand97] uses *split* and *collect* nodes for data partitioning that are similar to our *send* and *receive* nodes, except that only hash distribution is supported. To our knowledge, there exists no publication describing flow control mechanisms or resource management strategies. The degree of parallelism is chosen according to heuristics, like total number of processors, or number of outer table fragments for hash joins. In Oracle8 [Or99] multiple relational operators can be processed within the same process, but parallelism is limited to a 2-process depth at a time. Another simplifying aspect is that only one pre-determined degree of parallelism is used throughout the plan. This degree of parallelism can be only overridden by system limits and user hints. This approach of a uniform degree of parallelism for the entire PQEP is also adopted by Teradata [BF97], but proves to be suboptimal as confirmed by our experiments and performance analysis.

3.8 Summary

In this chapter we have presented and evaluated our approach to efficient management of intermediate query results in parallel database engines. Intra-query parallelism has been achieved by implementing the *data river* paradigm using communication segments. We have analyzed the importance of certain aspects in this paradigm, like dataflow control, partitioning, and merging of data streams as well as the granularity used for communication. Furthermore, we have shown how this paradigm can be significantly improved by block construction, cost-related degrees of parallelism and combination of execution units. We have pointed out the importance of these concepts towards achieving optimal speedup with minimal resource consumption. We have shown that the approach adopted by most commercial DBMSs of choosing a uniform degree of parallelism for the whole PQEP results in less efficiency, especially in forthcoming applications, like OLAP and parallel object-relational DBMS, where the operator costs in a plan can differ significantly. Additionally, we have presented the deadlock problem which occurs in parallel query execution systems if a pure demand-driven data flow is used.

Chapter 4
TOPAZ
a Multi-Phase Parallelizer

An open problem is how to integrate parallelization with existing optimizer technology. This chapter presents our approach towards achieving this goal. The corresponding strategies are implemented within our parallelizer component, called TOPAZ.

4.1 Introduction

Generally, the responsibility for query parallelization is taken over by the so-called parallelizer. Its task is to come up, without incurring too much overhead, with a parallel query execution plan that exploits various forms of parallelism, thereby achieving maximum speedup combined with lowest resource consumption. Among the most important requirements to be fulfilled by modern parallelizer technology are the following ones:

- *Extensibility:* This requirement is stressed by forthcoming application scenarios. Necessary SQL extensions are dealt with using concepts like user-defined functions [JM98] or designated (internal) operators [NJM97]. Similar issues are treated in the context of (parallel) object-relational database systems [StMo96].
- *Performance:* To conquer the complexity of the parallel search space [GHK92], [LVZ93], accurate pruning techniques are necessary.
- *Granularity of Parallelism:* A QEP consists of operators showing significantly dissimilar processing costs. Currently, cost models as well as parallelization strategies only deal with high-cost operators (also called *coarse-grain* operators [GGS96], [GI97]), as the performance speedup by means of parallelization is most profitable for this kind of operators. However, low-cost operators, if not treated the right way, can deteriorate query execution performance considerably. Hence, a flexible granularity of parallelism that provides a comprehensive treatment of low-cost and high-cost operators is necessary.
- *Economical Resource Allocation:* The maximum speedup obtainable is delimited by the critical path of the best execution schedule for the PQEP. Hence, to limit resource contention, no resources should be allocated to subplans that cannot reduce this measure. This

demand is particularly important in the case of queries that run over longer periods of time (e.g. DSS queries) and in multi-user environments.

- *Comprehensiveness:* In order to generate PQEPs of acceptable quality for all query types it is necessary to take into account all forms of intra-query parallelism.
- *Adaptability*: Hybrid (or hierarchical) system architectures are gaining popularity. However, the development of optimization techniques to exploit their full potential is still an open problem [Gr95], [BFV96].

Obviously, numerous techniques have to be devised and combined in order to meet all of the above listed requirements. As rule-driven optimizers [Lo88], [Gr95] have already proved to be extensible and efficient, we rely on that technology also for parallelization and propose a solution based on a top-down search strategy, called TOPAZ (**TO**p-down **PA**ralleli**Z**er). TOPAZ is based on a fully cost-based decision making. To reduce complexity, it splits parallelization into subsequent phases, each phase concentrating on particular aspects of parallel query execution. Moreover, by means of a global pre-analysis of the sequential plan first cost measures are derived that are used to guide and restrict the search process. TOPAZ combines high-cost as well as low-cost operators into blocks that are subject to coarse-grain parallelism. As shown in the previous chapter, this cost-based block building is important to achieve economical and efficient resource utilization. The rule-driven approach provides for the necessary abstraction to support any kind of operators as well as different architecture types, including hybrid ones. As already mentioned in Section 2.4, TOPAZ solely relies on the widespread concepts of iterators and data rivers common to (parallel) execution models. Hence, it fits as an enabling technology into most state-of-the-art (object-) relational systems. This is additionally supported by the fact that TOPAZ builds upon the Cascades Optimizer Framework [Gr95], which is also the basis of some commercial optimizers, like Microsoft's SQL Server [Gr96] and Tandem's Serverware SQL [Ce96].

In this chapter, we analyze the possibilities of integrating parallelization into established query optimization search engines and point out limitations of the heuristics used in state-of-the art parallelizers. A discussion of related work is given in Section 4.2. The Cascades Optimizer Framework is described in Section 4.3. In Section 4.4 a sample query taken from the TPC-D benchmark is introduced as a running example. In addition, the design of TOPAZ in terms of basic parallelization strategies as well as internal optimization and control measures is presented. In Section 4.5 the main phases of the parallelization task are described. The strategy for deadlock prevention on the parallelization level is presented in Section 4.6. An analysis and evaluation of the technology incorporated into TOPAZ is provided in Section 4.7. Finally, in Section 4.8 a summary completes the chapter.

4.2 Related Work

Query optimizers use continuously improved techniques to simplify the task of extending functionality, making search strategies more flexible, and increasing efficiency [HK+97], [PGK97],

[OL90]. In a parallel context, the search space becomes even more complex as resource utilization has to be considered as well. One way to tackle this problem is to develop specialized solutions for the exploration of the parallel search space. The other approach is to reuse existing sequential optimization techniques and to extend them by special heuristics.

4.2.1 Specialized Parallelization Techniques

The ***two-phase*** approach uses a traditional search strategy for optimization and a specialized one for parallelization of queries. The parallelization task is based in many cases on heuristics [HS93] or it is restricted to join orderings in combination with restricted forms of parallelism [Has95]. As the parallelizer is a separate system, easy reuse of infrastructure or technology improvements as mentioned above is prohibited. In addition, extensibility, as required by object-relational extensions, is limited.

Other approaches propose an integration of optimization and parallelization, but are still highly specialized for specific architectures or certain operator types. Some research concentrated on scheduling of joins that maximizes the effect of specific forms of parallelism. Thus concepts as ***right-deep*** [SD90], ***segmented right-deep*** [LC+93], ***zig-zag*** [ZZS93] and ***bushy*** trees [WFA95] have been elaborated. Although these strategies have achieved good performance for specific scenarios [STY93], they rely on the hash-join execution model and thus cannot be applied in a straightforward way to complex queries holding any kind of operators.

4.2.2 Parallelization Using Traditional Sequential Optimization Techniques

If the optimizer and the parallelizer are integrated, they both use the same search strategy. However, they differ in the exploration of the search space and the size of the portion explored. Due to its exponential complexity, ***exhaustive*** search is only feasible for very simple queries and is implemented in few research DBMSs, mainly for performance comparison purposes. ***Randomized*** algorithms have been mainly tested for join orderings in both the parallel and sequential context [LVZ93], [KD96], showing efficiency only for a high number of base tables. The best-known ***polynomial*** algorithm, the greedy paradigm [LST91], explores only a small portion of the search space, often ignoring some forms of parallelism. Thus, it is likely to fail the regions of good physical operator trees.

The ***bottom-up (dynamic programming)*** algorithm works iteratively over the number of base tables, constructing an optimal physical operator tree based on the expansion of optimal subtrees involving a smaller number of tables. To handle exponential complexity, a pruning function is introduced to realize a branch and bound strategy, i.e. it reduces the number of trees to be expanded in the remaining of the search algorithm. Pruning is achieved by comparing all subtrees which join the same set of relations with respect to an equivalence criteria and then discarding all trees of non-optimal cost.

As already mentioned, in order to tackle the even higher complexity of a parallel context, many PDBMSs use beside dynamic programming pruning also other simplifying heuristics. Thus, in

DB2 Parallel Edition [BF+95], [JMP97], the degree of parallelism of the operators is mainly determined by the partitioning of the inputs. In Teradata and Oracle's Parallel Query Option [BF97], [Or99] it even remains the same for the whole plan.

In ***top-down*** optimizers, such as Volcano [Gr94], Cascades [Gr95] and Columbia [SM+98, Xu98], the complexity explosion is also controlled by pruning. These optimizers compute costs for high-level plans before some lower-level plans are examined. These cost limits are then passed down to optimization of incrementally constructed QEPs and can thus prune plans whose predicted total costs exceed this limit. Some investigations [KD96] have yielded poor performance for top-down optimizers. However, these results referred to the Volcano search strategy, that meanwhile got improved in the new generation of top-down optimizers, especially concerning pruning techniques [Ju99], [SM+98], [Gr95].

W.r.t. parallelization, in Volcano the best sequential plan found is divided into several segments that are bracketed by *Exchange* operators. Please note that in this way parallelization, i.e. the computation of the degree of intra-operator parallelism or the determination of the segment boundaries, is done with another search strategy than that of the Volcano optimizer. Although top-down optimizers are used in other PDBMSs [Ce96] as well, we do not know of any publicly available report on how to decide on parallelization using this type of search engine.

To overcome the shortcomings of each optimization strategy in combination with certain query types, also ***hybrid*** optimizers have been proposed [ON+95], [McB+96]. Thus, region-based optimizers [ON+95] use different optimization techniques for different application scenarios. The EROC project [McB+96] combines top-down and bottom-up approaches in the NEATO join optimizer. Here, the bottom-up search strategy is used to enumerate all join orders and the top-down strategy is used to perform the mapping from logical to physical operators in a parallel environment.

As the optimizers based on bottom-up [Zo97, HK+97, JMP97] and top-down [Ce96, Gr96] search strategies are both extensible [Lo88, Gr95] and in addition the most frequently used in commercial DBMSs, we will concentrate in the following on the suitability of these two techniques for parallel query optimization.

4.2.3 Suitability of Traditional Parallelization Techniques for Upcoming Application Scenarios

In this section, we consider an example that reflects the changing requirements of upcoming applications. Thereby, we analyze the suitability of state-of-the-art parallelization strategies for this example scenario. As already mentioned, most database systems use for optimization as well as for parallelization a bottom-up search strategy. For this reason, we consider this search strategy for our analysis as well. Furthermore, we will also adopt the simplifying heuristics commonly used by bottom-up optimizers in a parallel search space, namely the limitation to a number of "interesting partitionings" that are usually chosen according to the partitioning of the inputs [BF+95, JMP97].

We will compare this traditional parallelization with a strategy where required physical proper-

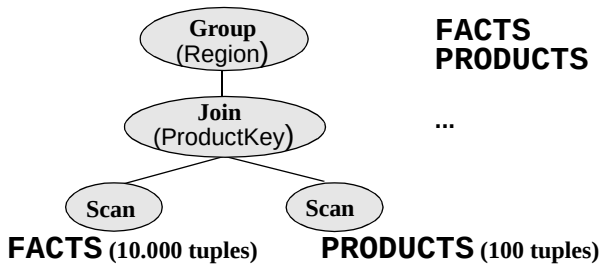


Fig 4.1: The QEP of the example DSS query

ties are determined at an early optimization stage and thus can be used to guide further search space exploration. Such a concept can be easily realized for instance within a top-down strategy as used by Cascades [Gr95]. Therefore, we will consider this search strategy as the second alternative to analyze.

In Fig. 4.1, we take as an example a data warehouse, where extreme data volumes necessarily impose data partitioning and parallelism. The considered star schema consists of a central FACTS table and several dimension tables, PRODUCTS being one of them. The given partitioning attributes that are in most cases key candidates are underlined. The proportion between table sizes correspond to typical state-of-the-art data warehouses [Schn97]. Furthermore, consider a DSS query that for instance analyzes the sales patterns of different regions:

SELECT	Region, *ratio_to_report* (Price * Quantity), *avg* (Quantity), *most_frequent* (Name), *most_frequent* (Manufacturer), *count* (distinct Color), *count* (distinct Type)
FROM	Facts, Products
WHERE	Facts.ProductKey = Products.ProductKey
GROUP BY	Region

For simplification, we have expressed some aggregations by functions. These can be UDFs or SQL extensions, as proposed already by some database vendors [Inf98, Re98]. The query is translated into a QEP, as depicted in Fig. 4.1. Grouping as well as any specified aggregation have been combined into one operator (*Group*). Depending on the concrete DBS this can be realized by a single dedicated operator, e.g. similar to a *DataCube* [GB+96], or a single UDF, or even a bunch of operators. In any case, if this operator gets parallelized, its input has to be partitioned according to the *Region* attribute. However, if the aggregation doesn't refer to a *holistic* [GB+96] function (e.g. *most_frequent*) or if it doesn't hold the *distinct* parameter, then a specific optimization based on a two-phase parallelization[1] is possible. Since our example contains both, this optimization is prohibited. Furthermore, assume that all attribute sizes are equal (10 bytes), the degree of partitioning for the FACTS table is 12 (one partition for each month) and 2 for the PRODUCTS table.

4.2.3.1 Traditional Parallelization

First, the execution of the join is optimized. As the two relations are not partitioned on the same attribute, the following possibilities are considered:

1. First, local aggregations are computed, each instance covering a portion of the data. Second, the local aggregates are merged to produce the global result [SN95].

1. Replicate the PRODUCTS table to each partition of the FACTS table.
   The transmission of the 7 participating attributes makes up a volume of:
   7 x 10 (bytes) x 12 (partitions of the FACTS table) x 100 (tuples) = 84.000 bytes.
2. Replicate the FACTS table.
   This implies the transmission of the 3 needed attributes, *ProductKey, Quantity* and *Region:*
   3 x 10 (bytes) x 2 (partitions of the PRODUCTS table) x 10.000 (tuples) = 600.000 bytes.
   This clearly exceeds the costs of variant 1. Additionally, in this case the degree of parallelism is only 2, implying also higher local processing costs for the parallel join instances.
3. Repartition both tables according to the join attribute *ProductKey.*
   The quantity to be communicated in this case is:
   3 x 10 (bytes) x 10.000 (tuples) + 7 x 10 (bytes) x 100 (tuples) = 307.000 bytes.

Next, the aggregation has to be optimized. Its input has to be partitioned according to *Region*, a condition that is not fulfilled by any of the above mentioned variants. Thus, the join result has to be repartitioned. This implies the transmission of 9 attributes (7 coming from the PRODUCTS table as well as *Quantity* and *Region*):

9 x 10 (bytes) x 10.000 (tuples) = 900.000 bytes.

Adding also the communication needed for the join, considering the best solution (Variant 1), the overall transmission volume is:

84.000 + 900.000 = 984.000 bytes.

4.2.3.2 Parallelization by Taking into Account Required Physical Properties

Since in this example the *group* operator is the most costly one, first this node gets parallelized, imposing as required physical property the partitioning of its input according to *Region*. This is considered when parallelizing the join operator. Thus, beside the variants from above the following is also taken into account:

4. Repartition the FACTS table according to *Region* and replicate the PRODUCTS table.
   In this case the communication overhead due to the repartitioning of the FACTS table is:
   3 x 10 (bytes) x 10.000 (tuples) = 300.000 bytes.
   This is independent of the degree of parallelism of the join operator. To have the same local processing costs as in variant 1, we consider this degree as being equal to 12. Thus, the PRODUCTS table has to be replicated to 12 partitions, implying the same communication overhead as in variant 1, i.e. 84.000 bytes.

In spite of the higher local communication costs needed for the join, variant 4 allows the execution of the aggregation without additional repartitioning costs. Thus, the final plan will have an overall communication that averages only to:

300.000 + 84.000 = 384.000 bytes.

4.2.4 Summary

Please note that bottom-up optimizers can also be extended to find the plan presented before, by

e.g. considering also other partitioning strategies than the “interesting partitionings” determined by the inputs. In [GHK92, LVZ93] solutions were proposed by modifying the equivalence criteria and extending the cost model. However, this reduces the effectivity of pruning.

In this example we have shown that by using physical properties derived at certain stages of top-down optimization as conditions for forthcoming stages, better parallel plans can be obtained. Generally, some crucial decisions in the parallel context refer to *physical* properties, as e.g. partitioning, degrees of parallelism and usage of resources, that have to be chosen in a way to guarantee overall efficiency and to minimize resource contention.

Given the above, it is favorable to come up very early with physical tree solutions, whose cost estimates can be used to perform a global plan analysis and to guide further parallel search space exploration. Thus, beside the quality of the plans also the performance of the optimization itself can be improved considerably. As mentioned above, this requirement is satisfied for instance by a top-down search engine, as Cascades. Hence, we decided to use this strategy for the TOPAZ parallelizer as well.

For the MIDAS project, it was important to first concentrate on the strategies needed to achieve efficient intra-query parallelism. Hence, for a first version of TOPAZ we decided to have as input a complete sequential physical plan that is generated by the TransBase optimizer. Later on, this component has been replaced by a top-down sequential optimizer (see Chapter 6). Thus, optimizer and parallelizer use the same search strategy, i.e. Cascades, but explore different search space regions with different rule sets.

In the following, we present the Cascades Optimizer Framework and later on we will concentrate on the parallelization effort.

4.3 The Cascades Optimizer Framework

The *Cascades Optimizer Framework* or shortly *Cascades* is a tool for the development of extensible, rule-driven and cost-based optimizers. It has been originally developed by Goetz Graefe at the Oregon Graduate Institute and extended by Leonard Shapiro and his group at the Portland State University. It achieves a substantial improvement over its predecessor Volcano in functionality, usability, and robustness without giving up extensibility.

The aim of Cascades is to support different data models as well as different query processing environments. In order to achieve this goal, it differentiates between the *model* and the *search engine*. The search engine defines the interfaces and provides the code to expand the search space and to search for the optimal plan. The model component describes the DBMS on which the queries run, and lists the equivalence transformations (rules) which are used by the search engine to expand the search space.

The *database implementor (DBI)* doesn’t have to take care of the internal features of the search engine. Instead, the DBI is in charge of developing the model that reflects the characteristics of the DBMS and the execution environment. Thus, the model incorporates the operators, properties of subplans, rules as well as a cost model. The advantage of such a strict sepa-

ration lies in the fact that the DBI can support different data models with the same search engine. This feature has been used in MIDAS to develop different models for the optimization and parallelization part.

4.3.1 Anatomy of the Cascades Optimizer Framework

In the following, we will present the Cascades Optimizer Framework and introduce some of the fundamental concepts of query optimization that are necessary for further understanding.

4.3.1.1 Physical and Logical Operators

Cascades models can be broken into two pieces, the *logical model* and the *physical model*. The logical model relates to the semantics of the database system, i.e. *what* is to be computed. The physical model relates to data structures, implementation algorithms and storage devices etc., i.e. *how* it is to be computed. Together the logical and physical models form a particular Cascades model which represents both the logical and physical aspects of the corresponding DBMS.

The logical model incorporates the *logical operators*. These are high-level operators that specify data transformations without specifying the physical execution algorithms to be used. Each logical operator takes a fixed number of inputs (which is called the *arity* of the

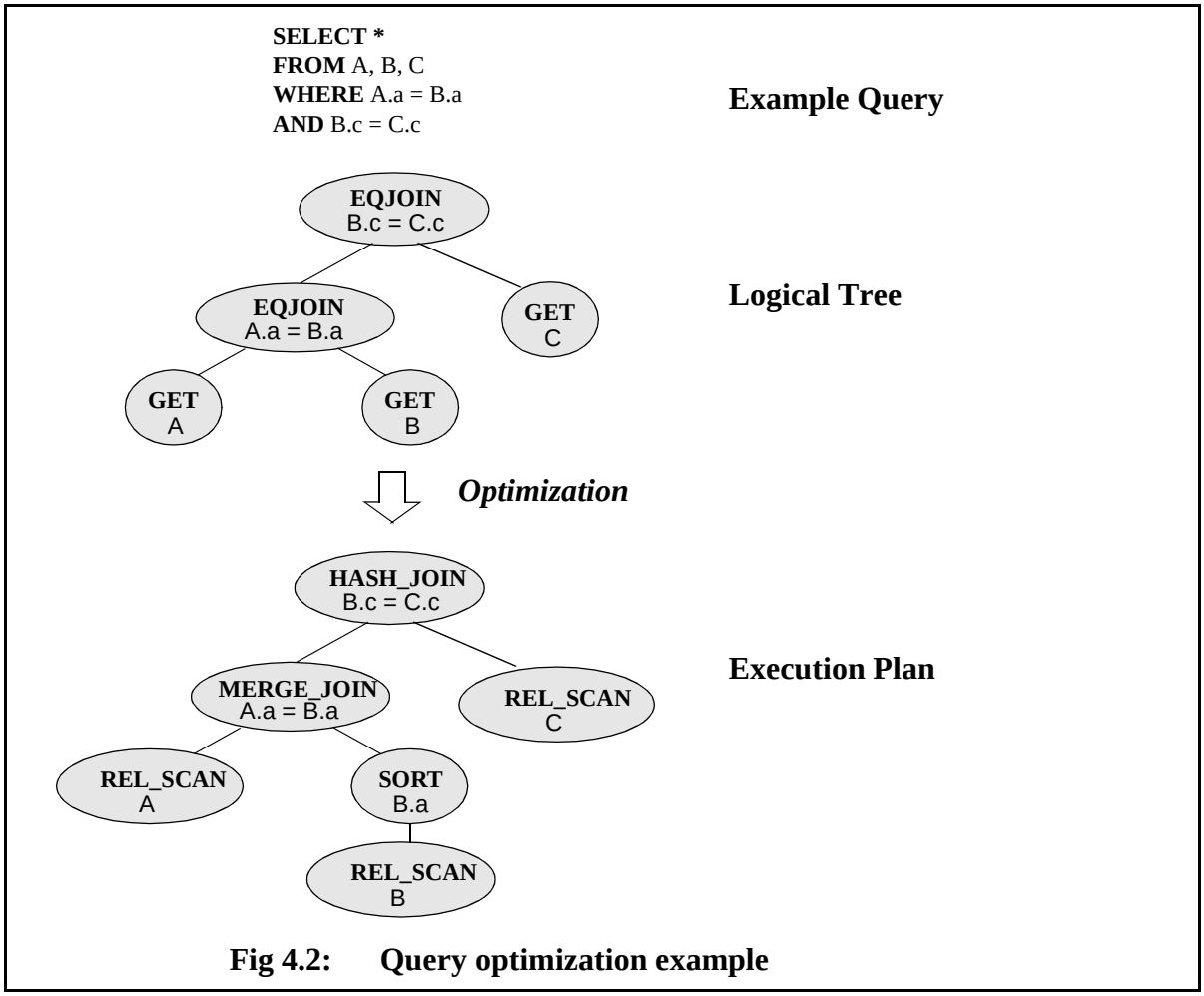


Fig 4.2: Query optimization example

operator) and may have parameters that distinguish the variant of an operator. Two typical logical operators are GET and EQJOIN. The GET operator has no input and one argument, which is the name of the stored relation. GET retrieves the tuples of the relation from disk and outputs the tuples for further operations. The EQJOIN operator has two inputs, namely the left and right tables to be joined, and one argument for the equality condition(s). This argument is expressed as ordered sets of join attributes relating to the left and right tables.

Physical operators represent specific algorithms that implement particular database operations. It is possible to use several physical execution algorithms for the implementation of a given logical operator. For instance, the EQJOIN operator can be implemented using *hash*, *sort-merge* or other algorithms. These specific algorithms can be implemented in different physical operators. Thus, two common physical operators are HASH_JOIN, which implements the *hash-join* algorithm, and MERGE_JOIN, which implements the *sort-merge join* algorithm. The typical algorithm for the GET logical operator is scanning the table in stored order, which is implemented in another physical operator REL_SCAN. Like logical operators, each physical operator also has a fixed number of inputs (which is the arity of the operator), and may have parameters.

The input of Cascades is a *tree of logical operators,* representing the initial query. Replacing the logical operators in a query tree by the physical operators which can implement them gives rise to a tree of physical operators which is nothing else but the *execution plan* for the given query. Fig. 4.2 shows a query optimization example. Here, the underlying cost model has imposed the implementation of the upper EQJOIN operator as a HASH_JOIN, while the bottom join has been implemented as a MERGE_JOIN. In this example the optimized physical plan preserves the join order from the logical tree. However, please note that a modification of the join order (or more generally the execution order) is possible as well. As shown by the memo structure that will be introduced in the following section, e.g. the possibility of performing a join between the tables *B* and *C* has also been evaluated. However, this alternative has been dropped because of higher processing costs.

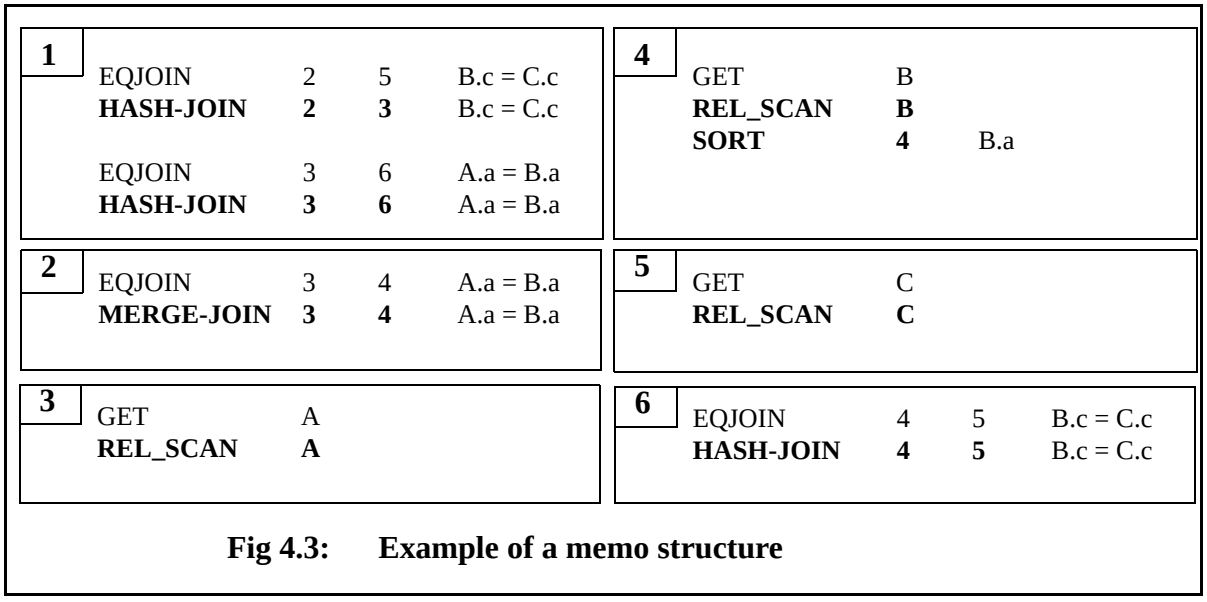


Fig 4.3: Example of a memo structure

4.3.1.2 The Memo Structure

The search space is represented by the so-called *memo structure*. Cascades uses *expressions* to represent subplans. An expression consists of an operator plus zero or more input expressions. An expression can be logical or physical based on the type of its operator. Expressions are *logically equivalent* if they have the same *logical properties*, i.e. the same cardinality, the same schema information etc.

In Cascades, a set of logically equivalent expressions define a *group*. To save space, the search space, i.e. memo structure, is represented as a set of groups, where groups take some other groups as input. There is a top group designated as the final group, corresponding to the result from the evaluation of the initial query. Fig. 4.3 shows the memo structure for the example query in Fig. 4.2. Physical operators are emphasized. The numbers following the operators denote the input groups. Each line ends with the parameters of the corresponding operators.

4.3.1.3 Rules

Rules are used by Cascades to generate the logically equivalent expressions of a given initial query. A rule is a description of how to transform an expression to a logically equivalent expression. A new expression is generated when a rule is applied to a given expression. Thus, the optimizer uses rules to expand the initial search space and generate all the logically equivalent expressions of a given initial query. Rules are part of the model and it is the task of the DBI to specify them.

Each rule is defined as a pair of *pattern* and *substitute*. A pattern defines the structure of the logical expression on which the rule can be applied. A substitute defines the structure of the result after applying the rule. When expanding the search space, the optimizer considers each logical expression, and checks if this expression matches any patterns of the rules in the rule set. If the pattern of a rule is matched, the rule fires to generate the new logically equivalent expression according to the substitute of the rule. Cascades uses expressions to represent patterns and substitutes. Patterns are always logical expressions, while substitutes can be logical or physical.

A rule is called *transformation rule* if its substitute is a logical expression. Fig. 4.4 shows an example of a transformation rule where the top SELECT operator is pushed down to one input of the EQJOIN operator.

A rule is called *implementation rule* if its substitute is a physical expression. For

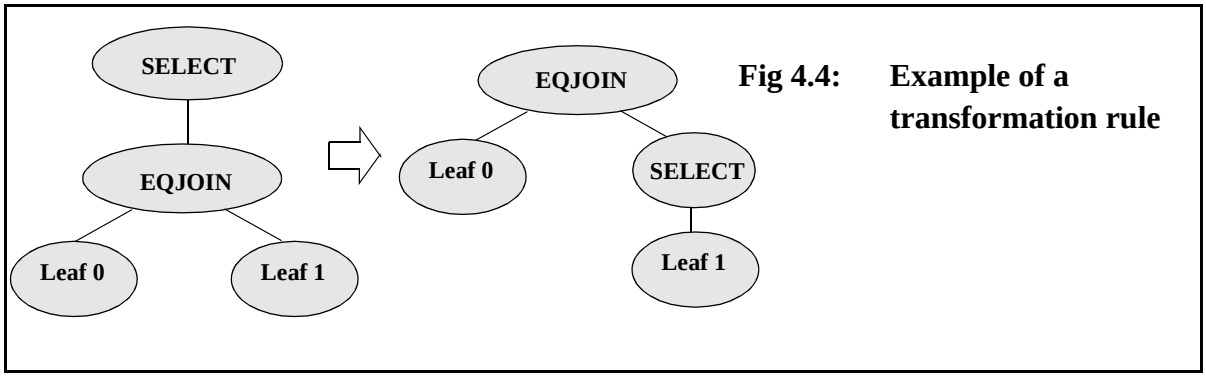


Fig 4.4: Example of a transformation rule

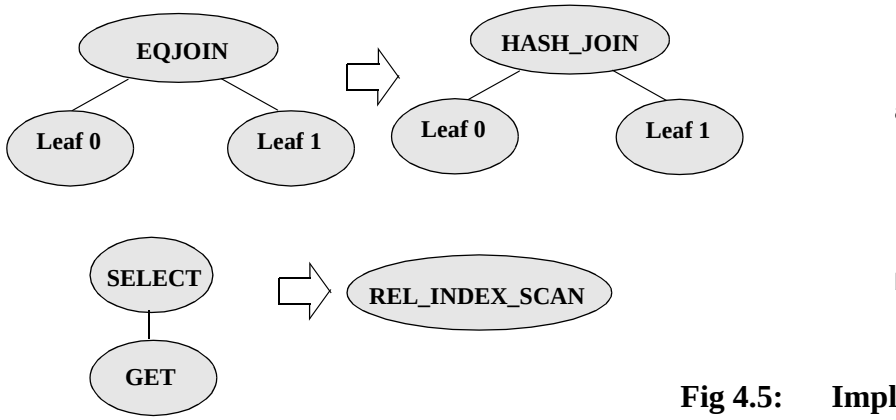


Fig 4.5: Implementation rules

instance, in Fig. 4.5a a logical join operator EQJOIN is transformed into a physical operator, in this case into a HASH_JOIN. Fig. 4.5b shows an example of a rule where two logical operators are transformed into a single physical operator, in this case into a REL_INDEX_SCAN, i.e. a table scan using an index (cf. Appendix B.4).

Enforcement rules are a special type of implementation rules, that can add physical operators to a group in order to enforce certain physical properties (see next section). An example of such an enforcement rule that is in charge of imposing the necessary sort ordering is given in Fig. 4.6. In this case, the SORT operator gets as input the same group where it has been inserted.

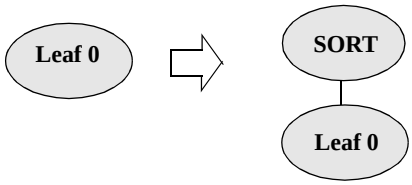


Fig 4.6: Enforcement rule

4.3.1.4 Properties

Cascades computes in each stage of the optimization process the properties of the resulting subplans. These properties describe the result, i.e. the output of the top operator of the respective subplan.

Logical properties are assigned to all subplans. They describe aspects of the result that are independent of the finally chosen implementation algorithm. Such properties are for instance schema, cardinality, attribute statistics, uniqueness, candidate keys and tracked functional dependencies. Since all expressions in a group are logically equivalent (see Section 4.3.1.2), they all have the same logical properties. Thus these properties only need to be calculated once for each group.

In contrast, *physical properties* can only be assigned to physical operators or subplans. They reflect those characteristics of a subplan that result from the employed physical operator. Such a physical property can be for instance the sort ordering.

Cascades further differentiates among *synthesized, required* and *excluded* physical properties. The synthesized properties are the ones that result from a concrete physical implementation of an operator. For instance in Fig. 4.2 the table *A* is sorted on attribute *a*. Hence a full table scan, implemented by the operator "REL_SCAN *A*" imposes the synthesized property *"sorted on a"*. This property is used for the subsequent implementation of the MERGE_JOIN operator, resulting in the fact that no sort operator is needed on its left input. Hence, in the course of the optimization process, the synthesized physical properties as well as the logical properties are calculated bottom-up, while required and excluded physical properties are calculated top-down. For instance, the physical operator MERGE_JOIN requires that both inputs are sorted on the join attribute. Hence, this requirement is propagated from top to bottom.

4.3.1.5 Functionality of Cascades

In Cascades, the optimization algorithm is broken into several parts, which are called *tasks*. All such task objects are collected in a task structure that is realized as a Last-In-First-Out stack. Thus, scheduling a task is very similar to invoking a function. The task is popped out of the stack and the appropriate method of the task is invoked.

As [Gr95] pointed out, other task structures can easily be envisioned. In particular, task objects can be reordered very easily at any point, enabling very flexible mechanisms for heuristic guidance and could even permit efficient parallel search (using shared memory).

The Cascades optimizer first copies the original query into the memo structure. This defines the initial search space. The entire optimization process is then triggered by a task to optimize the top group of the initial search space. This in turn triggers the optimization of smaller and smaller subgroups in the search space. Optimizing a group means finding the best plan in the group according to the cost model. Therefore, it applies rules to all expressions of this group. In this process, new tasks are placed into the task stack and new groups and expressions are added to the search space.

Given the above, it is obvious that the task of optimizing the top group requires that all the subgroups to complete their optimization as well. Hence, after this task is completed, the optimization is also finished, returning as a result the best plan of the top group.

4.4 TOPAZ Strategies

To provide the necessary abstraction, it is important to decouple optimization from some scheduling and load balancing aspects [Ta97]. As presented in Section 2.3, this is achieved in MIDAS through parameters. Thus, the goal of TOPAZ is to come up with a parameterized PQEP. Each parameter keeps track of particular plan properties, like memory allocation, buffer management etc., whose final adjustment has to be made according to the run-time system state by the QEC component.

We will exemplify the parallelization using query Q3 from the TPC-D benchmark [TPC95], whose SQL representation is given in Fig. 4.7a. The sequential execution plan of this query that

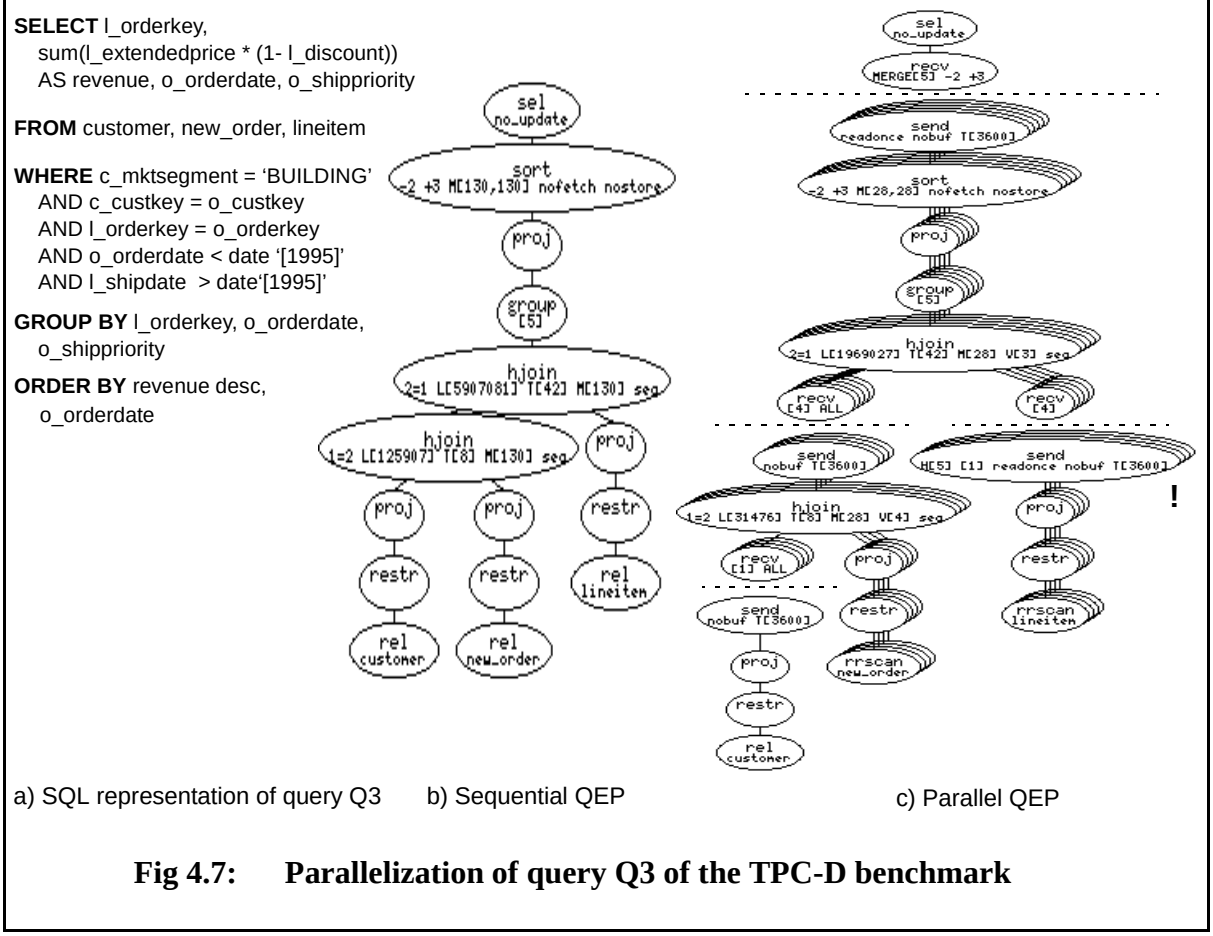


Fig 4.7: Parallelization of query Q3 of the TPC-D benchmark

serves as input for TOPAZ is depicted in Fig. 4.7b. Essentially, it consists of a 3-way join (performed by 2 *hash joins* on the tables CUSTOMER, NEW_ORDER, and LINEITEM) followed by a complex aggregation. In this scenario we further assume that the tables are physically partitioned across 4 disks in a round-robin manner. The resulting parallel query execution plan can be found in Fig. 4.7c.

In the following, we describe in detail some of the core strategies of TOPAZ. In Fig. 4.7c we have depicted the parallel plan generated for our running example. For a better understanding, the block boundaries are marked by dotted lines. This PQEP already shows some of the intrinsic characteristics resulting from our parallelizer that are quite different to the ones known from other approaches:

- cost-related degrees of parallelism and adjusted block sizes, saving scarce resources
- parameters allowing a fine-tuning of the execution plan to different application scenarios
- usage of all possible communication patterns to realize efficient intra-query parallelism.

4.4.1 Control of the Search Space

Exponential complexity [OL90] has forced optimizers to use different techniques to restrict the search space and to improve performance. One of these techniques is to prune expressions that cannot participate in the final, best plan. However, traditional optimization metrics are not suf-

ficient for parallel search spaces [GHK92], because, contrary to sequential optimization, physical resources, partitioning strategies, and scheduling play a vital role. A pruning strategy that doesn't take into account these aspects risks to miss the best parallel plan. Heuristic solutions, as extending the traditional pruning criteria by "interesting partitionings" are also insufficient, as shown in Section 4.2.3. The solutions proposed in [GHK92] and [LVZ93] refer to extensions of the optimization metric that account also for resource utilization. Thus, the costs for a single QEP fill up a vector, and a multidimensional "less-than" is needed to prune the search space. The problem with these approaches is that dynamic programming pruning techniques become generally ineffective and optimization effort explodes in terms of time and memory consumption, as it becomes comparable to exhaustive search. Recent work [GGS96], [GI97] propose a more relaxed cost metric that is based on approximations taking into account some global parameters as critical path length or average work per processing site. To our knowledge, there exists no published work on how to incorporate these cost metrics into existing search engines.

Our solution to these problems comprise the following extensions to top-down optimization:

1. **Cost Model** The strategies proposed in [GHK92], [LVZ93] are known to assure correct pruning. Based on these results, our cost model comprises besides CPU-costs also communication costs, memory usage, disk accesses, and blocking boundaries. In addition, rather than extending the search space to explore alternative plans holding different degrees of parallelism, these degrees are also incorporated into the cost model. Thus, the global processing costs of an operator, i.e. the sum of the costs of its inputs plus the operator's local processing costs, are calculated for different degrees of parallelism and maintained in an array (see Section 4.4.4). The TOPAZ cost model will be presented in more detail in Chapter 5.
2. **Phases** To overcome the drawback of poor optimization performance due to inefficient pruning, parallelization is split into different phases, each phase concentrating on particular aspects of parallel execution. The first phases focus on global optimization of the entire plan w.r.t. dataflow, execution time, and resource utilization. This allows the parallelizer to take global dependencies into account, detecting those locations in the plan where the benefit in exploiting some forms of parallelism is maximized. In the subsequent phases decisions are based on a local view of the QEP, i.e. a view restricted to only one operator or a block of operators and the costs involved in their execution. Another way to express this strategy is that each phase uses as a starting point the result of the previous one to expand a specific region of the search space. These regions do not overlap, since they are expanded using different transformations, i.e. different rule sets. However, the size of the explored search space regions decreases in each phase, as they refer to successively refined aspects of parallel query execution. The final refinement is made by the QEC; it can further adjust certain parameters, like memory usage, degree of parallelism etc. according to the run-time environment. Thus the overall approach to handling the huge search space for parallelization in MIDAS is neither enumeration nor randomization but a cost-based *multi-phase pruning*. This strategy is detailed in Section 4.5.
3. **Pruning Package (*ParPrune)*** Global parameters [GGS96], [GI97] are incorporated in TOPAZ by means of an additional pruning strategy. *ParPrune* further limits the complexity

in each phase, as it guides the search only towards promising regions of the search space. It works in combination with the top-down search engine and consists of two parts: first, in the course of a pre-analysis different global measures are calculated: critical path length, expected memory usage, average costs per CPU, average operator costs etc. Second, these measures serve as constraints (i.e. conditions for rule activations) for all subsequent parallelization phases. The *ParPrune* approach will be presented in more detail in Section 5.6. Apart of the fact that this strategy reduces the optimization effort itself, it can in some cases influence also the quality of the final plans, as e.g. the global pre-analysis permits a better estimation on the search space regions that are worthwhile to be explored in more detail.

4.4.2 Control of the Granularity of Parallelism

As mentioned in Section 3.5, the coarse-grain requirement is not always assured by practical database execution plans. An example coming from traditional QEPs is a restriction evaluating only a low-cost predicate. Some PDBMSs have solved this problem using heuristics, as e.g. parallelizing these operators always together with their predecessors. However, in parallel object-relational DBMSs this is not possible if e.g. a user-defined predicate or low-cost aggregation requires a special partitioning strategy. It is an open problem how to deal with these operators. Parallelizing them separately causes obviously too much overhead, while a sequential execution can cause bottlenecks at several places of the PQEP and thus suboptimal performance.

Therefore, in contrast to other strategies [GI97, GGS96], we incorporated into TOPAZ also the possibility of bundling together several operators into a `block`, i.e. to perform several operators within a single execution unit, as recommended in Section 3.5. The strategy is cost-based, as it accounts for operator costs, selectivities, and intermediate result sizes to determine block boundaries. Following the recommendations resulting from Section 3.5, an additional goal is to achieve mutually adjusted processing rates among communicating blocks.

The degree of parallelism of the resulting blocks is adjusted by TOPAZ to the actual block processing costs, i.e. the sum of the costs of the constituting operators. This guarantees overall efficiency and is in contrast to approaches used by other PDBMSs as e.g. choosing the same DOP for the whole QEP or limiting the considered degrees to a few alternatives [BF97, Or98, JMP97].

Intra-block parallelism is analogous to intra-operator parallelism and requires to execute several instances of the complete block by different execution units. Each instance has to work on different sets of data, i.e. the processing within one instance of the block is independent from all the other instances of this block. In the PQEP shown in Fig. 4.7c, the largest block is formed by the *sort*, projection (*proj*), *group*, and hash-join (*hjoin*) operators having a DOP of 5.

The necessary conditions to bundle operators within a block are: `same degrees of parallelism` and `same partitioning strategies`. Thus, in order to achieve efficient block building, a flexible control of these properties is necessary, as described in the following sections. However, these conditions are not sufficient. A cost-based analysis has to decide if a specific block construction also leads to a decrease of the overall processing costs.

4.4.3 Control of Partitioning Strategies

In order to have the necessary degrees of freedom, TOPAZ distinguishes between logical and physical data partitioning. As presented in Section 3.3.3, the strategies for physical data partitioning implemented in MIDAS are *round-robin, hash, range* and *user-defined partitioning.* Which of these techniques is used depends on the type of the operator that has to be parallelized. In many cases, the partitioning has to keep track of the attribute values, like in the case of hash- or range-based partitioning. For instance, in Fig. 4.7c the *send* operator highlighted by an exclamation mark performs a hash partitioning on the first attribute into 5 partitions as indicated in the operator description by the parameter *H[5] [1].* However, TOPAZ differentiates only between the following logical partitionings:

- ***Any*:** This parameter indicates that the parallelized operator (or block) doesn't necessarily need a specific partitioning (as e.g. the *sort* operator).
- ***Attr*:** If an operator needs a value-based partitioning on certain attributes (as e.g. in the case of certain aggregations), the corresponding *send* operator is extended by the *Attr* parameter together with the identifiers of the required partitioning attributes.

Thus, if a block construction becomes necessary in the course of parallelization, TOPAZ can change a less strict partitioning (like *Any*) into a stricter one (like *Attr*). This can be done easily, only by taking into consideration the required physical properties. At the end of the parallelization, when block construction is finalized, these logical parameters are mapped to one of the above mentioned physical partitioning strategies.

4.4.4 Control of the Degrees of Parallelism

Consider a QEP having two adjacent high-cost operators. In Fig. 4.8 (left), these are the final phase of a *sort* (merging of sorted runs) and an aggregation operator (*group*). As both of them are coarse-grain, both are processed using intra-operator parallelism. Suppose that by taking into account only the local costs of the operators and the intermediate result sizes, the best degree of parallelism for the *sort* operator results to 3 and that for the aggregation is 2. Due to the different degrees of parallelism, a repartitioning of the intermediate results of the *sort* operator is necessary, implicating high communication costs.

If the degree of parallelism of the *group* is increased to 3, pipelining between the two operators becomes possible. This reduces communication costs, but increases the number of execution units from 5 to 6. Actually, the optimal execution for the two operators would be within the same block, but with an increased degree of parallelism according to the higher block processing costs, as shown in Fig. 4.8 (right). This implies less execution units and less communication costs, as only the aggregated result of the *group* has to be transmitted. A plan with similar response time but reduced resource consumption is also more suitable for a multi-query environment.

Assume that this query is optimized by a search strategy that adopts local pruning. Considering e.g. a bottom-up optimizer, it first optimizes the *sort*, finding the best degree of parallelism of 3 and prunes all the other plans, as they are (locally) more expensive. At the next level, when opti-

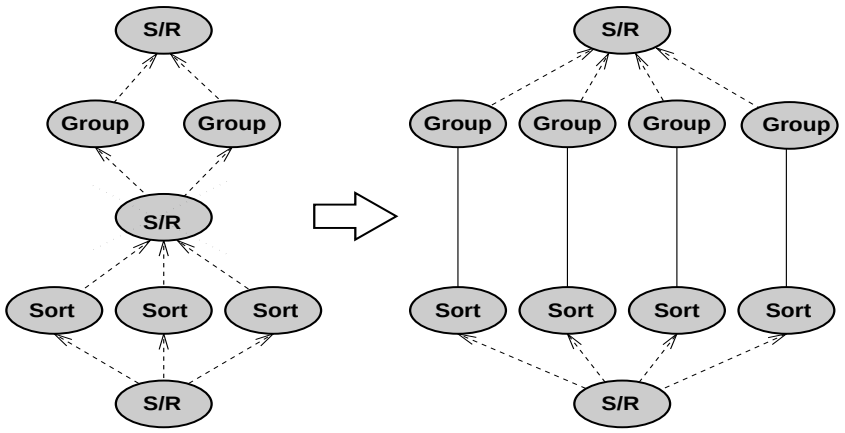


Fig 4.8: Adjusting the DOP for block construction

mizing the *group*, the search engine cannot find the best plan shown in Fig. 4.8 (right), because the search space doesn't contain the plan and costs for the *sort* operator in combination with a degree of parallelism of 4. However, keeping the plan alternatives for all possible DOPs is also an impractical solution with regard to optimizer performance.

We have elaborated the following solution to this problem: To keep the degrees of parallelism flexible, TOPAZ incorporates this aspect only in the cost model, without explicitly extending the search space with alternative plans that differ only in the degrees of parallelism. If an operator gets parallelized by partitioning its inputs, the corresponding *send* operator doesn't hold any specific information on the number of partitions. A parameter like *"Attr[2] 1"* in the course of the parallelization only means that this *send* operator performs a value-based partitioning on the first attribute and that the number of partitions is greater or equal 2. At the same time the costs of the operator are calculated for all possible degrees of parallelism, storing them in an array. This cost calculation is propagated upwards. The global processing costs of the successor can also be calculated correctly for different DOPs, since its local processing costs are known and the processing costs of its input are available for every considered DOP. Thus, e.g. the decision on combining two blocks can be taken on the basis of the lowest value in the cost array of the topmost operator. In the example, this is the *group* and the entry in its cost array corresponding to the minimal global processing costs will be found for a DOP of 4.

In Section 4.2, we have already mentioned some heuristics used in practice, as e.g. choosing the same DOP for the whole PQEP or limiting the considered degrees to a few alternatives [BF97], [Or99], [JMP97]. New query types, as e.g. DSS and object-relational ones, make the usage of CPU-intensive operators and UDFs more and more popular. In these scenarios, the operator costs in a QEP can differ significantly. We believe that the degree of parallelism for these operators can rely only on cost-based decisions, as in TOPAZ, whereas using only restricted heuristics like the ones mentioned above can lead to truly suboptimal parallel plans.

4.5 Multi-Phase Parallelization

In the following we describe the parallelization phases that exploit the strategies described in the previous section, using as example the TPC-D query Q3 (Fig. 4.7a). Please note that the PQEPs presented in each phase are complete physical trees, having specific data partitionings and degrees of parallelism, although we mentioned before that these aspects are kept flexible. In TOPAZ each phase can be separately turned on or off. Thus the following examples rather reflect the physical trees that are obtained if the phases are turned on successively, starting with the sequential one (Fig. 4.7b). We accentuate that this is only for illustration purposes, as the final parallel plan is the result of *all* constituting phases that explore different regions of the parallel search space. The corresponding rules are presented in Appendix B.2.

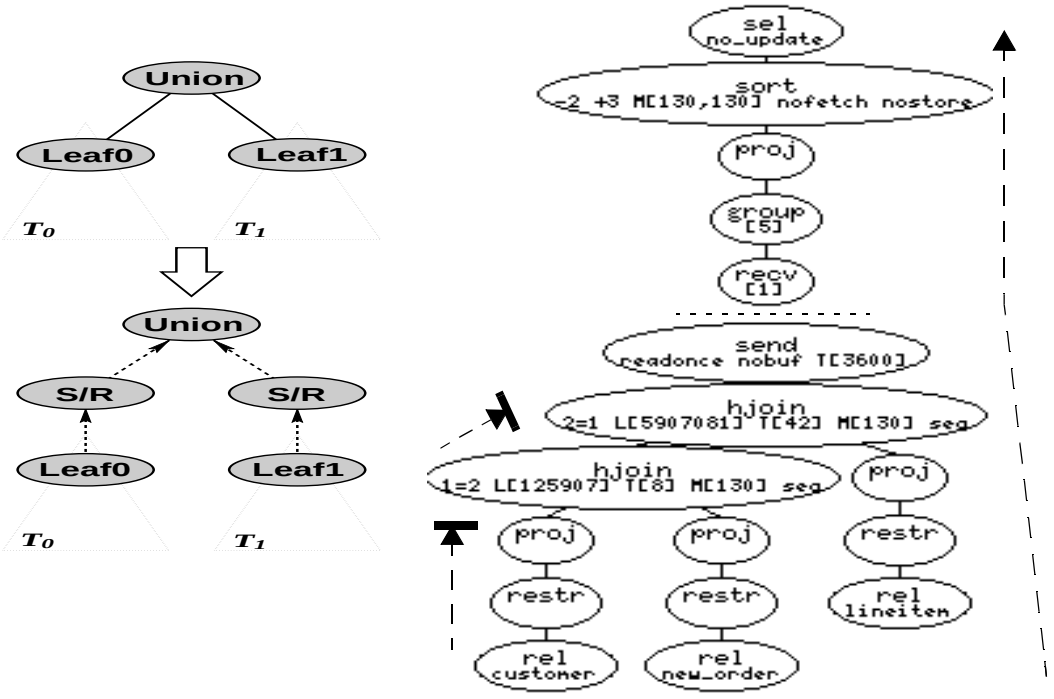
As each phase is characterized by a separate rule set, examples of representative rules and of rule applications will be provided as well. Since the *send* and *receive* operators appear always in pairs, they are internally considered as a single operator, called *S/R*, holding the parameters for the respective *send (S...)* and *receive (R...)* part. However, in a physical plan, they are represented separately at the end and at the beginning of neighboring blocks, constituting a data river.

4.5.1 Phase 1: Inter-Operator Parallelism and Refinement of Global Costs

This phase starts from the sequential plan and analyzes the possibility of reducing the critical path length through inter-operator parallelism. An additional goal is to achieve a mutually adjusted processing rate over all blocks in the QEP, thus considerably reducing query execution overhead, as described in Section 3.5. The transformations considered in this phase expand a search space region containing alternative plans that exploit only pipelining and independent parallelism. The decision criteria comprise sizes of intermediate results, expected resource consumption, processing costs of the emerging blocks as well as blocking operators.

A naive strategy would be to define a single rule for insertion of *S/R* nodes and let the search engine find the optimal places for inter-operator parallelism according to the cost model. But this increases unnecessarily the parallelization effort, since alternatives that are unlikely to lead to the best plan are explored as well. For example, pipelining shouldn't be considered in combination with subplans that are not on the critical path. This naive strategy would lead already for this first phase to an unacceptable performance. Hence, the considered alternatives are restricted by *ParPrune*. In this phase it accounts for the relative costs of the operators and the critical path length computed during the pre-analysis. Thus, inter-operation parallelism is considered only in combination with certain subplans and operators that are reasonable from a global point of view. In Fig. 4.9a, a rule for the insertion of pipelining *S/R* nodes below a binary operator is presented. The condition for the consideration of this transformation within a QEP is that both inputs *T0* and *T1* exceed certain cost limits.

Our example query resulting from this phase is presented in Fig. 4.9b. As shown by the interrupted dashed arrows, the left inputs of the join operators are blocking, since they are used to build the hash tables. Hence, efficient pipelining is only possible in the segment marked by the continuous dashed arrow at the right side of the figure. In this segment, the *group* is recognized



a) Rule example

b) Intermediate result after phase 1

Fig 4.9: Inter-operator parallelism

as a costly operator due to the size of the intermediate result and the high local processing cost. Thus the cost model has determined the introduction of only one pipelining edge, as shown in the figure. This results in two blocks with similar processing rates.

Please note that the goal of this phase is not to come up with the final set of edges for inter-operator parallelism. Due to modified cost proportions in the next phases, some of these edges may be replaced by neighboring ones. The result of this phase are refined cost limits that have been established w.r.t. critical path length and average block processing costs. These refined costs are exploited by the subsequent phases.

4.5.2 Phase 2: Intra-Operator Parallelism applied to High-Cost Operators

The result of the previous phase is now used to span a new search space region, exploring the possibility of further reducing the critical path length and block processing costs by controlled introduction of intra-operator parallelism. Therefore operators that already meet the coarse-grain demand are individually parallelized, bracketing them with *send-receive* nodes.

Depending on the type of the operator, one or both inputs have to be partitioned. Hence, partitioning *send* nodes are inserted such that each operator instance processes one partition. The intermediate results produced by these instances are collected by a *receive* node that is placed at the output of the operator. For each operator separate parallelization rules have been defined, considering the operators' characteristics, as e.g. some operator types admit more alternatives. As shown in Fig. 4.10a, e.g. a *hash-join* can be parallelized by partitioning both inputs or only

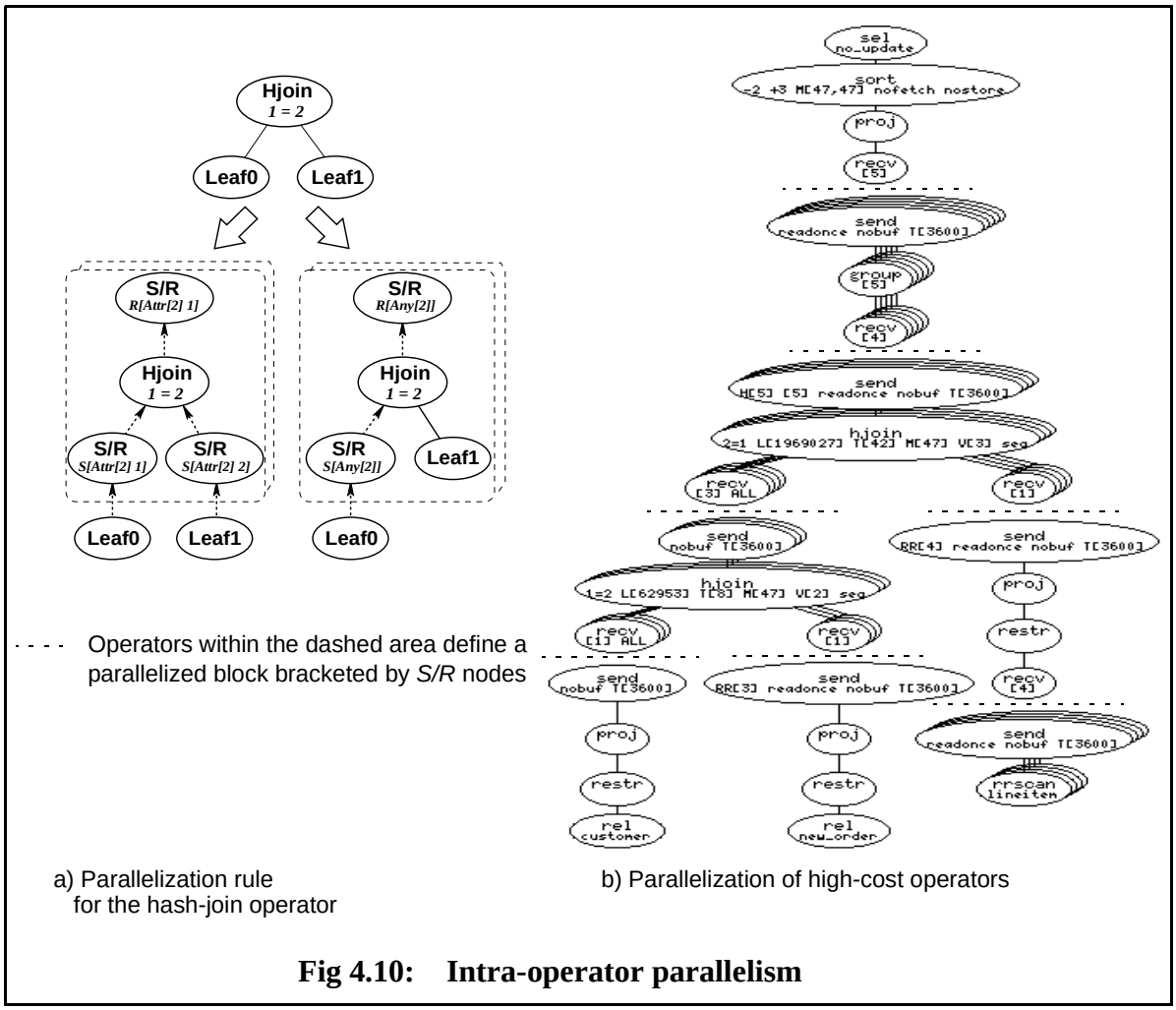


Fig 4.10: Intra-operator parallelism

one input combined with a replication of the other. The solution chosen by TOPAZ depends on the cost distribution in the QEP. For instance, if one of the inputs is replicated, there exists no requirement concerning the partitioning strategy of the other. Thus repartitioning can be omitted, an aspect that is especially beneficial if this input has a high cardinality. The *S/R* node parameters only indicate logical partitionings (*Attr* or *Any*) without specifying any concrete degrees of parallelism or physical partitioning strategies (see Section 4.4.3).

Global execution performance and critical path length are mostly influenced by nodes having high local processing costs. The effect of *ParPrune* in this phase is to take into account the average costs per operator computed during the pre-analysis and to consider only those operators that beside the coarse-grain requirement, also exceed a minimal cost limit.

In Fig. 4.10b the result for our example query is shown if parallelization is stopped at this stage. Please note that this is only for illustration purposes. Thus the parallelization includes also tasks that are usually performed only after the last phase, like mapping from logical to physical partitioning strategies and setting of concrete DOPs. These have been chosen according to intermediate result sizes, local processing costs, and disk partitioning. The parallelized operators are the *group* (DOP=5), the two *joins* (DOP=4 and DOP=3), and the *scan* of the LINEITEM table (*rrscan*: round-robin scan) with DOP=4. Obviously, in this example the *group* is the most costly operator, since the DOP assigned to it is the highest. The *group* requires a partitioning on the 5th attribute, as indicated by the parameter *(H[5] [5]...)* of the corresponding *send* operator,

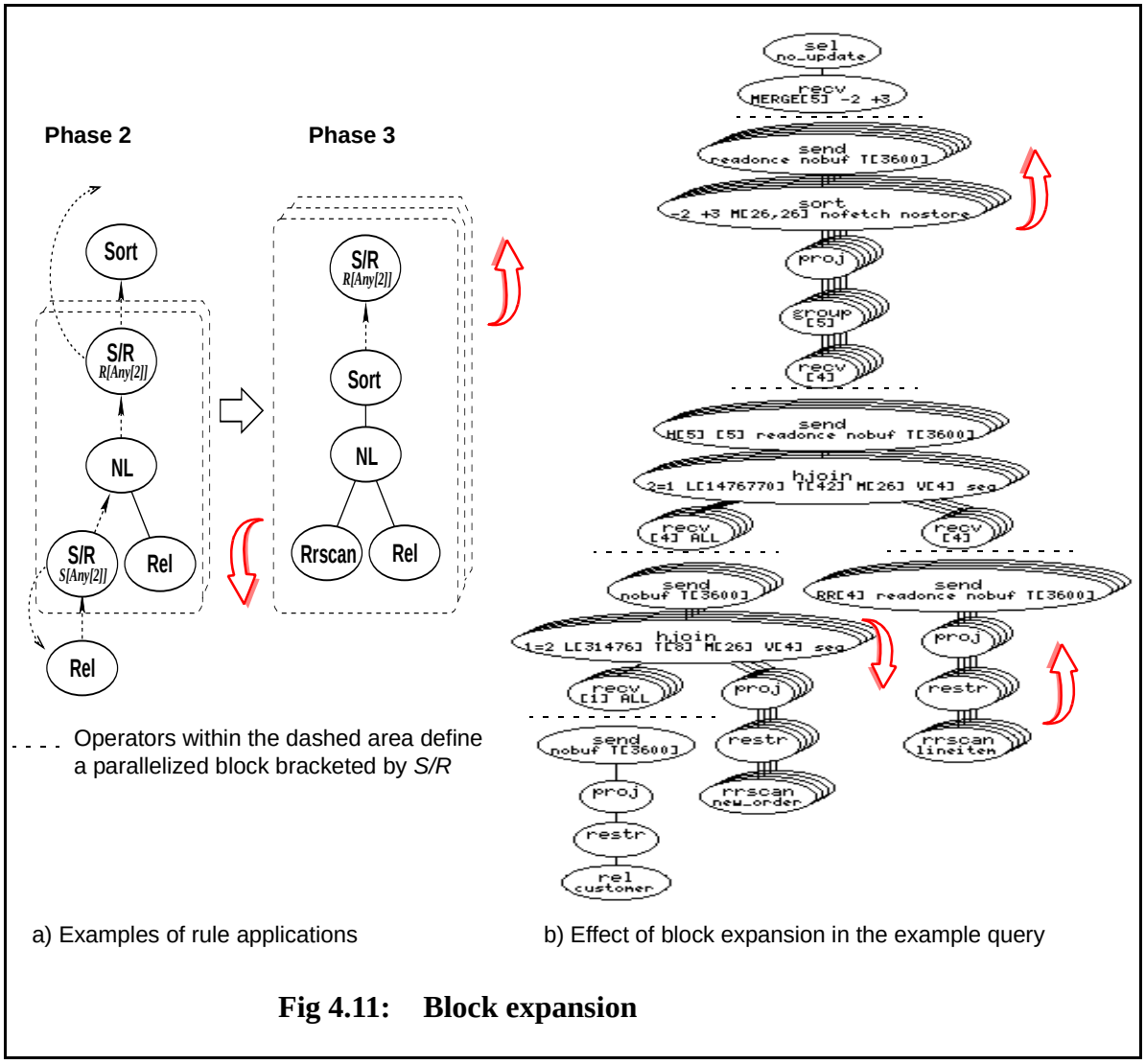
identifying a hash partitioning of attribute 5 into 5 buckets. In contrast to shared-disk or shared-everything architectures, in a shared-nothing environment the parallelization of the relation scans in this phase is restricted by the given physical disk partitioning strategies. This constraint can be modeled as an additional required physical property.

As a result of this phase simple blocks that hold one parallelized operator show up. The parallelization of these *driver nodes* impose certain physical properties, like data partitioning, degree of parallelism, and sort order that will bias the parallelization of the remaining operators.

4.5.3 Phase 3: Block Expansion and Treatment of Low-Cost Operators

Phase 3 analyzes the possibility of expanding the one-operator blocks obtained in the previous phase. The resulting blocks incorporate also operators that individually don't meet the coarse-grain requirement or have low processing costs. As shown in Section 4.4.2 this achieves a minimization of the resources needed to process the given set of operators and avoids bottlenecks. The DOPs are adjusted according to the block processing costs (see Section 4.4.4).

The corresponding search space region is expanded by transformations that slide the *S/R* oper-



a) Examples of rule applications

b) Effect of block expansion in the example query

Fig 4.11: Block expansion

ators towards not yet parallelized operators, thus including them into existing blocks. If in the course of this sliding two *S/R* nodes meet, they are transformed into a single repartitioning node. In Fig. 4.11a, a situation is shown where the nested-loop operator *(NL)* has been parallelized in Phase 2 by repartitioning an input and replicating the other. In Phase 3, the top *S/R* node is pushed up. As a result, the *Sort* operator becomes part of the block taking over the parallelization decisions and properties of that block. The other *S/R* node, i.e. the one below the *NL* operator, is pushed downwards, thus parallelizing the relation scan (*Rel* operator). This transformation is specific to shared-disk and shared-everything architectures, expressing an additional degree of freedom compared to shared-nothing environments. As stated before, in the latter case the scans have to be parallelized in Phase 2, accounting also for physical disk partitionings. The result of the two transformations is a block consisting of the 4 operators, having the same DOP and the same partitioning strategy. The *Rrscan* operator, a parallel scan, reads different partitions of the first input table in each block instance. The *Rel* operator reads the entire second input table and replicates it to all block instances.

All of the above mentioned transformations, e.g. pushing an *S/R* node through an operator, merging of two neighboring *S/R* nodes into a single repartitioning node etc., are defined as rules cf. Appendix B.2, the resulting plans being added to the search space and maintained according to cost-based decisions. To reduce the number of worthless transformations, *ParPrune* for instance checks in advance if a given partitioning strategy can be taken over by a candidate operator.

In our example query (Fig. 4.11b), the parallelization of the lower *hash join* has been extended downwards, parallelizing also the *scan* of the NEW_ORDER table and adjusting the DOP of the block from 3 to 4. As explained before, this adjustment is triggered from the increased cost of the resulting block, now holding 4 operators: the *hjoin*, *proj*, *restr* respectively *rrscan(new_order)*. Analogously, the parallelization of the LINEITEM *scan* and the *group* have been extended upwards. Due to low processing costs, the *scan* of the CUSTOMER table is done sequentially, however replicating the result for further parallel processing.

4.5.4 Phase 4: Block Combination Further Decreasing Parallelization Overhead

As described in Section 4.4.2, bundling coarse-grain blocks can lead to a further reduction of resource utilization and intra-query communication, thus contributing even to the decrease of the critical path length. Therefore, the last parallelization phase analyzes the possibility of combining adjacent blocks with comparable partitioning strategies.

Phase 4 operates with a single rule for the elimination of repartitioning nodes between two adjacent blocks. This is only possible if the partitioning strategy of the candidate blocks is equal or comparable. As shown in Section 4.4.3, this condition is satisfied if e.g. the logical partitioning of at least one block is *Any*. For the final plan shown in Fig. 4.7c, the *group* block has been bundled together with the upper *hash join* block adjusting the DOP to 5. The required partitioning imposed by the *group* has been taken into consideration by modifying the partitioning of the join block from round-robin (*send(RR[4]...)*) to hash (*send(H[5] [1]...)*), as highlighted by the excla-

mation mark. Hence, repartitioning has been pushed down to be performed *before* the join operator, where it is more beneficial w.r.t. intermediate result sizes. This shows again that TOPAZ keeps track of all cost factors also on a global level.

4.6 Preventing Deadlock Situations

In Section 3.4 we have presented some deadlock scenarios with possible solutions to resolve the data dependency. However, it is still an open problem how to recognize and prevent these situations. Most previous work [Has95], [HS93], [BDV96], [GI97], [GGS96] uses simplifying assumptions in their model that ignore issues like data flow or deadlocks. Nevertheless, in forthcoming application scenarios that refer also to object-relational enhancements, a comprehensive treatment of all aspects related to query processing is necessary.

One possibility to detect deadlocks would be to implement a sophisticated communication subsystem that controls the exchange of intermediate results, waiting situations and additional disk spoolings. Clearly, in a parallel and possibly distributed environment this would require a non-negligible overhead. Hence, we have elaborated a different strategy. This is based on shifting the recognition and treatment of deadlocks from runtime to plan generation time, yielding a generally applicable deadlock-aware optimization that produces deadlock-free parallel plans. In the following, we will present this approach followed by an evaluation w.r.t. applicability and efficiency.

4.6.1 Deadlock-Aware Parallelization

As presented in the previous sections, TOPAZ uses various physical properties that are taken into account in the course of parallelization. In order to detect potential deadlock scenarios, we have introduced an additional property, called *deadlock*. This is set to TRUE if the current operator could be part of a cycle caused by data dependencies. This property is propagated downwards, thus being taken into consideration for the parallelization of the inputs. Please observe that in this way the treatment of the *deadlock* property is similar to the one for the sorted-order property. The only difference is that the latter requirement is satisfied by the insertion of additional *sort* nodes, while the deadlock-free requirement is resolved by substituting certain *send* nodes from WAIT to *materializing, as* proposed in Section 3.4.

We will exemplify this strategy using as an example the subplan in Fig. 4.12. In this example, the subplan is parallelized by partitioning the inputs of the two join operators. This is achieved by the lower *S/R* pairs, that realize a value-based partitioning on the join attributes. As presented in Section 4.4.3, a parameter like *"Attr[2] 1"* in the course of parallelization means that the corresponding *send* operator performs a value-based, e.g. hash or range partitioning on the first attribute and that the number of partitions is greater or equal 2. The topmost *S/R* pair merges the intermediate results created by the subplan instances that are executed in parallel.

If during the top-down parallelization a *receive* node with a *demand-driven* merging strategy is

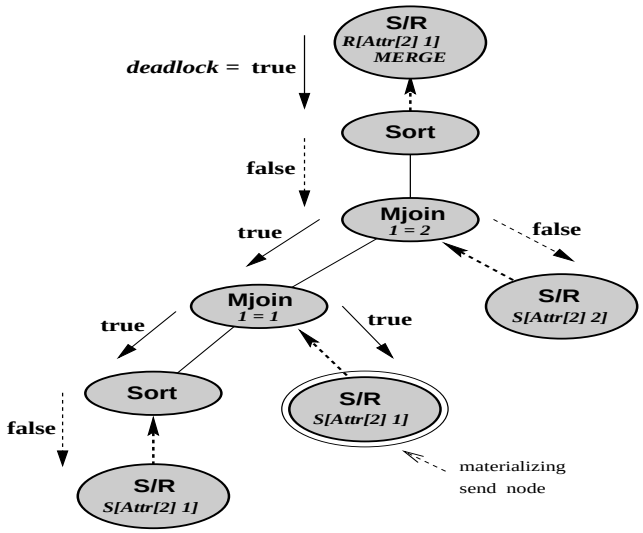


Fig 4.12: Preventing deadlocks situations by using top-down optimization

inserted, the *deadlock* property is set to TRUE, as these nodes can always be the origin of a potential deadlock scenario (see Sections 3.4.1 and 3.4.2). This is exemplified by the topmost *S/R* pair that realizes a sorted output from locally sorted data streams (as indicated by the MERGE parameter). However, if in the subsequent optimization stages the insertion of blocking operators is necessary, in this example the upper *sort* operator to realize locally sorted data streams as well as the lower *sort* operator inserted because of the *merge join*, the *deadlock* property can be reset to FALSE. As presented in Section 3.4, these operators resolve the data dependency.

In the case of binary operators with simultaneous input processing, in our example the *merge joins*, a deadlock is only possible, if they are parallelized via partitioning, repartitioning, or replication (see Section 3.4.3). However, setting *deadlock* to TRUE for both inputs is only necessary if the deadlock probability emerged already from the parallelization of the top nodes, i.e. if the *deadlock* property was already set to TRUE when reaching the current operator (as in the case of the lower *merge join* operator in Fig. 4.12). Otherwise, it is sufficient to propagate the deadlock alert to only one of the inputs. If the operator is parallelized by partitioning only one input while replicating the other, the deadlock probability will be propagated to the replicating input. The rationale behind this decision is that it is always the input with the smaller cardinality that is subject to replication, hence a materialization at this point is less costly. In addition, usually in this case intermediate results have to be materialized anyway, because of the multiple readers. If the operator is parallelized by partitioning both inputs, TOPAZ selects the input with the smaller cardinality to propagate the deadlock alert (as in the case of the upper *merge join* operator).

If the *deadlock* property is still set to TRUE when a *send* operator is reached, this one will be implemented as *materializing*. Thus, to guarantee a deadlock-free execution for the example in Fig. 4.12, from the three *send* operators, only one had to be implemented as *materializing*.

4.6.2 Assessment of the Approach

With the strategy presented, the data dependency could be resolved efficiently already in the course of plan generation, involving only minimal optimization overhead. The decision on where to introduce materialization points is cost-based, taking into account intermediate result cardinalities as well as operator processing costs.

In order to assess the benefits of this approach it is necessary to discuss the following cases:

- No data dependency detected

  If there is no data dependency detected at compile time, no deadlocks can occur. In this case our treatment of deadlocks is not affecting the final plan. The parallel QEP constructed will be the same as if the deadlock extension is not set on during optimization time, but with the additional guarantee that the execution of this plan will not result into any deadlock situation. Hence there is no need anymore for a runtime deadlock facility.

- Data dependency existent, but uniform processing rates

  Because of a potential deadlock scenario, the optimizer inserted one or several *materializing send* nodes. However, the *materializing* property leads to disk spoolings only if the communication buffers overflow. As presented in Section 3.3.6, if the processing rates of producing and consuming instances are comparable, a small buffer pool prevents disk I/O. From an execution point of view, in this situation the *materializing* strategy is not different from the WAIT strategy, because the similar processing rates do also prevent from buffer overflow, waiting situations as well as deadlocks. As a result our strategy does not lead to worse performance or additional overhead.

- Data dependency existent, but non-uniform processing rates

  If the processing rate of the consumer is much lower than that of the producer and the communication between the instances is realized via a *materializing send* operator, the overflown intermediate result pages will be forced to disk. However, this means that the data dependency detected in the optimization phase would have surely caused a scenario with at least two execution units in a waiting and later on in a deadlock situation. Hence, without the presented strategy for deadlock treatment during parallelization this scenario, in turn, has to be detected and resolved all at runtime. Truly this would result in more overhead and worse performance than compared to our solution.

By correlating necessary materialization points with the smallest possible cardinalities, the optimization-based strategy is able to minimize disk contention. Furthermore, this is also reinforced by the fact that the primary goal of TOPAZ is to generate parallel plans with uniform processing rates. As mentioned above, this prevents from disk I/O even if some *send* operators are implemented as *materializing*.

This is in contrast to a dedicated communication subsystem, that introduces runtime overhead to control the data flow and to devise necessary disk spoolings. In addition, these approaches frequently use a timeout-based strategy in order to recognize deadlocks. These idle situations clearly have a negative influence on query performance. Moreover, the runtime overhead to resolve them is additionally increased by the fact that deadlocks can have various other sources in a multi-user distributed system, for instance caused by various synchronization locks. In such

an environment, it is extremely important to assure that the execution of a stand-alone execution plan is deadlock-free.

4.7 Performance Investigation

The TOPAZ data and cost models have been implemented using the Cascades Optimizer Framework. The current version has approximately 80 rules, divided into 4 categories, one for each parallelization phase (see Appendix B.2). We have validated our approach by using different applications, such as OLAP, DSS, and digital libraries. In this section, we report on the performance of TOPAZ by using a series of TPC-D queries performed in a single-user environment on a 100 MB database, running on a cluster of 4 SUN-ULTRA1 workstations with 143 MHz Ultra SPARC processors, connected via a Fast Ethernet network. In order to perform a detailed analysis of the separate parallelization phases, we took the result of each plan and executed it on our cluster.

Fig. 4.13 shows the average speedups obtained after each phase for all queries of the test series, parallelized for the 4 workstations; the speedup obtained by our running example TPC-D query Q3 is illustrated in a separate curve. We would like to remind that this is for demonstration purposes, since parallelization is made up of all phases, the actual result being that obtained after Phase 4.

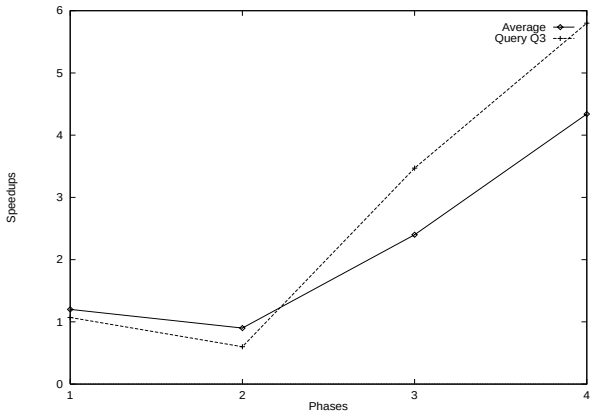


Fig 4.13: Speedups after each phase

The first two are only preparatory phases that result in the insertion of different forms of parallelism according to a global cost-based analysis (see Section 4.4.1). These are carried over in subsequent phases to the rest of the QEP, considering also physical properties in the top-down parallelization, as e.g. partitioning and sort orders (see Sections 4.5.3 and 4.5.4). These are the phases where the real speedups are achieved. In Phase 2 coarse-grain operators that significantly contribute to the critical path are parallelized separately. The negative speedup demonstrates quite dramatically our statement (Section 4.4.2) that (ignoring non-coarse-grain

Table 3.5 Speedup distribution in the test series

Total number of queries	16
# queries with superlinear speedup (4.5 to 13)	6
# queries with near-linear speedup (4)	5
# queries with sublinear speedup (1.5 to 3.5)	5

Table 3.6 Effect of pruning and global view on execution and parallelization

Resource and response time metrics	ParPrune off	ParPrune on
Average execution time for modified queries (ms)	25943	23717
Average number of execution units for modified queries	11.125	8.25
Overall average parallelization time (ms)	884	703

operators causes bottlenecks in parallel execution, thus influencing negatively performance. The difference between Phase 2 and 3, respectively Phase 3 and 4 shows the importance of block construction, optimal setting of degrees of parallelism, and other TOPAZ strategies as described in Section 4.4.

Please note that in some cases, as e.g. for query Q3, we obtained superlinear speedup see also Tab. 3.5). This is due to the fact that scaleup refers not only to CPUs, but also to other resources. Hence, if a query is parallelized correctly it can benefit also from parallel I/O facilities and from the increased database cache that can reduce disk spoolings. The results show the importance of incorporating these aspects into the cost model, as proposed by TOPAZ and presented in detail in Chapter 5. Of course, this situation can change in a multi-user environment, due to general resource contention.

Tab. 3.5 shows also some sublinear speedups. As mentioned before, the implemented base version of TOPAZ gets as an input a complete sequential tree, produced by a sequential optimizer. We have observed that some characteristics of these trees can influence the quality of the final parallel plan. Thus the suboptimal speedups are mostly related to queries containing a correlation, with this property preventing an efficient parallelization. The treatment of this problem within query optimization will be presented in Chapter 6. However, we have never observed a deterioration w.r.t. the (sequential) performance, as all TOPAZ strategies account for parallelization overhead and thus introduce parallelism only where it is truly beneficial.

W.r.t. the importance of a global view in the parallelization process, we have parallelized and executed the queries with and without the *ParPrune* technique that can be easily switched on or off in our prototype. As described in Section 4.4.1, *ParPrune* is used to provide an additional guidance throughout the parallelization phases. This is to reduce optimization complexity. However, as a side-effect, *ParPrune* can also improve the quality of the final plan as the global pre-analysis permits a better estimation on the search space regions that are worthwhile to be explored. In the test series, *ParPrune* modified the final plan in 50% of the test cases. As can be seen in Tab. 3.6, for these queries an additional performance improvement has been achieved. An interesting aspect is that this performance gain has been achieved with explicitly less resource consumption. We have only listed here the number of execution units, that in this way has been reduced by 34%. But even where *ParPrune* didn't come up with a more efficient plan, the best plan has been found with clearly less effort. This can be seen already by comparing the average parallelization times in the last row of Tab. 3.6. However, these numbers also include some organization overhead, as e.g. the time necessary to copy the QEPs into and out of the Cascades memory structure. Please note that the numbers are comparable to sequential optimization efforts.

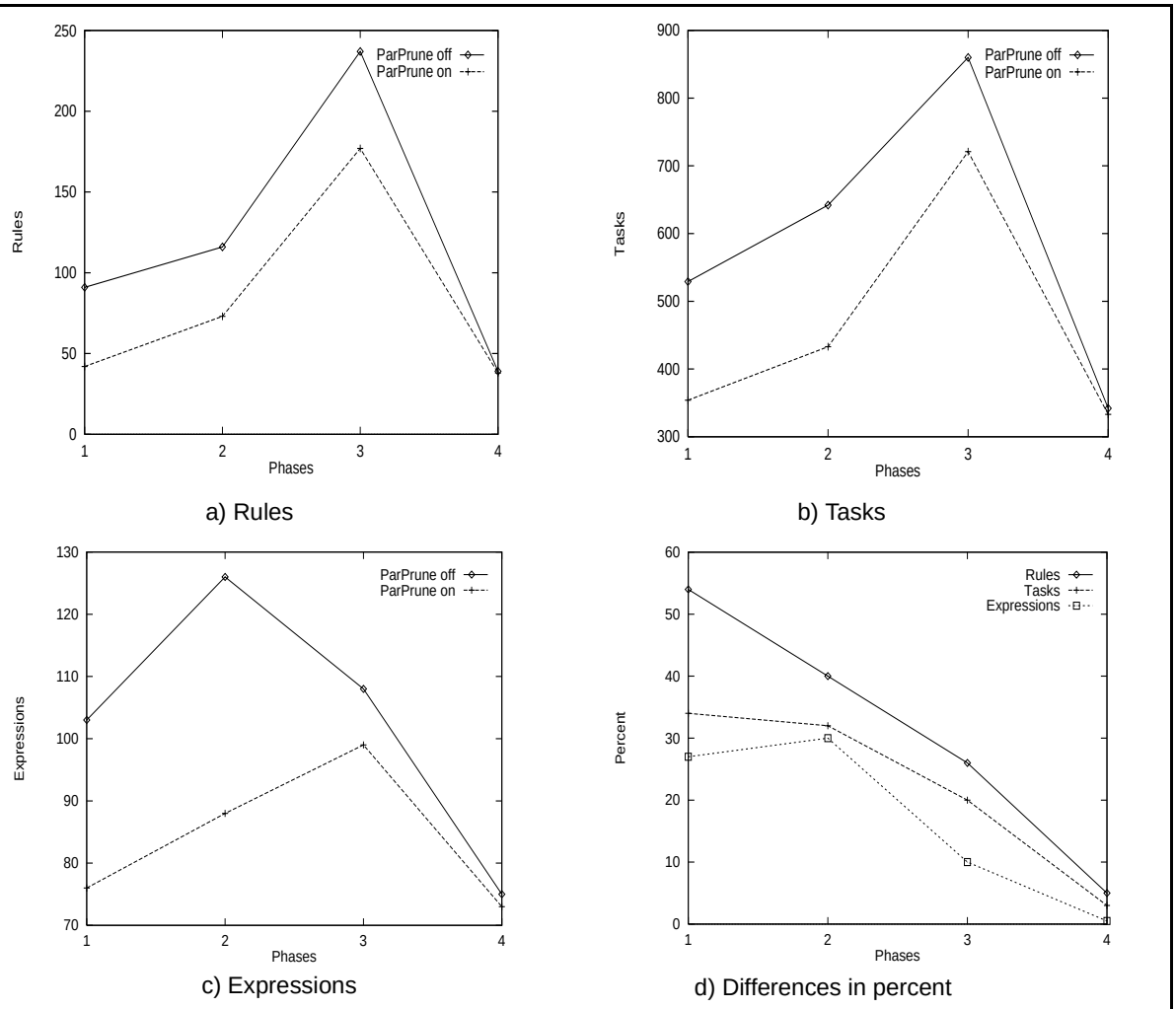


Fig 4.14: Influence of *ParPrune* on rules, tasks and expressions participating in each phase

To evaluate only the search complexity, we have used as measures the number of expressions generated, the number of tasks and the number of rule applications in the course of the parallelization. Please note that these measures have been presented in Section 4.3.

In Fig. 4.14a, b, and c the average number of rules, tasks, and expressions participating in each phase of the parallelization are compared. As can be seen, by using *ParPrune*, these numbers could be drastically reduced as compared to a non-pruned parallelization attempt. In order to get a better understanding, a summarization is given in Fig. 4.14d, showing for each phase the reductions (in percent) achieved for these measures. Thus, e.g. the number of applied rules in the first phase is reduced drastically, by 54%. Generally, the impact of *ParPrune* is the highest in the first two phases, as these are the ones that participate most in the determination of the final character of the PQEP. It is here that a guidance given by a pruning strategy can help the most in finding the right regions of the search space. Later on only a gradual refinement of the parallel plan takes place that translates to a search only around the regions found in the earlier phases. Thus, in these last phases pruning can only contribute to the reduction of unnecessary transformations and this impact is not so visible.

4.8 Summary

In this chapter we have shown that our approach, called TOPAZ, fulfills all basic requirements of a modern parallelizer. This is accomplished by a cost-based, rule-driven and multi-phase strategy. A thorough performance analysis and evaluation of our parallelizer technology clearly showed that the complex parallelization task can be conducted by TOPAZ's underlying parallelization strategies as well as internal optimization and control measures such as *ParPrune*. These measurements further indicate that the parallel plans created by TOPAZ are executable by state-of-the-art parallel database engines showing linear speedup. In summary, our investigations manifested that these results can only be achieved by the integration of all before mentioned parallelizer properties. However, considering the non-negligible overhead resulting from the usage of such a sophisticated parallelization strategy, the goal of TOPAZ is clearly the parallelization of large and complex queries, that can make truly profit of all its intrinsic features, including also its deadlock-preventing facility.

Chapter 5
The TOPAZ Cost Model

This section describes the cost model that has been designed and implemented as a part of the Cascades data model for the TOPAZ parallelizer.

5.1 Introduction

The starting point for the TOPAZ cost model constituted ideas from [GHK92] and [LVZ93]. However, significant extensions and modifications have been made in order to satisfy the requirements of a full-function, parallel object-relational DBMS as MIDAS.

Thus, a guiding principle in our cost model design has been to consider parallel processing for all types of operators. This comprehensive approach is in contrast to most previous work. In addition, the main contribution of this strategy is the incorporation of block building into the parallel cost model.

First, we focus entirely on the cost formulae that are necessary to express the various forms of intra-query parallelism. In order to avoid an unnecessary complication of the resulting expressions, we will make in this first part some simplifying assumptions, listed in the following:

A1. *Send and receive operators don't have local costs of their own.*

Thus, *send* and *receive* operators don't contribute to the total cost of a plan or subplan. Actually, the exchange of intermediate results involves non-negligible communication costs [Has95]. As shown also in Chapter 3, the communication overhead is even more significant if materializing *send* operators are employed. Thus, later on in this chapter (Sections 5.4.2 and 5.4.3) *send* and *receive* operators will be treated in the same way as other "regular" nodes, i.e. as operators that process their input according to the iterator model. In this representation the costs related to intra-query communication, i.e. the management of communication segments (see Chapter 3), can be considered as the operator's local costs.

A2. Unlimited resource availability

For simplification purposes, the cost formulae will be first derived without taking into account the resource consumption of different operators and subplans. That means that in a first step we will concentrate on the response time of a query, i.e. in the following the term *cost* is equivalent

to *response time*. Later on, in Section 5.5, we will also integrate resources into the cost model.

A3. The granularity for intra-query parallelism is one operator.

Thus, we will first examine the special case where all operators are separated by *send*/*receive* operators. In this way, they can be assigned independently to separate execution units. However, the results in Section 3.5 convincingly demonstrate the importance of block building. Hence, we extended the cost model as described in Section 5.3.4 to make the parallelizer cognizant of this aspect as well.

The chapter is organized as follows. Section 5.2 gives a brief overview on related work. In Section 5.3 we gradually derive the significant cost measures. The resulting cost formulae are summarized in Section 5.4. The influence of resources will be presented in Section 5.5. Finally, Section 5.6 provides a detailed description of the *ParPrune* strategy that has been introduced in Chapter 4.

5.2 Related Work

Cost models for parallel query optimization are influenced by the number of possible ways in which resources may be allocated to the execution plans.

Most existing cost models actually schedule operators onto resources and then calculate the response time [LVZ93, LST91, GHK92]. This approach is inadequate for our overall parallelization strategy where the final resource parameters are set by the query execution control component corresponding to the current run-time situation.

Other approaches [STY93, HS93] assume that all resources are used to execute each operator. This is in contradiction to the requirements posed by a multi-user environment.

More recently, the issue of designing efficiently computable and accurate estimator functions for response time in parallel database systems has been addressed in [GI97, GGS96]. However, these approaches are based on the simplifying assumption that each operator is parallelized separately. Hence, they don't support any execution models that implement also block building, as e.g. MIDAS.

5.3 Deriving the Cost Measures

In the following, we define some basic cost variables. Other variables that are needed for the TOPAZ cost model will be introduced gradually.

Given a subplan T with N as the root operator:

- $T_{local}(N)$
  are the local costs of operator N that are necessary to calculate all result tuples. These are intrinsic costs related to the algorithm implemented within the operator. That means that

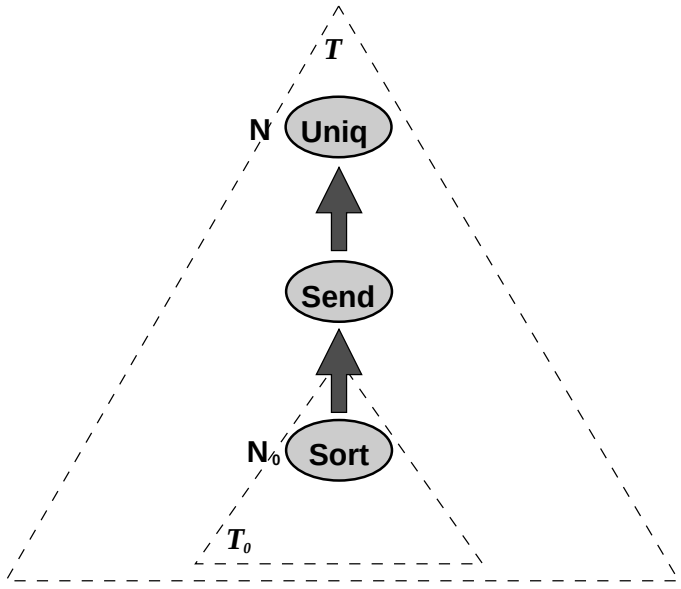


Fig 5.1: Sequential Execution

the costs of the inputs are not contained herein. For instance, in the QEP presented in Fig. 5.1, $T_{local}(N)$ refers only to the local processing costs of the *Uniq* operator.

- $N_0, N_1,...$
  denote the immediate *successors* of an operator *N*, i.e. the root operators of its immediate input subplans $T_0, T_1,...$ As already mentioned, in this first step *send* operators are not taken into account. Please note that in the following, we will use the term of successor and predecessor w.r.t. the position in the tree. Thus e.g. in Fig. 5.1, the *Sort* is the successor of the *Uniq* operator. However, the data flow, respectively processing within a tree, takes place from bottom to top as shown by the arrows in Fig. 5.1. Hence, in the course of evaluation the *Sort* operator transmits intermediate result tuples to its predecessor, the *Uniq* operator.

- $T_{total}(N)$
  are the total costs of subplan *T* with *N* as top operator. This is the time that is necessary to completely evaluate *T*. More precisely, in the iterator model employed also by MIDAS (see Section 2.4), $T_{total}(N)$ relates to the time after which operator *N* has calculated its last results tuple and has transmitted it to its predecessor. Hence, in the sequential execution, $T_{total}(N)$ is the sum of the local costs of operator *N* and the costs of its inputs:

$$T_{total}(N) = T_{local}(N) + \sum_{i=0}^{\kappa} T_{total}(N_i), \text{ where } k = \text{arity of } N.$$

Further on, we concentrate on some characteristic scenarios encountered while employing intra-query parallelism. Thereby, we present typical examples corresponding to each scenario. Based on these examples, we introduce new cost components, respectively develop new formulae. Following the convention used in the previous chapters, *send* and *receive* operators will be often represented as a single logical operator. Please note that the final cost formulae covering all possible scenarios and operators in the MIDAS cost model will be presented in Section 5.4.

5.3.1 Sequential Execution

Fig. 5.1 shows an example for *sequential execution (SE)* between two operators N and N_0 (respectively between the corresponding subplans T and T_0). The *Uniq* operator, a duplicate elimination, is separated by a *send* operator from the subplan T_0 having as a root the *sort* operator N_0. Hence, the two operators can be processed on two separate execution units. However, this strategy is not beneficial for this scenario. The *Uniq* operator can only start processing when its successor (in this case the *sort* operator) has delivered its first result tuple. However, the *sort* operator is a blocking operator, i.e. it has to process all of its input before delivering the first output tuple. At the time when this task is finished, the subplan T_0 is also completely evaluated. Hence, parallelism between N and T_0 is not possible. The two operators N and N_0 are evaluated one after the other, i.e. sequentially.

Given the above, the cost for N amounts to the sum of the total costs of N_0 and the local costs of N:

$$T_{total}(N) = T_{total}(N_0) + T_{local}(N).$$

5.3.2 Independent Parallel Execution

We define *Independent Parallel Execution (IPE)* as a processing scenario where among the implicated operators, respectively subplans, there are no data dependencies. In this case none of the operators is implicated in any waiting situations until some other operators deliver (intermediate) results. Hence, the transmission rate between communicating subplans has no relevance, either.

In Fig. 5.2 a typical example of IPE is given. The two subplans T_0 and T_1 are separated from the common predecessor *merge join* via *send* operators and can thus be scheduled on separate execution units for execution. There is no data dependency between them, because *sort* operators derive their complete intermediate result table before the next operation (in this case the *merge join*) can start. Hence no blocking between the two operators can occur.

We introduce the new binary operator ||, called *"Parallel"* that can be applied for the cost calculation of two operators, respectively subplans, that are executed in an IPE manner. Using this operator, the response time of the entire plan T can be expressed as follows:

$$T_{total}(N) = (T_{total}(N_0) \,||\, T_{total}(N_1)) + T_{local}(N).$$

First, both input subplans of N have to be evaluated completely. The resulting response time is expressed by $(T_{total}(N_0) \,||\, T_{total}(N_1))$, as introduced before for subplans executed in an IPE manner. Only at this point can operator N start processing. Thus, this is also a case of sequential execution between N and its successors. Therefore, the corresponding costs are added up.

For the evaluation of the expression $T_{total}(N_0) \,||\, T_{total}(N_1)$ it is important to assess the influence of resource contention on the final costs. If resources are not taken into account, the following formula is valid:

$$T_{total}(N_0) \,||\, T_{total}(N_1) = max\,(T_{total}(N_0), T_{total}(N_1)).$$

Thus, the total response time is the maximum of the response times of the two inputs. This esti-

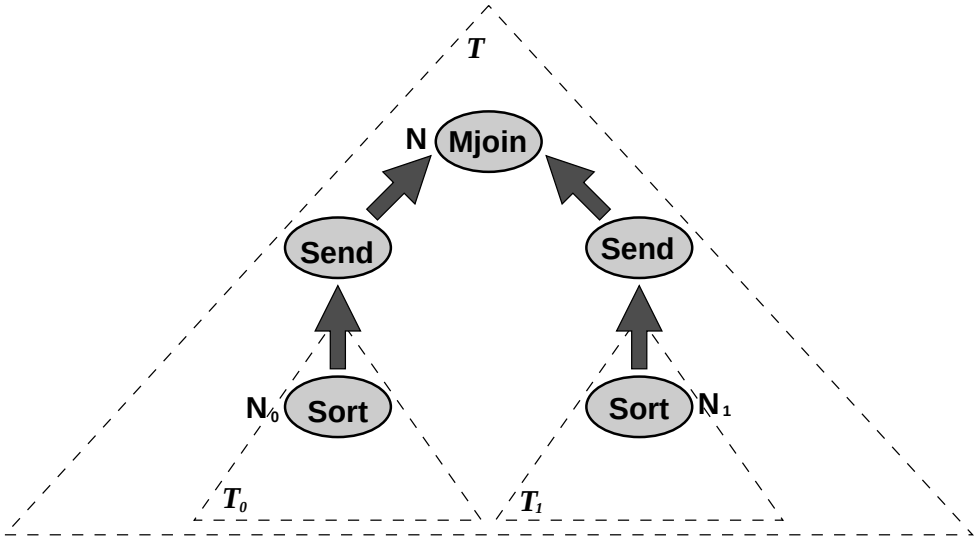


Fig 5.2: Independent Parallel Execution

mation is correct, if T_0 and T_1 access different resources and thus no contention occurs. However, the response time increases if the two subplans compete for (common) resources. In extreme cases IPE can even degenerate to sequential execution. As already mentioned, the treatment of resources will be presented in detail in Section 5.5.1.

Given the above, we can conclude that the cost of $T_{total}(N_0)$ || $T_{total}(N_1)$ lies between the following limits:

$$\underset{IPE}{max\ (T_{total}(N_0), T_{total}(N_1))} <= T_{total}(N_0) \text{ || } T_{total}(N_1) \underset{\text{-->}}{<=} \underset{SE}{T_{total}(N_0) + T_{total}(N_1)}$$

5.3.3 Dependent Parallel Execution

Dependent Parallel Execution (DPE) means that two operators, respectively two subplans are processed on different execution units, but there is a data dependency between them. As shown also in Chapter 3, if the processing rates of communicating subplans are not similar, waiting situations can occur. A typical DPE scenario is pipelining parallelism. Fig. 5.3a shows a simple example for dependent parallel execution. Two operators *N* and N_0, a duplicate elimination *uniq* and restriction *restr,* are split up into two subplans by a *send* operator. Since the *restr* operator is not blocking, it delivers its result tuples immediately to its predecessor and hence parallelism between *N* and N_0 is possible.

However, the actual extent of parallelism is dependent of the dataflow between the two operators. The most important factor to influence the dataflow are blocking boundaries, hence they must be incorporated into the cost model. In order to evaluate the actual response time of an operator, it is important to have knowledge on its processing start time. If its successor N_0 is a blocking operator, as e.g., the *sort* in Fig. 5.1, dependent parallel execution degenerates to sequential execution (cf. Section 5.3.1). However, if N_0 is not a blocking operator and doesn't have to wait for any successors of its own, *N* can start processing without delay. Hence, this situation resembles to some extent to an independent parallel execution. This can be expressed as

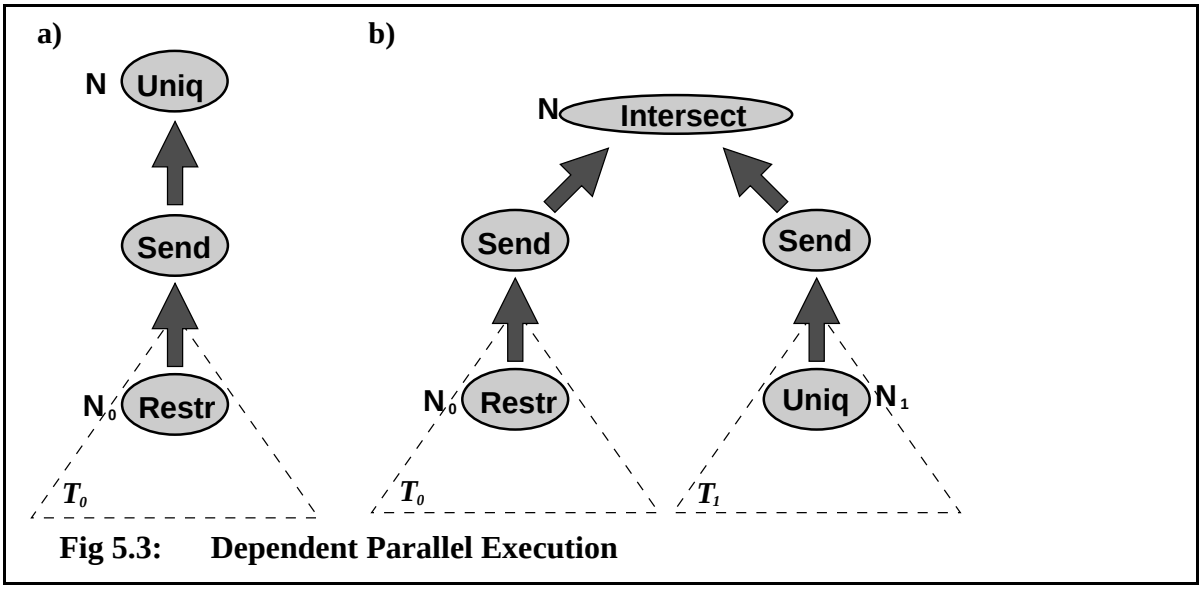


Fig 5.3: Dependent Parallel Execution

follows:

$max\ (T_{total}(N_0), T_{local}(N)) <= T_{total}(N) <= T_{total}(N_0) + T_{local}(N)$
$IPE \quad --> \quad DPE \quad --> \quad SE$

The delay until an operator receives its first tuple from its successor and can in turn start processing, will be further referred to as the costs of its `materialized front` [GHK92]. The materialized front of an operator *N* is the set of input subplans that must be completely evaluated before *N* can start evaluating its first input tuple. The name results from the fact that these are subplans containing blocking operators, hence often a materialization of intermediate results is necessary.

Therefore, we introduce a new cost component, called $T_{begin}(N)$. This component indicates at which time an operator *N* delivers its first (intermediate) result tuple to its predecessor. If *N* is a blocking operator we have:
$T_{begin}(N) = T_{total}(N)$,
because the entire subplan *T* having *N* as a root must be evaluated entirely before the first result tuple is handed over to the preceding operator.

The costs for the subplan in Fig. 5.3a can thus be calculated as follows:

$T_{total}(N) = T_{begin}(N_0) + ((T_{total}(N_0) - T_{begin}(N_0)) \,||\, T_{local}(N))$
$= T_{begin}(N_0) + max((T_{total}(N_0) - T_{begin}(N_0)), T_{local}(N))$
$T_{begin}(N) = T_{begin}(N_0)$ (since *N* is not a blocking operator).

The term $(T_{total}(N_0) - T_{begin}(N_0))$ denotes the remaining processing cost for subplan T_0, that has to be evaluated after N_0 has delivered its first tuple to *N*. If N_0 is a blocking operator, this expression equals to 0, as in this case N_0 would have processed all of its input entirely before delivering the first result tuple.

Fig. 5.3b shows a further example. In this case the inputs of the binary operator *Intersect* are also implicated in a DPE execution. The dependency results here from the processing rate of their common predecessor. This results in:

$T_{total}(N) = (T_{begin}(N_0) \,||\, T_{begin}(N_1)) +$

$$((T_{total}(N_0) - T_{begin}(N_0)) \,||\, (T_{total}(N_1) - T_{begin}(N_1)) \,||\, T_{local}(N))$$
$$= max(T_{begin}(N_0), T_{begin}(N_1)) +$$
$$max(T_{total}(N_0) - T_{begin}(N_0), T_{total}(N_1) - T_{begin}(N_1), T_{local}(N))$$

$T_{begin}(N) = (T_{begin}(N_0) \,||\, T_{begin}(N_1))$

$= max(T_{begin}(N_0), T_{begin}(N_1))$ (since N is not a blocking operator).

The materialized fronts of N_0 and N_1 can be processed in parallel in an IPE manner. The resulting response time is the cost of the materialized front of N. Since N can only start processing if both inputs have delivered their first tuple, $T_{begin}(N)$ is the maximum between the costs of the materialized fronts of the inputs.

The remaining subplans having the processing costs $T_{total}(N_0) - T_{begin}(N_0)$, $T_{total}(N_1) - T_{begin}(N_1)$, respectively $T_{local}(N)$ can now be processed parallely in a pipelining manner. Assuming a constant processing rate, the formula for IPE is here also valid.

Conclusion: In this section, we introduced the cost component $T_{begin}(N)$ that stands for the costs of the *materialized front* of operator N. These costs are caused by *blocking operators* in the PQEP. If resources are not taken into account, $T_{begin}(N)$ is equivalent to the time when operator N has evaluated its first intermediate result tuple.

5.3.4 Block Building

In this section, we extend the cost model to deal also with block building. As explained in Section 3.5, a block can be viewed similarly to a single operator whose processing cost is the sum of the processing costs of the constituting operators. Hence, in this case parallelism is exploited among the various blocks, rather than within a block. Fig. 5.4 shows a QEP that is split up into two blocks. The relation scan operator *rel* and the projection *proj* are processed parallely in a DPE manner with the restriction *restr* and the duplicate elimination *uniq*.

In this case a new cost component, called $T_{process}$, is needed to add up the local costs of the constituting operators within a block. Thus, the blocks are regarded as virtual operators (N_1' and N' in Fig. 5.4) with correspondingly increased processing costs.

The following cases are possible in the bottom-up cost calculation of a plan:

1. N is a leaf or a *receive* operator:
   $T_{process}(N) = T_{local}(N)$
2. else: (the arity of $N = n$, $n > 0$)
   $T_{process}(N) = ((\sum_{i=0}^{n-1} T_{process}(N_i)) + T_{local}(N)$.

Thus, we use a similar cost calculation technique within a block as employed for traditional sequential execution presented in Section 5.1. However, modeling a set of operators within a block as a single virtual operator gives rise to difficulties if the block contains also blocking operators. As already mentioned, in a parallel environment it is important to keep track of the costs of the materialized front(s). These costs are determined by blocking operators and are propagated for further cost calculations by means of the T_{begin} cost component. For a block containing one or several blocking operators we have different values of T_{begin} within a block as

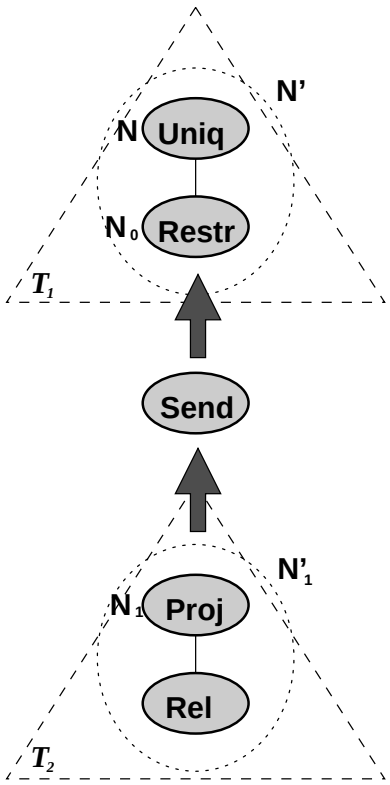


Fig 5.4: Dependent parallel execution of 2 blocks

well. Hence, if this block is modeled as a single virtual operator, the final value of this cost component is undefined. Therefore, we decided to bundle together operators of a block only as far as the next blocking node or *send* operator. In this way, a block containing *m* blocking operators is modeled as *m+1* virtual operators.

Hence, in the bottom-up cost calculation we have:

$T_{process}(N) = 0$, if N is blocking.

If multiple blocks are processed in a DPE manner, the final cost is influenced by the most costly block. Hence, it is important to retain this information during cost calculation. For this reason, the new cost component T_{max} is introduced.

For a given operator N, T_{max} is calculated in the following way:

1. Leaf or blocking operator:
   $T_{max}(N) = 0$
2. *send* operator:
   $T_{max}(N) = (T_{process}(N) \,||\, T_{max}(N_0))$.
3. else: (the arity of $N = n$, $n > 0$)
   $T_{max}(N) = T_{max}(N_0) \,||\, ... \,||\, T_{max}(N_{n-1})$

Hence, for blocking operators T_{max} is reset to 0. This is due to the fact that for predecessors the treatment of DPE-style parallelism below the given (blocking) operator becomes superfluous. If the root of the local block is reached (case 2), T_{max} can be evaluated as being the maximum between the processing cost of the local block ($T_{process}(N)$) and the costs of its successor blocks ($T_{max}(N_0)$). Finally, for all other operators, the inputs are processed in a DPE-type parallelism. Hence, the corresponding formula applies also for the calculation of T_{max} (case 3). This case

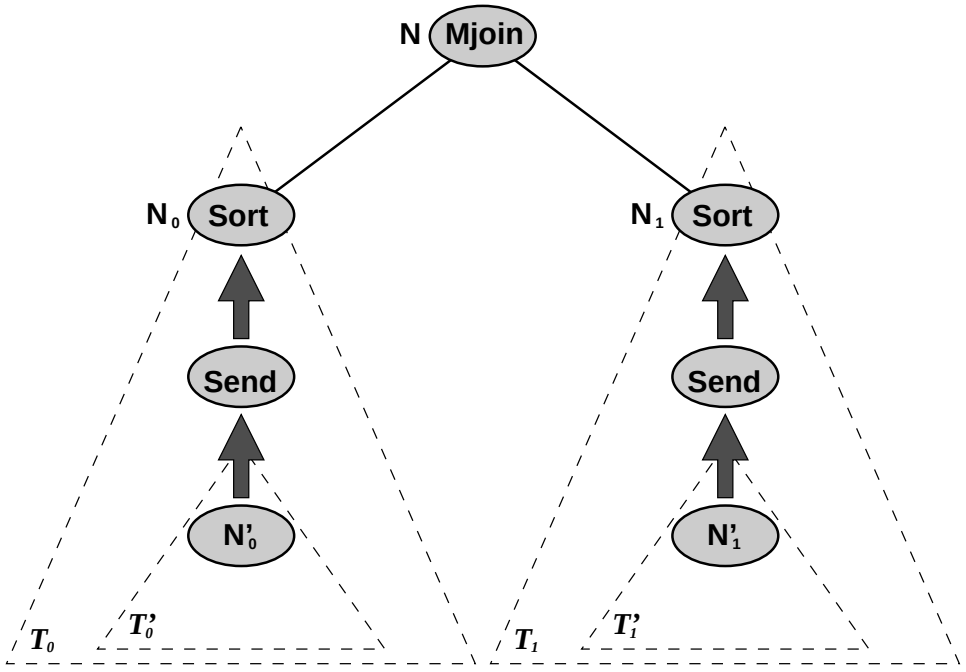


Fig 5.5: Block building with a materialized front

includes also unary operators, for which the value of T_{max} is propagated unchanged.

Given the above, the total cost of a unary operator N results to:

$T_{total}(N) = T_{begin}(N_0) + (T_{process}(N) \,||\, T_{max}(N_0))$.

Block building influences the calculation of the materialized front for binary operators as well. Thus, the formula
$T_{begin}(N) = (T_{begin}(N_0) \,||\, T_{begin}(N_1)) = max(T_{begin}(N_0), T_{begin}(N_1))$, given in Section 5.3.3 is no longer valid.

This can be exemplified using the PQEP in Fig. 5.5. Here, the *merge join* and the two blocking *sort* operators are incorporated within the same block. The costs of the materialized front of N_0 ($T_{begin}(N_0)$) are at least as high as the local costs of the left *sort* (N_0); an analog observation is valid for the costs of the materialized front of the right input N_1. However, the two *sort* operators are not processed in parallel, but sequentially, since they are incorporated within the same block. Hence, it is not correct to use the *max* operator as in the above formula. Using the sum is not correct either, because in this way the costs of the materialized fronts of T_0' and T_1' would also be added up. However, these fronts are executed in parallel. In order to assess the costs in this situation correctly, the new cost component $T_{parallel}$ is introduced. In the bottom-up cost calculation, $T_{parallel}(N)$ stands for the costs of the materialized front of the most recent subplan that runs in parallel with N.

Thus, for the root operator N of a block, i.e. a *send* operator, we have:

$T_{parallel}(N) = T_{begin}(N)$.

In all other cases $T_{parallel}(N) = T_{parallel}(N_0)$

For the example in Fig. 5.5: $T_{parallel}(N_0) = T_{begin}(N_0')$, $T_{parallel}(N_1) = T_{begin}(N_1')$

Thus, the costs for the materialized front of *N* result to:

$T_{begin}(N) = (T_{begin}(N_0) \,||\, T_{parallel}(N_1)) + (T_{begin}(N_1) - T_{parallel}(N_1))$.

The *merge join* requires first its left input tuple, i.e. the left input T_0 constituted of the subplan T_0' and the *sort* N_0 are evaluated first. Meanwhile, the subplan T_1' starts processing in parallel, as well. Thus, the materialized front of N_0 runs in parallel with the materialized front of the parallel input subplan T_1' of N_1. This is reflected by the expression $(T_{begin}(N_0) \,||\, T_{parallel}(N_1))$. Later on, the evaluation continues with the right input of the *merge join*. Thereby, $(T_{begin}(N_1) - T_{parallel}(N_1))$ expresses the remaining costs of the materialized front of N_1.

Conclusion: In this section, the consideration of block building has imposed the introduction of three further cost components:

$T_{process}$ reflects the local costs of a virtual operator. Such virtual operators are obtained by dividing a block into several units, corresponding to the number of blocking operators contained within the block.

T_{max} is introduced to account for the virtual operator with the highest costs in a chain of blocks that are executed in parallel. Obviously, the final cost of a PQEP is mostly influenced by its most expensive operators. The cost component T_{max} is used to model this aspect in the case of block building as well, i.e. for virtual operators.

$T_{parallel}$ is employed to express the costs of materializing fronts that are executed in parallel.

5.3.5 Multiple Evaluation of the Inputs

In this section, we concentrate on binary operators that evaluate their right input in a repeated manner. Examples of such operators in the MIDAS execution model are the *nested loops*, *cartesian product*, the *restriction* and *projection*.

If the right input contains at least one parallel subplan, only the operators belonging to the local block need to be evaluated repeatedly. The detached subplans have to be processed only once, because in the data river concept presented in Chapter 3, the derived intermediate results (managed within communication segments) are available until the end of transaction.

Intuitively, in such a case the following formula could be used:

$T_{process}(N) = T_{local}(N) + T_{process}(N_0) + card0 * T_{process}(N_1)$,
where *card0* is the cardinality of the left input.

However, this formula is not correct if the right input contains within the same block also a blocking operator, like in Fig. 5.6. As defined in Section 5.3.4, $T_{process}(N_1)$ reflects only the costs that come up *above* the materialized front, hence in this case above the *sort* operator. Apart of this, the costs of some operators vary if they are evaluated repeatedly, as e.g. the *correlation* or the *send* operators.

Therefore it is important to introduce a further component, called $T_{again}(N)$, that indicates the costs that are necessary to evaluate the subplan with *N* as a root operator once again.

For the *receive* operator we have:

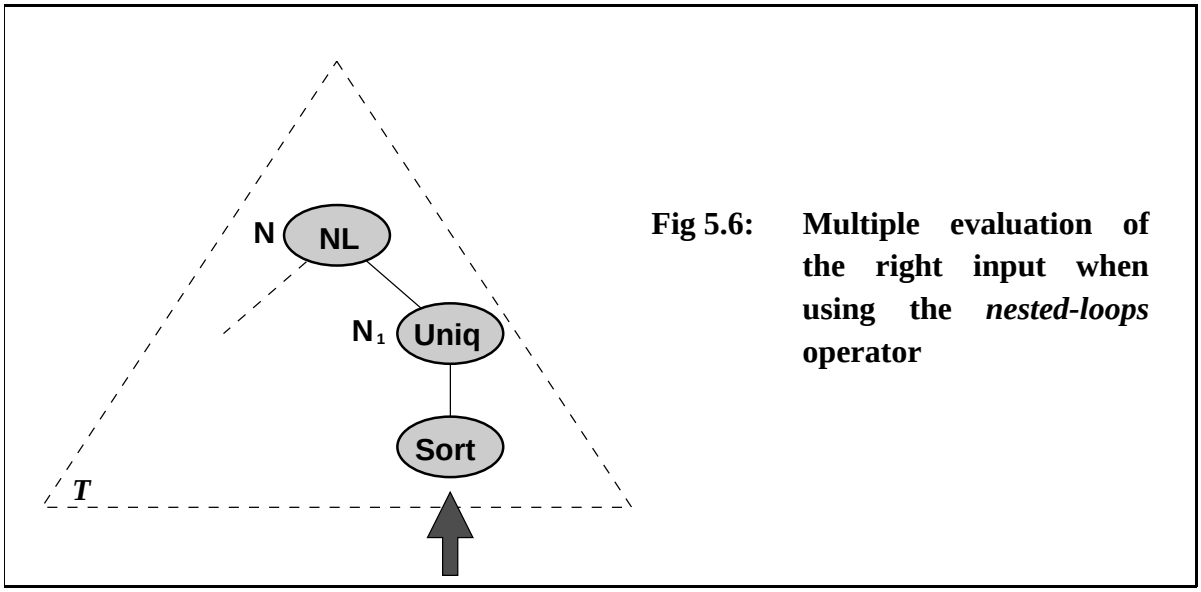


Fig 5.6: Multiple evaluation of the right input when using the *nested-loops* operator

$T_{again}(N)$ = 0, if the previously derived intermediate results are not materialized to disk. Contrary, additional I/O costs occur (see Section 3.3)

Given an operator N with arity n, where the processing costs for a repeated evaluation remain constant. In this case

$$T_{again}(N) = ((\sum_{i=0}^{n-1} T_{again}(N_i)) + T_{local}(N).$$

Please note that in contrast to $T_{process}$ presented in Section 5.3.4, T_{again} is not reset to 0 if N is blocking.

Hence, for the *nested loops* operator in Fig. 5.6, we have:

$$T_{process}(N) = T_{local}(N) + T_{process}(N_0) + T_{process}(N_1) + (card0 - 1) * T_{again}(N_1).$$

Conclusion: As presented in this section, operators that evaluate their input multiple times impose the introduction of a new cost component, called T_{again}. This is due to the fact that in a parallel environment the costs necessary to repeatedly evaluate a subplan are possibly lowered by reused intermediate results.

5.3.6 Materializing Send Operators

A further special case not yet taken into account by the cost model are materializing *send* operators. As defined in Section 3.3.2, these are operators that implement the NOBUF and WRITEOUT strategies and hence write their intermediate results to disk in case of an overflow. In this way, such *send* operators as well as the corresponding blocks are independent of their predecessors. Thus, DPE is modified to IPE, in an analog way as described in Section 5.3.4 for blocking operators, i.e. materialized fronts. However, the difference lies in the fact that, although the NOBUF and WRITEOUT *send* operators are materializing, they are not blocking, because their predecessor can immediately access the intermediate results via a data river.

In order to incorporate this kind of parallelism into the cost model as well, we introduce a further cost component, called T_{end}. This variable remains 0 until the (bottom-up) cost calculation reaches a *materializing send* operator.

At this point, it is equaled to the total costs of the current block.

Thus, for a given operator *N*, T_{end} is calculated in the following way:

1. Leaf operator:
   $T_{end}(N) = 0$
2. unary operator:
   $T_{end}(N) = T_{end}(N_0)$
3. binary operator:
   $T_{end}(N) = (T_{end}(N_0) \,||\, T_{end}(N_1))$
4. *materializing send* operator:
   $T_{end}(N) = T_{total}(N)$ and $T_{max}(N) = 0$.

Then, for the total costs of a node *N* above a *materializing send* operator we have:

$$T_{total}(N) = \begin{cases} T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N) \,||\, (T_{end}(N) - T_{begin}(N))), & \text{if } T_{end}(N) > T_{begin}(N) \\ T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N)), & \text{if } T_{end}(N) <= T_{begin}(N). \end{cases}$$

The term $(T_{end}(N) - T_{begin}(N))$ indicates that after the materialized front of *N* has been processed (reflected by the cost component T_{begin}), *N* can run in an IPE manner with the rest of T_{end} (if $T_{end}(N) > T_{begin}(N)$). For a better understanding, we will illustrate this situation in the following example scenario.

Conclusion: In this section, we have introduced the cost component T_{end}. This reflects the time when a block transmitting its intermediate results via a materializing *send* operator finishes processing.

5.3.7 Example of a Cost Calculation

In this section we will effectuate a cost calculation for the QEP presented in Fig. 5.7 based on the model presented previously. This plan consists of 2 blocks, connected via a *send/receive* operator pair. We assume the total costs of subplan T_0 and T_1 as being $T_{total}(N_0) = 10$, respectively $T_{total}(S) = 15$. The *merge join* has the local costs of $T_{local}(N) = 5$. The left input of the *merge join* is subplan T_0. This has as root the blocking operator *sort*. Therefore, this subplan has to be completely evaluated, before operator *N* requires its first right input tuple. This is expressed by the following:

$T_{begin}(N_0) = T_{total}(N_0) = 10$

$T_{process}(N_0) = 0$

$T_{max}(N_0) = T_{end}(N_0) = 0.$

Assuming that the costs of the materialized front of T_1 are 0, we have:

$T_{begin}(S) = 0$

$T_{process}(S) = 15.$

For simplification purposes, communication costs are ignored in this example. However, please

note that they are also part of the cost model and will be introduced later. In the following, we consider 3 different scenarios.

1. First, we assume that *S* is a *send* operator with a WAIT strategy, i.e. the producer (subplan T_1) is stalled until the consumer (the *merge join*) requests its first right input tuple (cf. Section 3.3.2). Only at this time subplan T_1 starts processing. However, DPE-style parallelism with the *merge join* is possible.

S: $T_{max}\ (S) = T_{process}\ (S) = 15$
$T_{end}\ (S) = 0$
$T_{parallel}\ (S) = T_{begin}\ (S) = 0$

N_1: $T_{begin}\ (N_1) = T_{begin}\ (S) = 0$
$T_{max}\ (N_1) = T_{max}\ (S) = 15$
$T_{end}\ (N_1) = T_{end}\ (S) = 0$
$T_{process}\ (N_1) = 0$

N: $T_{process}\ (N) = T_{process}\ (N_0) + T_{process}\ (N_1) + T_{local}(N) = 0 + 0 + 5 = 5$
$T_{begin}\ (N) = (T_{begin}(N_0) \,||\, T_{parallel}\ (N_1)) + (T_{begin}(N_1) - T_{parallel}\ (N_1)) =$
$(10 \,||\, 0) + (0 - 0) = 10$
$T_{max}\ (N) = T_{max}(N_0) \,||\, T_{max}(N_1) = 0 \,||\, 15 = 15$
$T_{end}\ (N) = T_{end}(N_0) \,||\, T_{end}(N_1) = 0 \,||\, 0 = 0$
$T_{total}(N) = T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N)) =$
$10 + (5 \,||\, 15) = 10 + 15 = \underline{\underline{25.}}$

2. Here, we assume that operator S is a materializing *send* node. In this case, no blocking can occur and subplan T_1 can immediately start processing. Thus, T_0 and T_1 run in an IPE manner. After T_0 has finished, operator *N* can be processed in parallel with the rest of T_1.

S: $T_{max}\ (S) = 0$
$T_{end}\ (S) = 15$

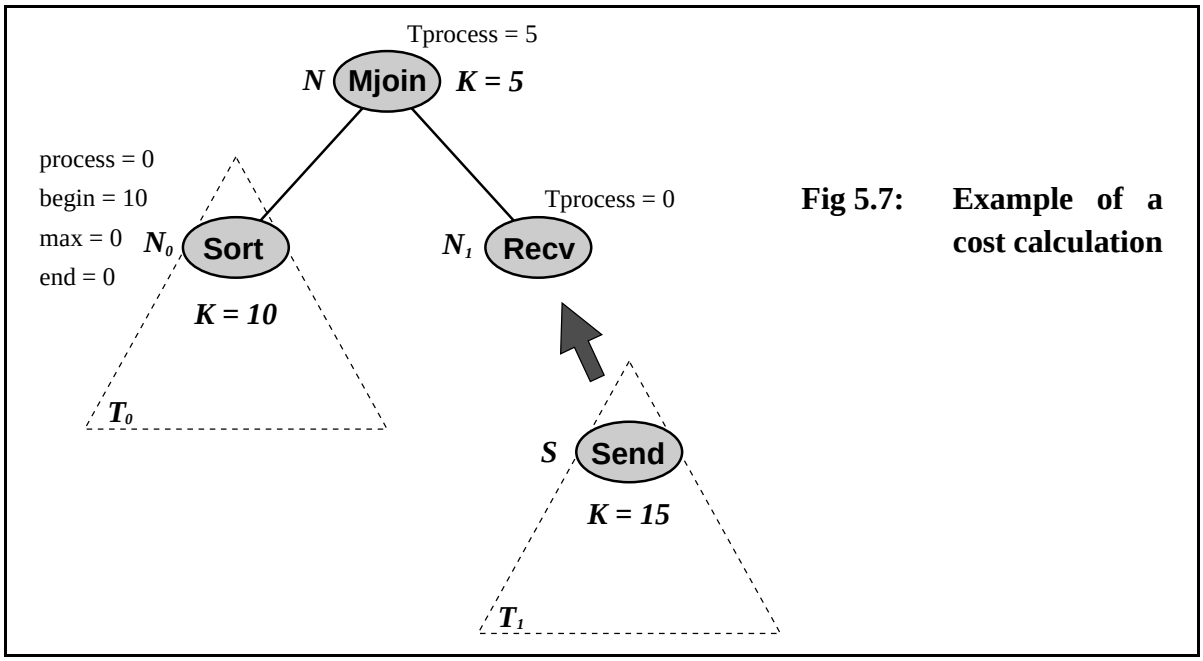


Fig 5.7: Example of a cost calculation

$T_{parallel}(S) = T_{begin}(S) = 0$

N_1: $T_{begin}(N_1) = T_{begin}(S) = 0$
$T_{max}(N_1) = T_{max}(S) = 0$
$T_{end}(N_1) = T_{end}(S) = 15$
$T_{process}(N_1) = 0$

N: $T_{process}(N) = T_{process}(N_0) + T_{process}(N_1) + T_{local}(N) = 0 + 0 + 5 = 5$
$T_{begin}(N) = (T_{begin}(N_0) \,||\, T_{parallel}(N_1)) + (T_{begin}(N_1) - T_{parallel}(N_1)) =$
$(10 \,||\, 0) + (0 - 0) = 10$
$T_{max}(N) = T_{max}(N_0) \,||\, T_{max}(N_1) = 0 \,||\, 0 = 0$
$T_{end}(N) = T_{end}(N_0) \,||\, T_{end}(N_1) = 0 \,||\, 15 = 15$
$T_{total}(N) = T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N) \,||\, (T_{end}(N) - T_{begin}(N)))$
$10 + (5 \,||\, 0 \,||\, 15 - 10) = \underline{\underline{15.}}$

3. This case emphasizes the difference between a blocking operator and a materializing *send* node. Therefore, we assume that subplan T_1 contains a blocking operator, e.g. a *sort*, right below the *send* operator. The materialized front of T_1 is in this case equal to the total costs, hence $T_{begin}(S) = 15$. T_0 and T_1 run in an IPE manner in this case as well**.** However, in contrast to case 2, no parallelism is possible between N and T_1.

S: $T_{max}(S) = T_{process}(S) = 0$
$T_{end}(S) = 0$
$T_{parallel}(S) = T_{begin}(S) = 15$

N_1: $T_{begin}(N_1) = T_{begin}(S) = 15$
$T_{parallel}(N_1) = T_{parallel}(S) = 15$
$T_{max}(N_1) = T_{max}(S) = 0$
$T_{end}(N_1) = T_{end}(S) = 0$
$T_{process}(N_1) = 0$

N: $T_{process}(N) = T_{process}(N_0) + T_{process}(N_1) + T_{local}(N) = 0 + 0 + 5 = 5$
$T_{begin}(N) = (T_{begin}(N_0) \,||\, T_{parallel}(N_1)) + (T_{begin}(N_1) - T_{parallel}(N_1)) =$
$(10 \,||\, 15) + (15 - 15) = 15$
$T_{max}(N) = T_{max}(N_0) \,||\, T_{max}(N_1) = 0 \,||\, 0 = 0$
$T_{end}(N) = T_{end}(N_0) \,||\, T_{end}(N_1) = 0 \,||\, 0 = 0$
$T_{total}(N) = T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N)) =$
$15 + (5 \,||\, 0) = \underline{\underline{20.}}$

An interesting result of this example is that the total costs are highest when a *send* operator with the WAIT strategy is employed. This is due to the lacking independent parallelism between the subplans T_0 and T_1. Since the cost of the materialized front of T_1 is 0, the pipe blocks right from the beginning of the execution. If only response time is taken into account, like in this simple example, the lowest costs result from the usage of a materializing *send* operator. The explanation is that this operator allows both IPE-style parallelism between T_0 and T_1 as well as pipelining parallelism between T_1 and N. However, in reality and as shown in Section 3.3.5, materializing *send* operators implicate non-negligible I/O costs. As already mentioned, in this example

the local costs for the *send* operator have been ignored.

5.3.8 Intra-Operator Parallelism

This section concentrates on the extensions that are necessary to incorporate also intra-operator parallelism into the cost model. As presented in Chapter 3, in the data river paradigm operator instances follow the same iterator model as operators. Thus, the formulae presented until now for SE, IPE and DPE parallelism must not be modified. Intra-operator parallelism rather changes the local costs of operators. In this model, we assume that if an operator (or block) is split up into instances, there is a IPE-style parallelism between them. This assumption is always correct in a *data-driven* consumption, i.e. employing an ASYNCH *receive* operator (see Section 3.3.4), and in the case when materializing *send* nodes are used. Only in the case of a demand-driven consumption in the presence of data skew, IPE can degenerate to DPE. However, as it is extremely difficult to keep track of data skew in intermediate result streams, the cost model will not take this latter aspect into account.

Thus, each instance can be regarded as an operator (respectively block) whose local costs have been lowered correspondingly. At the same time, the costs of the partitioning *send* operators have to be increased. If resources are not taken into account, for an operator *N* split up into *m* instances, the local costs of an instance are:

$T_{local}^{m}(N) = T_{local}(N) / m.$

However, instead of dividing the local costs by the number of partitions, TOPAZ invokes for instances the same cost formula as for regular operators (blocks), but with correspondingly lowered cardinalities. If $card_1,..., card_m$ are the cardinalities of the *m* partitions, for the total cardinality we have:

$card = \sum_{i=0}^{m} card_i.$

Then, the response time of operator *N* is determined by the instance having the highest cardinality. We define $T_{local}(N, card)$ as the local costs of operator *N* when the input cardinality is *card*. Thus, for the total costs of operator *N* having *m* instances we have:

$T_{local}(N) = T_{local}(N, card_1) \,||\, ... \,||\, T_{local}(N, card_m) = max_{i=1}^{m} (T_{local}(N, card_i)).$

As already mentioned, in TOPAZ we assume a uniform data distribution, hence we have:

$T_{local}(N) = T_{local}(N, card/m) \,||\, ... \,||\, T_{local}(N, card/m)$

If resources are not taken into account, this expression is equivalent to:

$T_{local}(N) = T_{local}(N, card/m).$

Please note that in the presence of resource contention this formula provides different results than the expression $T_{local}(N) / m$. Consider as an example a *sort* operator with an input cardinality *card* that exceeds the available buffer size. In this case additional disk spoolings are necessary that are also reflected in the value of $T_{local}(N)$. If intra-operator parallelism is applied, the cardinality of an instance is reduced to *card/m*. In many cases, this lowers or even eliminates any I/O costs completely. Hence, we have $T_{local}(N, card/m) <= T_{local}(N) / m.$

5.4 The Cost Formulae

In the following, we will summarize the cost formulae introduced in the previous sections. We will begin with the formulae for regular operators. *Send/receive* operators as well as operators that process their inputs repeatedly or sequentially will be treated in separate sections. For more accuracy, we introduce some further cost components:

- $T_{lbegin}(N)$ reflects the local start-up costs of operator N. A typical example is a table scan, where the delay is caused by the operations necessary to physically open a file.
- The actual processing costs of an operator are given by $T_{lprocess}(N)$. Thus, in the usual case we have $T_{local}(N) = T_{lbegin}(N) + T_{lprocess}(N)$.
- $N_{EU}(N)$ defines the number of execution units that are necessary to process the subplan having operator N as root. This component will come to application in Section 5.5.2.

All cost components and constants are summarized once more in Table 5.1 and Table 5.2.

Table 5.1 Constants used for the cost calculation

*card*0	Cardinality of the (left) input of *N*
card	Cardinality of the output of *N*
pages	Size of the output relation in MIDAS pages
cachesize	Number of database buffer pages
t_{disk}	Time to read/write a MIDAS page to disk

For blocking operators we introduce the operation *sync(N)*. This operation summarizes all settings corresponding to this operator type as described in Section 5.3

sync(N):

$$T_{begin}(N) = T_{total}(N)$$
$$T_{process}(N) = 0$$
$$T_{max}(N) = 0$$
$$T_{end}(N) = 0$$

Table 5.2 Cost variables

$T_{total}(N)$	Total costs of the subplan having *N* as a root operator
$T_{lprocess}(N)$	Local processing costs of operator *N*
$T_{lbegin}(N)$	Local start-up costs of operator *N*
$T_{local}(N)$	Total local costs of operator *N*
$T_{begin}(N)$	Costs of operator *N* that are necessary to deliver its first (intermediate) result tuple to its predecessor
$T_{process}(N)$	Local processing costs of the virtual operator constituted from *N* and all subsequent nodes, as far as the next *send* or blocking operator
$T_{again}(N)$	Costs for the repeated execution of the subplan having *N* as a root operator

Table 5.2 Cost variables

$T_{parallel}(N)$	Costs of the materialized front of the last parallel subplan below *N*; corresponds to the time when the first tuple is delivered to the block having *N* as a root operator
$T_{max}(N)$	Processing costs of the subplans running parallel to *N* in a DPE manner
$T_{end}(N)$	Total costs of the last subplan below *N* that contains a materializing *send* operator
$N_{EU}(N)$	Number of execution units to process the subplan having operator *N* as root

5.4.1 Regular operators

This section details the cost formulae for regular operators. This category includes the main part of the operators in the MIDAS cost model. The term “regular” means that they don’t necessitate a special treatment w.r.t. their cost calculation. For a better understanding, they are divided into different subclasses, corresponding to their arity. Please note that both the formulae as well as the order in which they are listed correspond to a bottom-up cost calculation.

5.4.1.1 Leaf operators

$$T_{local}(N) = T_{lbegin}(N) + T_{lprocess}(N)$$
$$T_{total}(N) = T_{local}(N)$$
$$T_{begin}(N) = T_{lbegin}(N)$$
$$T_{process}(N) = T_{lprocess}(N)$$
$$T_{again}(N) = T_{total}(N)$$
$$T_{parallel}(N) = 0$$
$$T_{max}(N) = 0$$
$$T_{end}(N) = 0$$

If *N* is a blocking operator: *synch(N)*

5.4.1.2 Unary operators

$$T_{local}(N) = T_{lbegin}(N) + T_{lprocess}(N)$$
$$T_{process}(N) = T_{process}(N_0) + T_{lprocess}(N)$$
$$T_{again}(N) = T_{again}(N_0) + T_{local}(N)$$
$$T_{begin}(N) = T_{begin}(N_0) + T_{lbegin}(N)$$
$$T_{parallel}(N) = T_{parallel}(N_0)$$
$$T_{max}(N) = T_{max}(N_0)$$
$$T_{end}(N) = T_{end}(N_0)$$

$$T_{total}(N) = \begin{cases} T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N) \,||\, (T_{end}(N) - T_{begin}(N))), & \text{if } T_{end}(N) > T_{begin}(N) \\ T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N)), & \text{if } T_{end}(N) <= T_{begin}(N). \end{cases}$$

If *N* is a blocking operator: *synch(N)*

5.4.1.3 Binary operators

$$T_{local}(N) = T_{lbegin}(N) + T_{lprocess}(N)$$
$$T_{process}(N) = T_{process}(N_0) + T_{process}(N_1) + T_{lprocess}(N)$$
$$T_{again}(N) = T_{again}(N_0) + T_{again}(N_1) + T_{local}(N)$$
$$T_{begin}(N) = (T_{begin}(N_0) \,||\, T_{parallel}(N_1)) + (T_{begin}(N_1) - T_{parallel}(N_1)) + T_{lbegin}(N)$$
$$T_{parallel}(N) = T_{parallel}(N_0)) \,||\, T_{parallel}(N_1))$$
$$T_{max}(N) = T_{max}(N_0) \,||\, T_{max}(N_1)$$
$$T_{end}(N) = T_{end}(N_0) \,||\, T_{end}(N_1)$$

$$T_{total}(N) = \begin{cases} T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N) \,||\, (T_{end}(N) - T_{begin}(N))), & \text{if } T_{end}(N) > T_{begin}(N) \\ T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N)), & \text{if } T_{end}(N) <= T_{begin}(N). \end{cases}$$

If *N* is a blocking operator: *synch(N)*

5.4.1.4 Operators having arity n (n >= 2)

$$T_{local}(N) = T_{lbegin}(N) + T_{lprocess}(N)$$
$$T_{process}(N) = (\sum_{i=0}^{n-1} T_{process}(N_i)) + T_{lprocess}(N)$$
$$T_{again}(N) = (\sum_{i=0}^{n-1} T_{again}(N_i)) + T_{local}(N)$$

$$T_{begin}(N) = T_{begin}(N_0)$$

for i:= 1 TO n-1 do

$$T_{begin}(N) = (T_{begin}(N) \,||\, T_{parallel}(N_i)) + T_{begin}(N_i) - T_{parallel}(N_i))$$

end

$$T_{begin}(N) = T_{begin}(N) + T_{lbegin}(N)$$
$$T_{parallel}(N) = T_{parallel}(N_0) \,||\, ... \,||\, T_{parallel}(N_{n-1})$$
$$T_{max}(N) = T_{max}(N_0) \,||\, ... \,||\, T_{max}(N_{n-1})$$
$$T_{end}(N) = T_{end}(N_0) \,||\, ... \,||\, T_{end}(N_{n-1})$$

$$T_{total}(N) = \begin{cases} T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N) \,||\, (T_{end}(N) - T_{begin}(N))), & \text{if } T_{end}(N) > T_{begin}(N) \\ T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N)), & \text{if } T_{end}(N) <= T_{begin}(N). \end{cases}$$

If *N* is a blocking operator: *synch(N)*

5.4.2 Send Operators

As already mentioned, *send* and *receive* operators are internally represented as a single operator, as they always appear in pairs. However, for the cost calculation, they are regarded as two (virtual) independent operators that belong to two different blocks, with corresponding cost formulae. This treatment has been already illustrated by the example given in Fig. 5.7.

The *send* operator is the root of the current block. Thus, when the bottom-up cost calculation reaches this operator, the evaluation of the entire block is finished as well. If the *send* operator is materializing, the block cannot be stalled by predecessors, hence the cost component T_{end} is equal to the total cost of the block and the variable T_{max} can be reset to 0.

By definition, the component T_{begin} indicates when the current operator, i.e. *send*, delivers its first tuple to its predecessor (in this case the *receive* operator). However, as described in Section 3.3.2. the granule of dataflow in MIDAS is a page. Hence, the delay for the first input tuple of the *receive* operator is additionally increased by the time necessary to fill up a MIDAS page. In order to assess this additional delay, the total processing costs of the current operator have to be divided by the number of MIDAS pages that have to be transmitted.

As for the other cost components, the variable $T_{parallel}$ is set to the same value as T_{begin}, according to Section 5.3.4 for *send* operators. The component T_{again} is not relevant, as *send* operators cannot be reset and repeatedly evaluated.

The above considerations can be summarized as follows:

$T_{local}(N) = T_{lbegin}(N) + T_{lprocess}(N)$
$T_{process}(N) = T_{process}(N_0) + T_{lprocess}(N)$
$T_{begin}(N) = T_{begin}(N_0) + T_{lbegin}(N)$
$T_{endprocess} = T_{end}(N_0) - T_{begin}(N)$
$T_{begin}(N) = T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N_0) \,||\, T_{endprocess}) / pages$
$T_{parallel}(N) = T_{begin}(N)$

1. if N is materializing:

$T_{total}(N) = T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N_0) \,||\, T_{endprocess})$
$T_{max}(N) = 0$
$T_{end}(N) = T_{total}(N)$

2. else:

$T_{max}(N) = T_{process}(N) \,||\, T_{max}(N_0)$
$T_{end}(N) = T_{end}(N_0)$
$T_{total}(N) = T_{begin}(N) + (T_{max}(N) \,||\, T_{endprocess})$

Please note that we used the term $T_{endprocess}$ as an auxiliary variable to express the remaining work after the materializing front of N has been processed.

5.4.3 Receive Operators

Since *receive* operators are leaf operators of inner blocks, in a bottom-up evaluation they initiate the cost calculation for the subsequent block. Therefore, the local processing costs of a block, reflected by $T_{process}$, are set to the local processing costs of the *receive* operator.

For the calculation of T_{again} we make the following approximation: if the size of the input relation (in MIDAS pages) is less or equal to the database buffer size, we assume that no materialization of the intermediate results is necessary, hence no costs occur. Otherwise, the pages have to be read from disk, that in turn implicates I/O costs.

$T_{local}(N) = T_{lbegin}(N) + T_{lprocess}(N)$
$T_{process}(N) = T_{lprocess}(N)$
$T_{begin}(N) = T_{begin}(N_0) + T_{lbegin}(N)$
$T_{parallel}(N) = T_{parallel}(N_0)$

$T_{max}(N) = T_{max}(N_0)$

$T_{end}(N) = T_{end}(N_0)$

$T_{total}(N) = T_{begin}(N) + (T_{process}(N) \,||\, T_{max}(N) \,||\, (T_{end}(N) - T_{begin}(N)))$

1. if *pages > cachesize*

$T_{again}(N) = pages * t_{disk}$

2. else

$T_{again}(N) = 0$

5.4.4 The Operators Restriction, Projection, Nested Loops and Cartesian Product

The operators *nested loops* and *cartesian product* perform for each left input tuple a complete and repeated evaluation of their right input. In MIDAS, the *restriction* and *projection* follow the same execution model to evaluate their predicates. The total costs are therefore mostly influenced by the cardinality of the left input and the costs of the right input subplan. The operators have minimal internal tuple processing costs, i.e. minimal local costs of their own. Hence, for the calculation of the component T_{local} we will only consider the costs that result from the repeated execution of the right input.

Thus, following changes to the formulae given in Section 5.4.1.3 are necessary:

$T_{local}(N) = T_{lbegin}(N) + T_{lprocess}(N) + (card0 - 1)*T_{again}(N_1)$

$T_{process}(N) = T_{lprocess}(N) + T_{process}(N_0) + T_{process}(N_1) + (card0 - 1)*T_{again}(N_1)$

$T_{again}(N) = T_{lbegin}(N) + T_{lprocess}(N) + T_{again}(N_0) + card0 *T_{again}(N_1)$

5.4.5 Hash Joins

The *hash join* operator processes its inputs sequentially. First, the build relation is completely

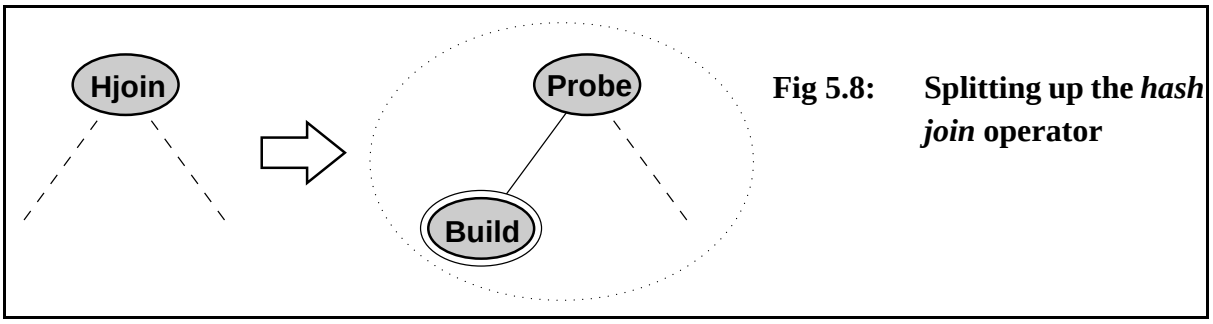


Fig 5.8: Splitting up the *hash join* operator

evaluated, followed by the (mostly larger) probe relation. During the construction of the hash table, no tuples are transmitted to the predecessor. Only in the probe phase pipelining parallelism becomes possible. Hence the *hash join* is a `partly blocking` operator.

In order to use the ability of the TOPAZ cost model to differentiate between blocking and non-blocking operators, the *hash join* is split up into two virtual nodes: a *build* and a *probe* (cf.

Fig. 5.8). As mentioned in Section 5.4.2, *send/receive* operators are treated in an analogous way. For the *hash join*, the *probe* can be regarded as a regular and the *build* as a blocking operator. Thus, the formulae given in Section 5.4.1.2 apply correspondingly.

5.5 Resources

We now extend the cost model to take into account also the resource consumption of operators, respectively subplans. The most important resources are CPU, disk, network communication as well as main memory. Please note that if different parts of a query are processed in parallel and they access the same resources, the independency assumption is no longer satisfied. For a correct calculus, this aspect has to be modeled correspondingly.

Resource consumption has an even more important role in a multi-user scenario. However, as already mentioned, it is the task of the query execution control (QEC), to account also for the concrete run-time environment. TOPAZ performs its cost calculations for given configurations. These appear as parameters in the final plan, together with the corresponding cost values. At query execution time, the QEC analyzes the given runtime environment and decides on the final resource parameters (like memory consumption and degree of parallelism).

However, if the workload is approximately known in advance, in terms as e.g. number of users, typical resource consumption of a query etc., it is also possible to tune TOPAZ correspondingly. This can be done by either increasing the costs estimated for resource consumption or by reducing the available resources for each query. Then, the cost calculations will be performed only for these environments, thus reducing optimization overhead.

5.5.1 Resource Usage Model

The resource utilization of operators and subplans can be modeled by the following pair for each resource:
(t, w), where *t* is the time after which the resource is freed and *w* is the effective time during which the resource it used.

Thus, we always have $t >= w$. If $t > w$, the resource is not utilized permanently. Since it is extremely hard to assess the concrete utilization times, we assume that the resources are used uniformly during period *t*. We further assume that all resources can be used in a *time-sharing* mode. That means that if for a given usage *(t, w)*, we have $t > w$, the resource can be used by other plans during a period $t - w$.

The main memory is treated in a special way. This resource is not preemptive, i.e. it cannot be shared by multiple execution units. Therefore, the uniformity assumption does not apply for this resource. As operators have different memory requirements, the resource consumption is variable during the execution of a plan. For scheduling purposes, it is also important to operate not only with the maximum memory requirement of a query, but with more exact distributions (see Chapter 7). Therefore, we decided to perform a given parallelization task under certain memory

limits. This is treated as a constraint and the available memory is divided proportionally on the constituting operators. If the available memory is not sufficient for specific operators, materializations of intermediate results are necessary. This is reflected by an increased total response time. However, as mentioned above, if the typical workload is not known in advance, for a single query multiple parallelization tasks are performed, also with varying memory limits. The results are then transmitted to the QEC as parameters. Depending on the run-time environments, the QEC may favor for instance a plan having a higher response time but a lower memory consumption.

If an operator or subplan using resources $R_1,...,R_n$, has a response time of t, the corresponding costs can be modeled by a resource vector:
$\vec{r} = (t, \vec{w})$,
where t is the response time of the subplan and the vector $\vec{w} = (w_1,...,w_n)$ reflects the effective utilization time for each resource $R_1,...,R_n$. Thereby, we have the following invariant:
$t >= max \int_{i=1}^{n} (w_i)$,
meaning that the total response time takes at least as long as the most costly resource.

In this way the cost model can be modified to take also resource usage into account. This can be done by substituting in the cost formulae from Section 5.4 the pure response time with a resource vector $\vec{r} = (t, \vec{w})$ that reflects also the resource consumption. Thus, all cost components, such as T_{begin}, $T_{process}$, become also resource vectors. However, in order to leave the formulae given in Section 5.4 unchanged, it is necessary to extend the definition of the appropriate operations on resource vectors as well.

Given two resource vectors $\vec{r}_1 = (t_1, \vec{w}_1)$ and $\vec{r}_2 = (t_2, \vec{w}_2)$ with $\vec{w}_i = (w^i_1,...,w^i_n)$ denoting the work of n resources $R_1,...,R_n$, we define the following operations:

- **Operation +**
  $\vec{r}_1 + \vec{r}_2 = (t_1 + t_2, \vec{w}_1 + \vec{w}_2) = (t_1 + t_2, (w^1_1 + w^2_1,...,w^1_n + w^2_n))$.
  Hence, for the addition of two cost components the response times t_1 and t_2 are added up as usual. In addition, since the total resource consumption also increases, for each resource R_i the effective utilization times are also summed up.

- **Operation -**
  $\vec{r}_1 - \vec{r}_2 = (t_1 - t_2, \vec{w}_1 - \vec{w}_2) = (t_1 - t_2, (w^1_1 - w^2_1,...,w^1_n - w^2_n))$.
  The difference is treated in an analogous way as the addition. In this way, the resource consumption of the remaining subplan is taken into account correctly.

- **Operations * and /**

  These operations are defined only between resource vectors and scalars. Thus, given a scalar value f, we have:
  $\vec{r}_1 * f = f * \vec{r}_1 = (f * t_1, f * \vec{w}_1) = (f * t_1, (f * w^1_1,...,f * w^1_n))$.
  $\vec{r}_1 / f = (t_1 / f, \vec{w}_1 / f) = (t_1 / f, (w^1_1 / f,...,w^1_n / f))$.

- **Operation “Parallel” (||)**

  Given two operators (respectively subplans) that are involved in an independent parallel execution (see Section 5.3.2). Until now, we have considered for the calculation of the total

response time the maximum between the separate response times. However, this is only correct if the two operators access different resources or if a common resource is accessed at different times. By using vectors as defined before, we also have an information on resource utilization times. This information is important, as during a given time period, each resource can be used in a time-sharing mode by multiple operators (subplans). The total response time and the total resource utilization can then be calculated correctly by means of the operation ||:

$\vec{r}_1 \,||\, \vec{r}_2 = (t, \vec{w})$,

where $\vec{w} = \vec{w}_1 + \vec{w}_2$ and $t = max(t_1, t_2, max_{i=1}^{n}(w_i))$.

The resource consumption of the two operators (subplans) is thus added up for each separate resource, similarly as for the addition. However, the total response time is derived by building the maximum over the two separate response times and the highest resource consumption. In this way, the invariant $t >= max_{i=1}^{n}(w_i)$ is always guaranteed as well.

Consider the following two examples for two resources R_1 and R_2:

Example 5.1: $\vec{r}_1 = (10, (5, 7))$; $\vec{r}_2 = (7, (5, 2))$

$\vec{r}_1 \,||\, \vec{r}_2 = (10, (10, 9))$

In this case, the response time of the first query is long enough to cover both resource utilization times. Hence, the total response time is determined only by this component.

Example 5.2: $\vec{r}_1 = (10, (5, 7))$; $\vec{r}_2 = (7, (5, 6))$

$\vec{r}_1 \,||\, \vec{r}_2 = (max(10, 7, max(10, 13)), (10, 13)) = (13, (10, 13))$

In this case, contention has occurred during the utilization of the second resource. Thus, the two subplans can no longer operate independently. This is reflected by the increased total response time that in this case is mostly influenced by the consumption of the second resource.

With the operations presented, the TOPAZ cost model calculates for each operator in the execution plan the cost variables T_{local}, T_{total} etc. presented in Table 5.2. These variables are modeled as resource vectors. The calculation is performed separately for each degree of parallelism and the results are kept in an array, as presented in Section 4.4.4. The degrees of parallelism considered is limited by the *ParPrune* strategy (see Section 5.6).

5.5.2 Defining the Pruning Metric

The total costs for an operator N, respectively for a subplan having N as a root operator are given by the component $T_{total}(N)$. As introduced before, this is now constituted as a resource vector $(t, \vec{w})$ having the following form:

$T_{total}(N) = (t, (w_{cpu}, w_{io}, w_{comm}))$,

where t is the response time of N, w_{cpu} reflects the processor costs, w_{io} the effective I/O time and w_{comm} reflects the actual communication cost[1].

However, query optimizers, as the Cascades Optimizer Framework (see Section 4.3), deal with

cost values over which a total order can be defined. This is important for pruning decisions, i.e. on which plan to eliminate from the search space. However, when dealing with multidimensional cost values, as the resource vector in our cost model, there exists no pruning metric that provides a total order. In [GHK92], an *l*-dimensional compare function is proposed to provide a partial order in *l* dimensions. However, in this case, instead of keeping *one* plan for each subset of relations (and eventually for each required physical property), it is necessary to keep a *set* of (incomparable) plans. This makes the search complexity unacceptable. As a conclusion [GHK92] propose to have a limited number of dimensions in the pruning metric, or even to ignore some resources.

In contrast, our solution is based on transforming the resource vector into a single `cost value`. Thereby, it is important to account for both response time and resource contention. Thus, if a parallel plan shows only a minimal speedup, but a significant overhead as compared to the sequential case, it is more favorable from the throughput point of view to choose the sequential plan. Therefore, for the calculation of the cost value, all three components, namely the response time *t*, the resource consumption $\vec{w}$ and the number of execution units, have to be taken into consideration.

The formula for the calculation of the cost value *v(N)* in TOPAZ is the following:

$$
\begin{aligned}
v(N) = \; & (t + (cpu + io + comm) * RESOURCE\_FACTOR) * \\
& (1 + (N_{EU} / N\_CPUS) * EU\_FACTOR), \\
\text{where} \quad & t := T_{total}(N).t \\
& cpu := T_{total}(N).w_{cpu} \\
& io := T_{total}(N).w_{io} \\
& comm := T_{total}(N).w_{comm}. \\
& N\_CPUS = \text{number of available processors} \\
& RESOURCE\_FACTOR, EU\_FACTOR = \text{tuning factors.}
\end{aligned}
$$

The number of execution units N_{EU} is divided by the number of available processors. Thus, the influence of this component is only significant, if the PQEP requires a high number of execution units as compared to the total number of processors. The influence of the resources can be further varied by means of the *RESOURCE_FACTOR* and *EU_FACTOR* components. Please note that if the value of these factors is 0, *v(N)* is equivalent to the response time.

5.6 The *ParPrune* Strategy

This section describes the way by which the quality of the plans derived by the parallelizer can be influenced by an appropriate analysis that precedes the actual parallelization task. As presented in Section 4.7, the influence of this pre-analysis is twofold:

1. First, the parallelization complexity in terms of tasks, generated expressions and applied rules can be reduced considerably. This is due to the fact that only relevant regions of the

1. The main memory is not included into the vector, because memory, as described above, is treated as a constraint during parallelization.

search space are analyzed in detail. These regions are characterized by the fact that the plan portions that cause the highest costs are parallelized in the most favorable way, i.e. the influence of parallelization is maximized. Hence, during the pre-analysis it is important to detect these portions and to eliminate those subplans from consideration that e.g. have no influence on the critical path length.

2. An even more important reason for the usage of a pre-analysis is the fact that in this way the generated plans show a significantly reduced resource consumption. As shown in Section 4.7, for instance the number of execution units could be reduced considerably.

This second aspect is due to the *independency assumption* of query optimizers. Thus, in order to reduce the search complexity through adequate pruning techniques, optimizers assume that if two plans differ only in a subplan, then the plan with the better subplan is also the better plan. Hence, given a specific plan portion, it is sufficient to find and keep the best plan corresponding this portion, and the other variants can be pruned from further search space exploration. In the parallel search space, this has the consequence that *locally* optimal plans make the best profit out of resource allocation, even if they don't contribute in a significant way to the reduction of some *global* aspects, as e.g. critical plan length. More precisely, in a parallel search space data dependencies and resource contention between parts of a plan are crucial factors that violate the independency assumption.

Therefore, in order to use the search strategy of query optimizers also for parallelization purposes it is important to use some *constraints* in the optimization process. These are derived by the calculation of some global measures of the query execution plan and will be presented in the following sections.

5.6.1 Average Cost per CPU

First, the average cost per CPU (*avg_cost*) is calculated. Thereby, the costs of concurrently running operators are added up and divided by the number of available processors. In Phase 1 of the parallelization process, inter-operator parallelism is only applied to subplans whose total cost exceed *avg_cost*. As mentioned in Section 4.5.1, the goals hereby are to achieve mutually adjusted processing rates over all blocks and to reduce the critical path length through inter-operator parallelism. The global measure *avg_cost* prohibits the construction of blocks having much too dissimilar costs. As explained above, these would come to existence due to the independency assumption of optimizer technology. Thus, keeping the locally optimal plan for each portion leads to the construction of fine-granular blocks at the beginning of the parallelization process (when all resources are available). This has as a consequence that the lack of resources in the subsequent parallelization prohibits the introduction of parallelism in the rest of the plan.

The strategy can be exemplified by the QEP in Fig. 5.9. Here, the local costs are given as numbers near the corresponding operators. The total costs of the plan are *80*. Assuming that the number of available processors is *4*, the value of *avg_cost* results into *80 / 4= 20*.

Hence, in Phase 1 the introduction of inter-operator parallelism is considered only in combination with the edges *A* to *F*. Finally, the application of the cost model determines the introduction

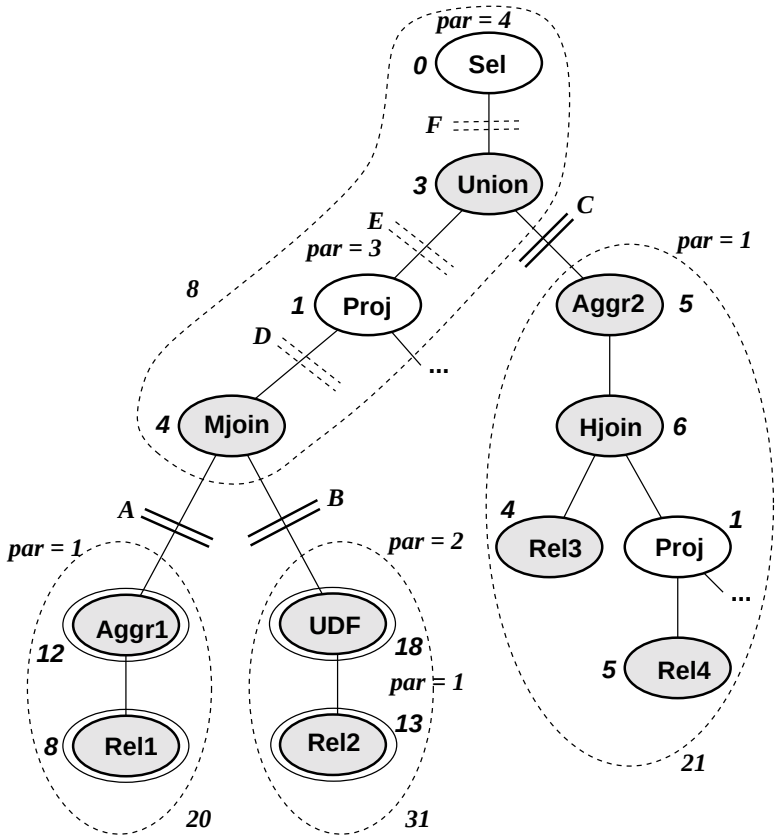


Fig 5.9: The *ParPrune* Strategy

of pipes only on the edges *A, B* and *C*. Without the *ParPrune* strategy, the search engine would try to apply pipelining parallelism on all edges, meaning an increased parallelization overhead. In addition, the quality of the plan would also be influenced. For example, due to the independency assumption as explained above, the parallelization could have ended with the introduction of *send-receive* operators above the *hash join* operator as well.

5.6.2 Average Cost per Operator

Global considerations are important for the application of intra-operator parallelism as well. As shown in Section 4.5.2, global execution performance and critical path length are mostly influenced by nodes having high local processing costs. In order to find these operators, TOPAZ calculates during the pre-analysis the average cost per operator (*avg_nodecost*) by dividing the total costs of the QEP through the number of operators. However, for the calculation of the average only those operators are involved whose local costs are not negligible. Thus, for instance operators denoting a predicate, like *eq, attr* etc., are not taken into account.

In Phase 2 of TOPAZ, the parallelization rules apply only to operators whose costs exceed *avg_nodecost*. For the example QEP in Fig. 5.9, the global measure *avg_nodecost* results to *80/10=8*, since for the division only the operators marked by a gray background are taken into consideration. In this way, there remain *4* driver nodes whose costs exceed the limit of *8*, marked in Fig. 5.9 by a bounding circle. Through *ParPrune*, only these operators will be involved in

Phase 2 of the parallelization process.

5.6.3 Maximal Degree of Parallelism

The driver nodes for the consideration of intra-operator parallelism are derived through the measure *avg_nodecost*. However, the calculation of the corresponding degrees of parallelism is still an open problem. The independency assumption leads here also to a local view of each separate plan portion, having as an effect that for each operator or block the locally optimal degree of parallelism is chosen. This in turn leads to an altogether suboptimal resource distribution in the query execution plan. Hence, it is important to find a mechanism that limits the DOP for operators, respectively subplans.

In TOPAZ, this is done through the measure *maximal degree of parallelism (par)*, that is derived for each operator *N* according to the costs of the subplan that has *N* as root. Thereby, the degree of parallelism is chosen proportionally to the total costs of the plan and the available number of CPUs:

$par\ (N) = N\_CPUS * T_{total}(N) / T_{total}$,
where *N_CPUS =available number of processors* and T_{total} *= total cost of the entire QEP.*

During the cost calculation, the degree of parallelism for each operator *N* is considered only up to the measure *par(N)*. From this set of degrees, the cost model chooses the best one, even if it differs from the locally optimal degree of parallelism.

The mechanism is exemplified in Fig. 5.9. We will derive the value of *par* in a top-down manner, marking only those locations where the value of *par* changes. For the top operators *Sel* and *Union* the cost of the subplan having these operators as root is 80, corresponding to the total cost of the QEP. Thus, the maximal degree of parallelism can be allocated, that is equal to the number of available processors: *par = 4 * 80 / 80 = 4.*

The cost of the two input subplans of the *Union* operator are *56* (left input) and *21* (right input). Hence, the value of *par* for the left input operator, *Proj*, is calculated as follows:
*par = 4 * 56 / 80 = 2,8*, that is rounded to *3*.

For the right input, we have:
*par = 4 * 21 / 80 = 1,05 = 1*.

The calculation is performed in an analog way for all operators. As results from Fig. 5.9, in this way from the 4 potential driver nodes derived in the last section, there remains only one operator, namely the user-defined function UDF, that will be parallelized separately in Phase 2. The corresponding maximal degree of parallelism is 2. However, as explained in Section 4.5.3, through block expansion, this parallelization can be extended downwards. If this is considered to be favorable by the cost model, the resulting block holds the operators *UDF* and *Rel2* and has a maximal degree of parallelism of 2. On the other hand, the parallelization can be also extended upwards, thus involving also the *Mjoin* operator. The resulting block can already have a DOP of up to 3, as given by the *par* value of the topmost operator (in this case the *Mjoin* operator). This process can continue as far as the top operator of the QEP. The DOPs of the resulting block will then be considered up to the maximal value of 4 (given by the *par* value of the *Sel* operator).

From this set of degrees, TOPAZ will choose the best one. For the example QEP, this will probably be 2, as the cost model also has to take into consideration that there are two additional blocks (below edges *A* and *C*) that run in parallel, each of them having a degree of parallelism of 1.

5.7 Summary

In this chapter we have presented the cost model employed by the TOPAZ parallelizer. In contrast to most related work, this approach models intra-query parallelism employing data rivers in a comprehensive way, including blocking boundaries, resource usage as well as block building. Nevertheless, operator characteristics are encapsulated within the corresponding cost formulae. Hence, this generic approach is also suitable for object-relational extensions. New operators or user-defined functions merely have to provide their private cost functions that will be taken into consideration accordingly. The linear speedups presented in the previous chapters convincingly demonstrate the accuracy of the TOPAZ cost model.

Chapter 6

Enhancing Optimization for a Subsequent Parallelization: the Quasi-Parallel Cost Model

This section presents the strategies used in the optimizer component of MIDAS. Thereby, a special emphasis is put on the consideration of parallelism already in the sequential plan generation phase. As an effect, the subsequent parallelization yields more efficiently executable PQEPs.

6.1 Introduction

The aim of the TOPAZ parallelizer, presented in the previous chapters, is to achieve the best possible speedup by means of different forms of parallelism from a *given* sequential plan.

As mentioned in Section 4.2.4, for the first version of TOPAZ this sequential plan has been produced by the TransBase optimizer. This component is not cost-based. Instead, the initial (sub)plan is transformed step by step into a new (sub)plan by using heuristics. Since there is no possibility to compare the costs of the plans before and after a transformation, there is no guarantee on the fact that the final plan is also the best one.

In addition, it is still an open question how to obtain the overall best parallel plan. Prior work on *two-phase* parallelization [HS93, Has95] has shown that this cannot be achieved by simply parallelizing the best sequential plan. This is due to the fact that in a traditional two-phase parallelization scheme, the optimization, i.e. the generation of the sequential plan, is completely decoupled from the subsequent parallelization task. This often results in plans that are inherently sequential and, consequently, unable to exploit the available parallelism. On the other hand, using a detailed resource scheduling model during plan generation, as advocated by the *one-phase* approach, can have tremendous impact on optimizer complexity and optimization cost. For instance a dynamic programming algorithm, as employed by bottom-up optimizers must use much stricter pruning criteria that account for the use of system resources. This leads to a combinatorial explosion in the state that must be maintained while building the plan, rendering the algorithm impractical even for small query sizes [GI97].

An approach towards integrating some knowledge on parallelism already into the optimization is made in [Has95]. Here, the optimization is split up into a join ordering module and a post-pass optimization module. The first phase, i.e. optimization, is then performed as follows. For each processing tree generated by the join ordering module, the post-pass determines the edges for repartitioning, s.t. the overall communication costs for the given tree are minimized. The derived costs are then returned to the join ordering module, that continues to search for a better ordering. Even though this approach is limited to join orderings, the integration of a complex post-pass to the optimization problem augments its high computational complexity. It is not clear if such supplement efforts are compensated by significantly better execution plans. Moreover, since the join ordering is still a stand-alone module, the approach bears also the common drawbacks of 2-phase parallelization mentioned in Section 4.2.1, namely limited extensibility, difficult reuse of technology improvements etc.

In contrast, our approach is to provide an abstraction within the optimizer that lies between the two extreme ends of a spectrum, incorporating either detailed knowledge, as in the one-phase approach, or no knowledge (cf. the two-phase approach) of the parallel environment. Thus, the incorporation of this knowledge should be effective enough to influence the sequential plan generation, but at the same time not too detailed in order to keep the optimization task tractable. We decided to realize this goal by means of a new cost model, which we called *quasi-parallel.* Thus, the search engine of the optimizer employing this cost model concentrates only on promising regions in the (sequential) search space w.r.t. the subsequent parallelization task. In this way, the optimization can be regarded as an additional phase in the overall *multi-phase parallelization scheme* of the input query, corresponding to our strategy to conquer the complexity of the parallel search space.

However, in some situations queries are executed in non-parallel environments as well. Hence, for these scenarios it is still necessary to be able to derive also the best sequential plan. This is realized by means of a traditional cost model to be used by the optimizer that we called *sequential* (or short *seq*).

Given the above, the main goals of the new optimizer component to be integrated into the MIDAS system, have been the following:

1. improve the quality of the generated sequential plans by replacing the heuristic TransBase optimizer by a cost-based one.
2. make the optimization process cognizant of the subsequent parallelization as well. Hereby, the optimization overhead should be minimal. In contrast to [Has95], in our approach the plan generation itself should be influenced. This should be reflected among others by the shape of the generated execution plan, as well as the employed physical operators.
3. use the same search engine for both optimization and parallelization in order to achieve an integrated environment.

The corresponding concepts will be presented as follows. In Section 6.2 we present the main design decisions for the new optimizer component. The cost model for the sequential case will be detailed in Section 6.3. Our approach towards accounting for parallelism in the optimization phase, namely the *quasi-parallel* cost model, will be presented in Section 6.4. Furthermore, a

performance analysis in Section 6.5 and a summary in Section 6.6 concludes the chapter.

6.2 The Model-M Optimizer

Since for the implementation of TOPAZ we employed Cascades, we decided to use the same framework for the realization of the optimizer as well. In this way optimizer and parallelizer use the same (top-down) search strategy, i.e. Cascades, but explore different search space regions through different models.

The model used for the optimization phase is called *Model-M* (*M* for MIDAS). As presented in Section 4.3, the role of the model is to introduce logical and physical operators, compute logical and physical properties and to present rules that can be used to generate the search space. The cost model will be presented separately in the following sections.

The *logical operators*, listed in Appendix B.3, were defined specifically for the subset of SQL used in DSS queries [Bi97, Hi98, KB98]. The *physical operators* are given in Appendix B.4 and correspond to the operators used in the MIDAS execution engine (see Appendix B.1).

As presented in Section 4.3.1.3, *rules* are divided into transformation rules and implementation rules. The rules corresponding to *Model-M* are presented in Appendix B.5. In order to minimize the number of rule application and thus reduce search complexity, we decided to keep some passes of the TransBase optimizer as a pre-optimization step. For instance, assuming that it is almost always beneficial to push down selections and projections, the pre-optimization pushes the SELECT and PROJECT operators in the initial query down as far as possible. That is, in the initial query plan that is the input to the *Model-M* optimizer, these operators lie directly over the *GET* logical operator, if possible. As a result, SELECT and PROJECT push-down rules are not required in *Model-M*.

A general problem of the rule-based optimizers is the generation of *duplicates*. This is the repeated generation of the same expression by different sequences of transformation rules. Thus, simple transformational rules such as join associativity and join commutativity are complete in that they will generate every possible join order. However, they generate many expressions more than once. This is very costly since there are already order 3^n expressions in the memo for an n relation join without any duplicates [Bi97]. In addition, the application of rules is a relatively expensive process for the optimizer. A solution to this problem is presented in [PGK97]. Here, a *duplicate-free rule set (DFRS)* for generating all possible join orders is introduced. These rules generate all of the expressions in a logical search space (for example the join order search space) without generating any expression twice. We have implemented the duplicate free rule set as a general method to reduce search complexity in the *Model-M* optimizer as well. The corresponding rules bear the DFRS prefix in Appendix B.5.

As mentioned in the previous section, one of the goals of the new optimizer is to take into account the subsequent parallelization. Since the search engine, used also by the parallelizer TOPAZ, had to be left unchanged, the best possibility to achieve this goal is to influence the

model part of the optimizer framework, more precisely the cost model. Therefore, as already mentioned in the introduction, we have elaborated two different cost models: one for the sequential optimization and one for the subsequent parallelization task. These will be presented in the following sections. Thereby, we use the same notation and terminology as in Chapter 5.

6.3 The Sequential Cost Model

The goal of the *seq* cost model is to assess the best plan for a sequential execution, i.e. for a non-parallel processing environment. In this case, the resulting plan does not contain any parallel constructs. Hence, similar to the scenario presented in Section 5.3.1, the total costs of a (sub)plan are calculated by adding up the local costs of the root operator and the total costs of its inputs. Thereby, similar to the approach used by the TOPAZ cost model, the formulae for the calculation of the local costs are provided by the operator's specification and taken into account accordingly. This results in an abstraction that supports upcoming extended database functionality. However, from the cost components introduced in Section 5.4, only $T_{local}(N)$ and $T_{total}(N)$ are necessary for the *seq* cost model.

In this simplified model, the following formulae are used for the cost calculation of operators:

- N = a regular operator having arity n:
  $T_{total}(N) = T_{local}(N) + (\sum_{i=0}^{n-1} T_{total}(N_i))$.

- N = binary operator with multiple evaluation of the right input:
  $T_{total}(N) = T_{local}(N) + T_{total}(N_0) + card0 * T_{total}(N_1)$.

An exception from the above is introduced for constructs that involve a correlation. A typical representative of this class is the *index nested-loops (INL) join* operator. Consider the logical QEP from Fig. 6.1a. This can be transformed into a physical QEP using e.g. a HASH_JOIN as in Fig. 6.1b. In this case, the logical GET operator has also been transformed into a full table scan (REL_SCAN). Another alternative is given by the QEP in Fig. 6.1c, where the EQJOIN has been transformed into an *index nested-loops join*. In this case, for each tuple *T* of the left

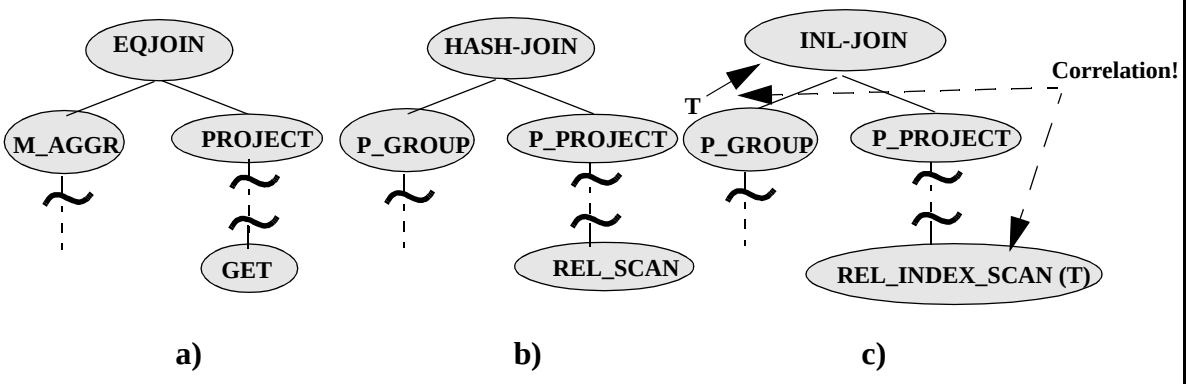


Fig 6.1: Alternative query execution plans

input, an index scan is performed on the right input taking into consideration only the value of the join attribute of tuple *T*. Here, we have to deal with a *correlation*, i.e. the processing of the right input is dependent of the current value of *T*.

What are the possibilities to model such a correlation situation in an optimizer framework like Cascades, that relies on the independency assumption (see Chapter 4)? Similar problems arise when dealing with queries containing e.g. *exist* or *(not) in* clauses. Calculating the costs of the right input is especially difficult. As already mentioned, the cost calculation is performed bottom-up, taking into account the input cardinalities. But in a correlation scenario these cardinalities are variable. For instance in Fig. 6.1c, the number of tuples delivered by the REL_INDEX_SCAN operator varies for each access. Hence, we decided to encapsulate the information referring to the correlation entirely within the INL_JOIN operator. Thus, both inputs can be optimized in the usual manner. The output cardinality of the REL_INDEX_SCAN operator is the same as for an usual full table scan operator. However, when the cost calculation reaches the INL_JOIN operator, the following formula is used:

$$T_{total}(N) = T_{local}(N) + T_{total}(N_0) + card0 * T_{total}(N_1) * index\_factor.$$

The component *index_factor* indicates the fraction of work that has to be done in each iteration for the evaluation of the right input as compared to a cartesian product. It is calculated with the help of the logical property *unique_card*. This property reflects the number of distinct values for each column in the schema.

The following example illustrates the calculation of the *index_factor.*

Example 5.3: Consider that the right input in Fig. 6.1c has a *unique_card* of 20 for the join attribute and the cardinality of this input is 2000. Then, assuming a uniform value distribution, the number of tuples to be transmitted to the INL_JOIN operator in each iteration averages to 2000/20 = 100 tuples.

Hence the *index_factor* can be calculated as the reverse of the *unique_card* of the right join attribute. Please note that the cardinality of the right input is already considered in the above formula by the component $T_{total}(N_1)$.

6.4 The Quasi-Parallel Cost Model

By employing the *quasi-parallel* cost model, we aim to introduce knowledge on the subsequent parallelization task into the sequential plan generation as well. Thereby, possible parallel constructs should be favored and eventual deterrents of parallelism should be eliminated. At the same time, the optimization, respectively parallelization complexity should not increase significantly. In this way, the optimization can be regarded as a (first) phase of the overall parallelization scheme. Similar to the different phases of TOPAZ, the *Model-M* optimizer employing the *quasi-parallel* cost model leads the optimization process towards promising regions of the search space.

In the following, we concentrate on different aspects of optimization that mostly influence the

subsequent parallelization and hence should be considered by the *quasi-parallel* cost model.

6.4.1 Blocking operators

As presented already in Chapter 5, in a parallel processing environment, the critical path and thereby also the response time of a query execution plan is mostly influenced by blocking operators. Fig. 6.2 shows two alternative (sub)plans of a query. The local costs are given as numbers near the corresponding operators. The first plan (Fig. 6.2a) contains a blocking operator. In the *seq* cost model, the total cost of this plan is**:**

T_{total} *(seq) = 30 + 30 = 60.*

If we now consider only pipeline parallelism, the total cost of this plan is also
T_{total} *(par) = 30 + 30 = 60,*
as the blocking boundary prohibits parallelism between the two operators and hence dependent parallel execution (DPE) degenerates to sequential execution (SE).

The second plan alternative, shown in Fig. 6.2b, does not contain any blocking boundaries. However, the constituting operators are more costly than in the first plan. In the sequential execution scenario, for the total costs of this plan we have:

T_{total} *(seq) = 40 + 40 = 80.*

Clearly, these costs are higher than in the first variant, hence a traditional optimizer would choose variant a). However, if pipelining parallelism is employed and assuming that enough resources are available, the total costs of this plan are:

T_{total} *(quasi-par) = 40 || 40 = max (40, 40) = 40.*

Hence, for a parallel environment employing only pipelining parallelism, the second alternative is more favorable.

In order to assess the influence of blocking operators on the total response time, we decided to incorporate also the components T_{begin} and $T_{process}$ into the cost calculation. As defined in Section 5.3.2, $T_{begin}(N)$ indicates the time when operator N delivers its first (intermediate) result tuple to its predecessor and thus accounts for the costs of its materialized front. The component $T_{process}$ is needed to add up the local costs of the constituting operators between two blocking boundaries.

Thus, analog to Section 5.3.4, we have:

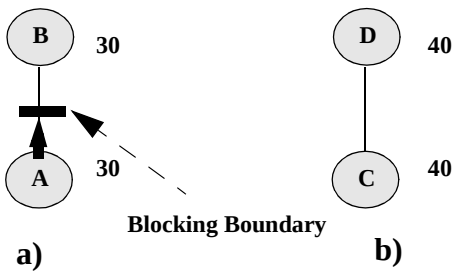


Fig 6.2: Influence of blocking operators

1. N is leaf operator:
   $T_{process}(N) = T_{local}(N)$
2. else: (the arity of $N = n$)
   $T_{process}(N) = (\sum_{i=0}^{n-1} T_{process}(N_i)) + T_{local}(N)$.
3. N is blocking operator:
   $T_{begin}(N) = T_{total}(N)$
   $T_{process}(N) = 0$.

6.4.2 Favoring Independent Parallelism

The shape of the query execution plan is also of interest for a subsequent parallelization. For instance, in a left- or right-deep tree only dependent parallel execution (DPE) is possible. As shown in Section 5.3.3, DPE can deteriorate to SE in the presence of resource contention or if blocking operators are involved. Hence, an additional goal of the *quasi-parallel* cost model is to guide the search space exploration towards regions with high potential for independent parallel execution (IPE). Hereby, operators having more than one input bear a special interest. This is due to the fact that the different input subplans can run parallely in a real IPE manner if enough resources are available (cf. Section 5.3.2).

On the other hand, operators having more than one input are usually costly. Hence, in the *quasi-parallel* cost model we assume these operators will be detached from their inputs by TOPAZ. This is reflected by the following formulae given for binary operators.

- N = a regular binary operator:
  $T_{begin}(N) = MAX\ (T_{begin}(N_0), T_{begin}(N_1))$
  $T_{process}(N) = MAX\ (T_{local}(N), T_{process}(N_0), T_{process}(N_1))$
  $T_{total}(N) = T_{process}(N) + T_{begin}(N)$.
- N = binary operator with multiple evaluation of the right input:
  $T_{begin}(N) = MAX\ (T_{begin}(N_0), T_{begin}(N_1))$
  $T_{process}(N) = MAX\ (T_{local}(N), T_{process}(N_0), card0 * T_{total}(N_1))$
  $T_{total}(N) = T_{process}(N) + T_{begin}(N)$.
- N = *INL-Join*:
  $T_{begin}(N) = T_{begin}(N_0) + T_{begin}(N_1)$
  $T_{process}(N) = MAX\ (T_{local}(N), T_{process}(N_0), card0 * T_{total}(N_1) * index\_factor)$
  $T_{total}(N) = T_{process}(N) + T_{begin}(N)$.
  Here, the materialized fronts of the two inputs are added up as an effect of the correlation (see Section 6.3). The right input can only start working when the left input has delivered its first tuple.
- N = *Hash Join*:
  $T_{begin}(N) = MAX\ (T_{total}(N_0), T_{begin}(N_1))$
  $T_{process}(N) = MAX\ (T_{local}(N), T_{process}(N_1))$
  $T_{total}(N) = T_{process}(N) + T_{begin}(N)$

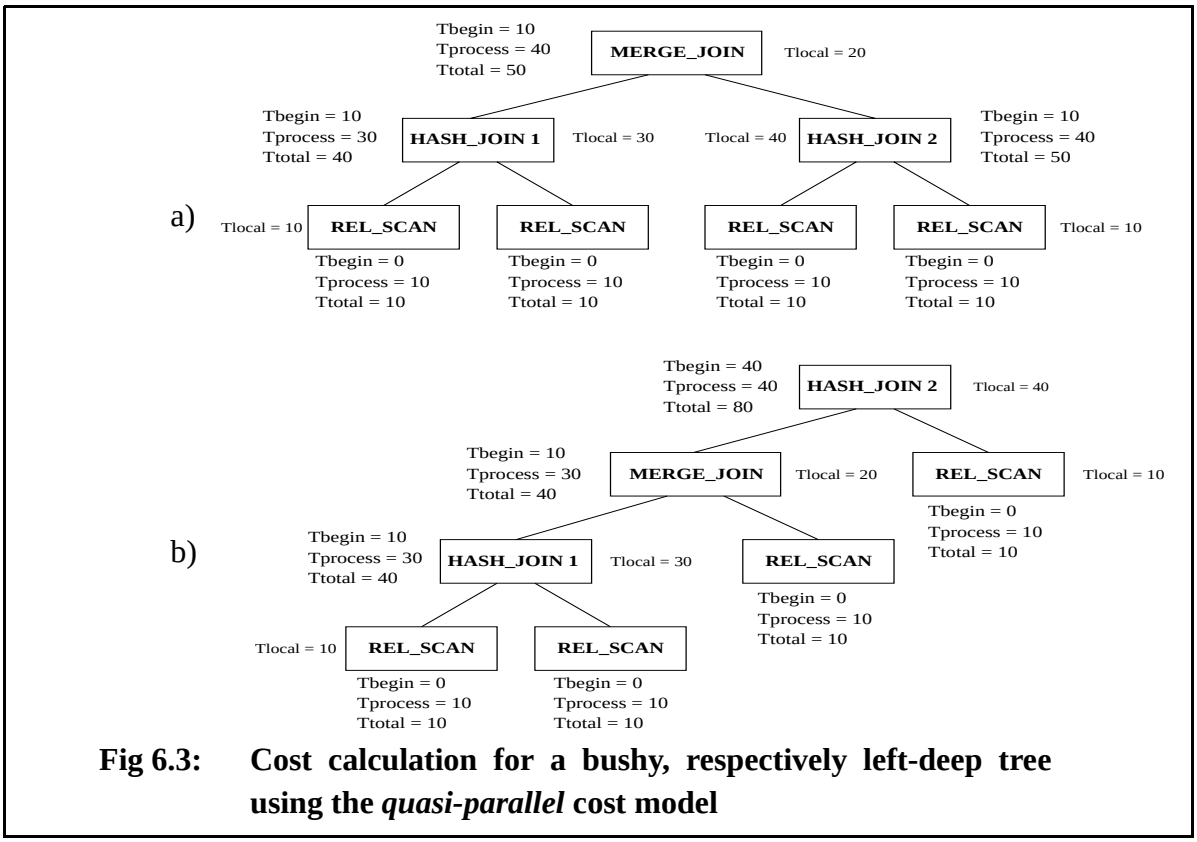


Fig 6.3: Cost calculation for a bushy, respectively left-deep tree using the *quasi-parallel* cost model

In this case, we assumed that N_0 is the build relation and N_1 is the probe relation. Hence, the evaluation of the left input has to be completed in order to build the hash table. Only at this moment can the first result tuple be delivered to the predecessor.

With these cost formulae, plans with high potential for IPE are favored. This is illustrated also by the example given in Fig. 6.3. Here, we made the simplifying assumption that the local costs of the operators remain the same for different join ordering strategies. The cost calculation is performed for a bushy (Fig. 6.3a), respectively left-deep (Fig. 6.3b) tree. It starts with the leaf operators and uses the derived costs for the calculation of the predecessors, cf. the above given formulae for hash-, respectively merge-joins (the latter being considered as regular binary operators). The total cost of the trees corresponds to the total cost of the top operators. As results from Fig. 6.3, the *quasi-parallel* cost model yields for the bushy tree a total cost of 50, while in the case of the left-deep tree this cost is 80. Hence, the first alternative will be chosen.

Contrary to binary operators, parallelism between unary operators and their successors is not incorporated into the cost model. The motivation for this decision lies in the fact that splitting up a pipeline into multiple blocks can only be done on the basis of global considerations as shown in Section 5.6. This is the task of the subsequent phases of the TOPAZ parallelizer. On the other hand, without blocking operators the best sequential (sub)plan for a pipe usually corresponds also to the best parallel subplan.

Nevertheless, if blocking operators are involved, the *quasi-parallel* cost model will influence the physical implementation of unary operators along a pipe as well. An example is given in Fig. 6.4. Here, the M_UNIQ logical operator from Fig. 6.4a is first transformed into the physical duplicate elimination P_UNIQ, that requires a sorted input (Fig. 6.4b). The second alterna-

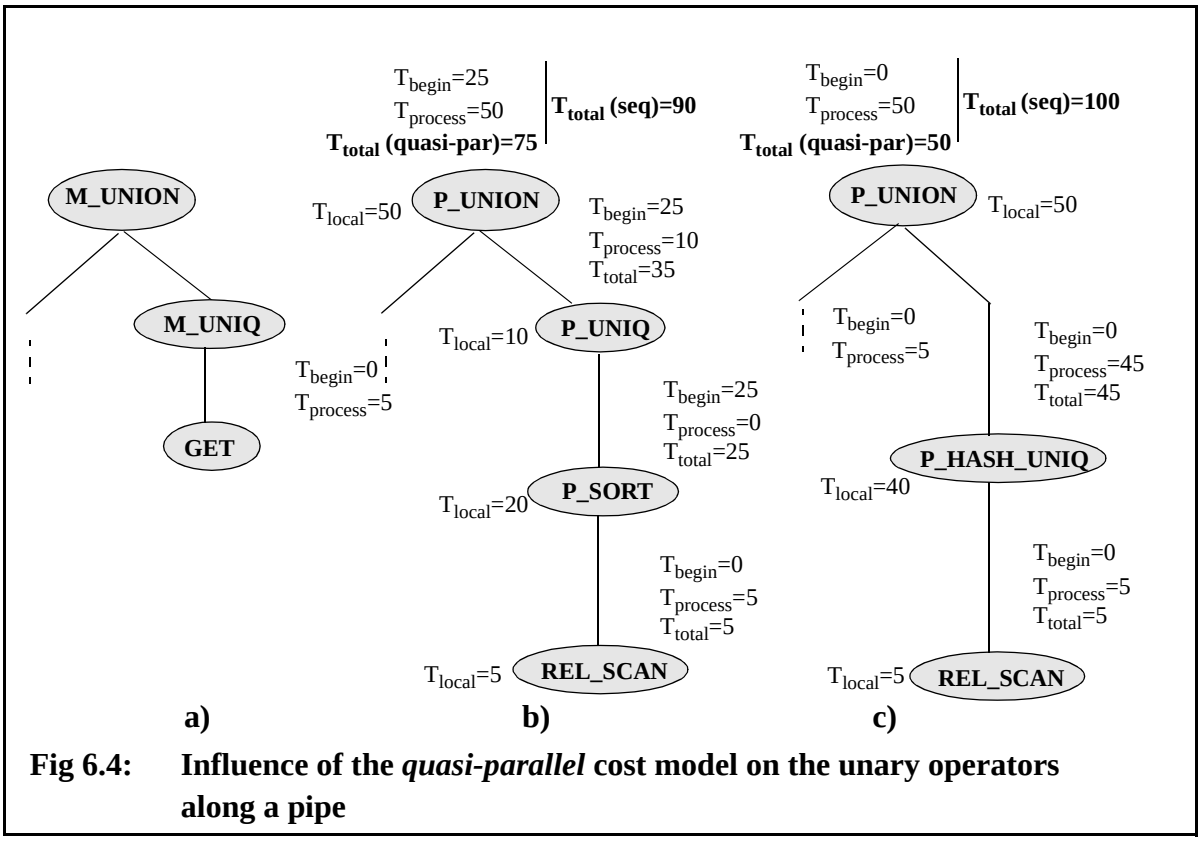


Fig 6.4: Influence of the *quasi-parallel* cost model on the unary operators along a pipe

tive, shown in Fig. 6.4c, is the hash-based P_HASH_UNIQ operator, that has higher local costs but does not require a sorted input. The total costs of the plans for the *seq* cost model are *90*, respectively *100*, rendering the first alternative as the best plan.

However, in this QEP the critical path is mostly influenced by the binary P_UNION operator. Hence, in a parallel environment it is not necessary to choose the best possible physical implementation for the duplicate elimination M_UNIQ (in this case the P_SORT/P_UNIQ combination), since this will run anyway in parallel with the costly P_UNION operator. On the contrary, if the best sequential alternative contains a blocking operator, as e.g. the *sort* operator in Fig. 6.4b, this is even counterproductive, since it causes a delay until the first input tuple reaches the P_UNION (reflected by the T_{begin} cost component of this operator that is different from null). This example enforces once more that parallelizing the best sequential physical plan does not yield the best parallel plan.

By employing the *quasi-parallel* cost model as shown in Fig. 6.4, the total costs of the first alternative *(75)* are higher than that of the second *(50)*. Hence, this second plan will be the one to be transmitted to the parallelizer.

6.4.3 Degrees of Parallelism

In the last section, we have shown that the consideration of blocking edges in the *quasi-parallel* cost model influences the physical operators that constitute the final plan produced by the optimizer. However, in some cases the suitability of some operators for a parallel scenario becomes only evident if different DOPs are taken into account. Hence, analogously to the approach

described in Section 4.4.2, we decided to use cost arrays corresponding to each possible degree of parallelism already for the optimization phase. However, please note that the goal is to produce a sequential physical plan with a high affinity for intra-query parallelism. The decision on where to introduce the different forms of parallelism is still made by the TOPAZ parallelizer.

In order to assess the costs corresponding to different DOPs, it is important to consider the parallelization possibilities of operators as well. As mentioned in Section 4.5.2, some operators can be parallelized in different ways. Thus, e.g. the *hash join* can be parallelized by partitioning both inputs or by partitioning only one input and replicating the other. In some cases, as e.g. for some user-defined functions or user-defined table operators [Ja99], the lack of suitable partitioning functions imposes the replication of one or several inputs. The same argumentation applies also for subplans containing a correlation.

For the cost calculation corresponding to a given DOP it is important to know whether the operator (respectively subplan) is partitioned or replicated. We solved this problem by introducing two additional physical properties into the *quasi-parallel* cost model:

- ***Part*:** in this case the operator or subplan is partitioned by means of a suitable partitioning function
- ***Repl*:** this physical property stands for a replicated operator or subplan

Please note that this strategy is used for a more adequate cost estimation in the presence of a possible intra-operator parallelism. As already mentioned, the goal is to explore search space regions where this form of parallelism can be efficiently used without unnecessarily blowing up the search complexity. Hence, any concrete partitioning functions and parallelization particularities are not taken into account. This task and the corresponding accurate cost calculation will be effectuated by the different phases of the parallelizer.

With the help of cost arrays for different DOPs and assuming that each instance of an operator or subplan can work independently of all the other instances, the consideration of intra-operator parallelism can be done without any changes to the cost formulae introduced in the previous sections. This aspect will only be reflected by the local operator costs. As defined in Section 5.3.8, $T_{local}(N, card)$ stands for the local costs of operator N when the input cardinality is *card*. Thus, assuming a uniform data distribution, the costs of operator N having m instances is calculated as follows:

- $T_{local}(N) = T_{local}(N, card/m)$, for the *Part* context
- $T_{local}(N) = T_{local}(N, card)$, if the optimization is made in the *Repl* context.

The optimization starts with the top operator using the physical property *Part*. In the course of optimization, the parallelization possibilities of each operator are taken into account, continuing the optimization of the inputs with the corresponding physical property. Thereby, the costs are calculated for each possible degree of parallelism. When the cost calculation is finished, the best plan is chosen according to the lowest values in the cost arrays of the alternative (sub)plans.

This strategy is exemplified by the cost calculation of the logical plan given in Fig. 6.1a. A portion of the corresponding memo structure is depicted in Fig. 6.5. Here, for each physical operator that is contained in a group (see Section 4.3.1.2) the appropriate cost array is calculated by

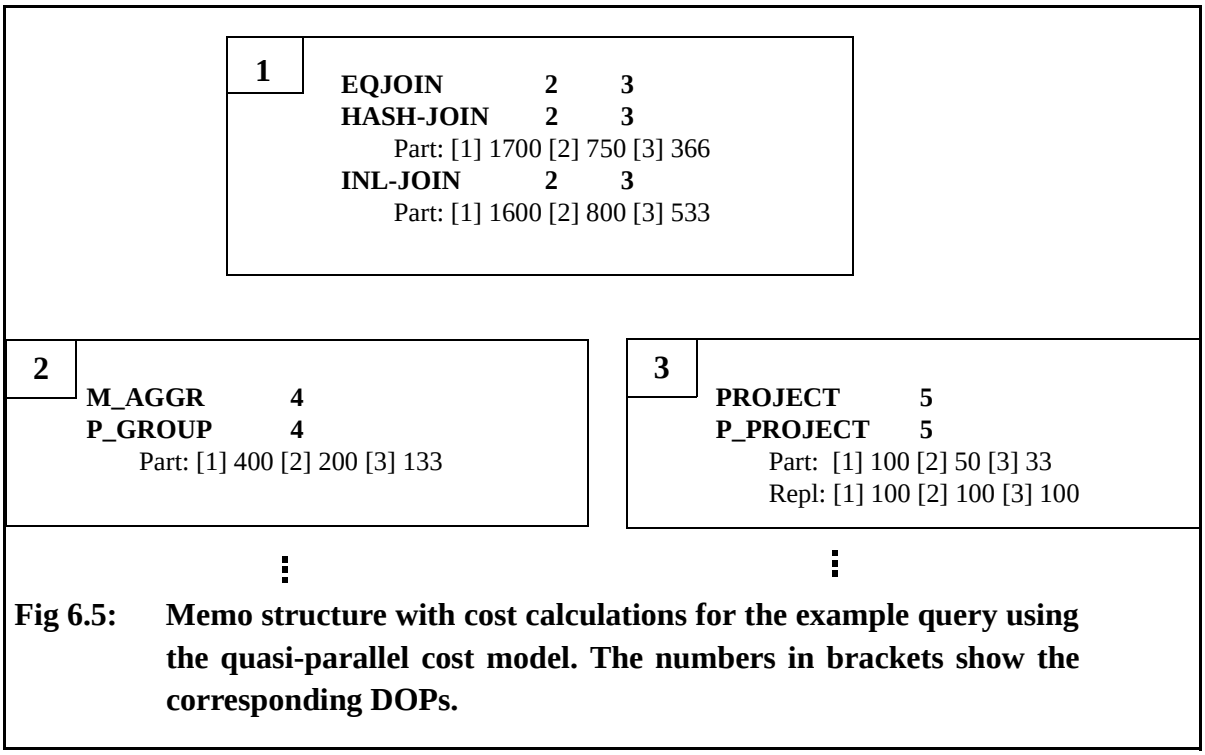


Fig 6.5: Memo structure with cost calculations for the example query using the quasi-parallel cost model. The numbers in brackets show the corresponding DOPs.

taking into consideration the current context. Thereby, we assume that the database contains 3 processing sites, hence a DOP of up to 3 is taken into consideration.

The optimization starts with the top operator contained in the first group, i.e. the EQJOIN, by transforming it into a HASH_JOIN (cf. Fig. 6.1b). This operator can be most efficiently parallelized by partitioning both of its inputs. Hence, the optimization of the inputs continues with the physical property *Part*. The top operators of the resulting physical subplans are contained in group *2* and *3* together with the corresponding cost arrays. For the final cost calculation, it is important to assess the local costs of the HASH-JOIN operator. Assume that the input cardinality of this operator is *card*. We further assume that for this cardinality, the hash table cannot be kept completely in main memory, i.e. I/O operations are necessary. For a DOP=2, the available memory size is also doubled, thus less disk spoolings are necessary. For a DOP=3 the hash tables can be maintained entirely in main memory, thus don't causing any I/O costs. Hence, for the different degrees of parallelism, we assume to have the following costs:

T_{local}*(HASH-JOIN, card/1)* = 1200
T_{local}*(HASH-JOIN, card/2)* = 500
T_{local}*(HASH-JOIN, card/3)* = 200.

For the final cost array, the local costs of the top operator are added to the total costs of its inputs corresponding to each DOP[1]:

- N = HASH-JOIN:
  [1] T_{total}*(N)* = 1200 + 400 (left input) + 100 (right input) = 1700
  [2] T_{total}*(N)* = 500 + 200 (left input) + 50 (right input for the context *Part*) = 750
  [3] T_{total}*(N)* = 200 + 133 (left input) + 33 (right input for the context *Part*) = 366.

1. For simplification purposes, we have not mixed in this example the strategy for favoring IPE with the strategy for taking into consideration the DOPs. Hence the costs are calculated by addition rather than using the maximum as in Section 6.4.2.

For the second physical alternative an *index-nested loops join* is employed (cf. Fig. 6.1c). Since this operator implies a correlation, as explained in Section 6.3, it only can be parallelized by replicating its right input. Thus, the optimization of this input is made in the *Repl* context, yielding the costs shown in the second line of group *3*. On the other hand, assume that the local costs of the INL-JOIN operator are 600 for the sequential case. Since this operator cannot make profit of increasing memory sizes, the local costs of an instance decreases linearly with the degree of parallelism:

$T_{local}(INL\text{-}JOIN, card/1) = 600$
$T_{local}(INL\text{-}JOIN, card/2) = 300$
$T_{local}(INL\text{-}JOIN, card/3) = 200.$

For the final cost array, assuming the left input cardinality *card0* as being 6000 and the *unique_card* of the right input as being 1000, we have:

- N = INL-JOIN:

  [1] $T_{total}(N) = T_{local}(N) + T_{total}(N_0) + card0 * index\_factor * T_{total}(N_1) =$
  $= T_{local}(N) + T_{total}(N_0) + card0 / unique\_card * T_{total}(N_1) =$
  $= 600 + 400 + 6000/1000 * 100 =$
  $= 600 + 400 + 6 * 100 = \underline{1600}$

  Analogously:

  [2] $T_{total}(N) = 300 + 200 + 3 * 100 = 800$

  [3] $T_{total}(N) = 200 + 133 + 2 * 100 = \underline{\underline{533}}.$

By comparing the resulting cost arrays, we observe that the variant containing the INL-JOIN operator is more favorable for the sequential case, i.e. DOP=1. For a traditional cost model, this is the final plan that is produced by the optimizer. However, if intra-operator parallelism is applied, we have significantly lower response times for the plan variant containing the HASH_JOIN, especially for a DOP = 3. In a traditional two-phase parallelization, this plan would have been missed. By applying the *quasi-parallel* cost model in the optimization phase, the resulting plan is chosen according to the lowest value in the cost arrays. In this case, this is the variant containing the *hash join*, for the DOP=3. Hence, this is the plan that will be handed to the parallelizer.

6.5 Performance Measurements

Our goal in this section is to determine the effectiveness of our the *Model-M* optimizer as well as of the *quasi-parallel* cost model. As mentioned at the beginning of this section, the first goal was to improve the quality of the sequential plans as compared to the plans produced by the *TransBase* optimizer. In order to verify if this goal has been reached, we have used selected queries from the from the TPC-D benchmark.[1] These queries have been optimized, parallelized and processed on the same workstation cluster as presented in Section 3.3.5. Please note that we

1. Since the *Model-M* optimizer cannot yet resolve subqueries, we omitted corresponding queries from our performance measurements.

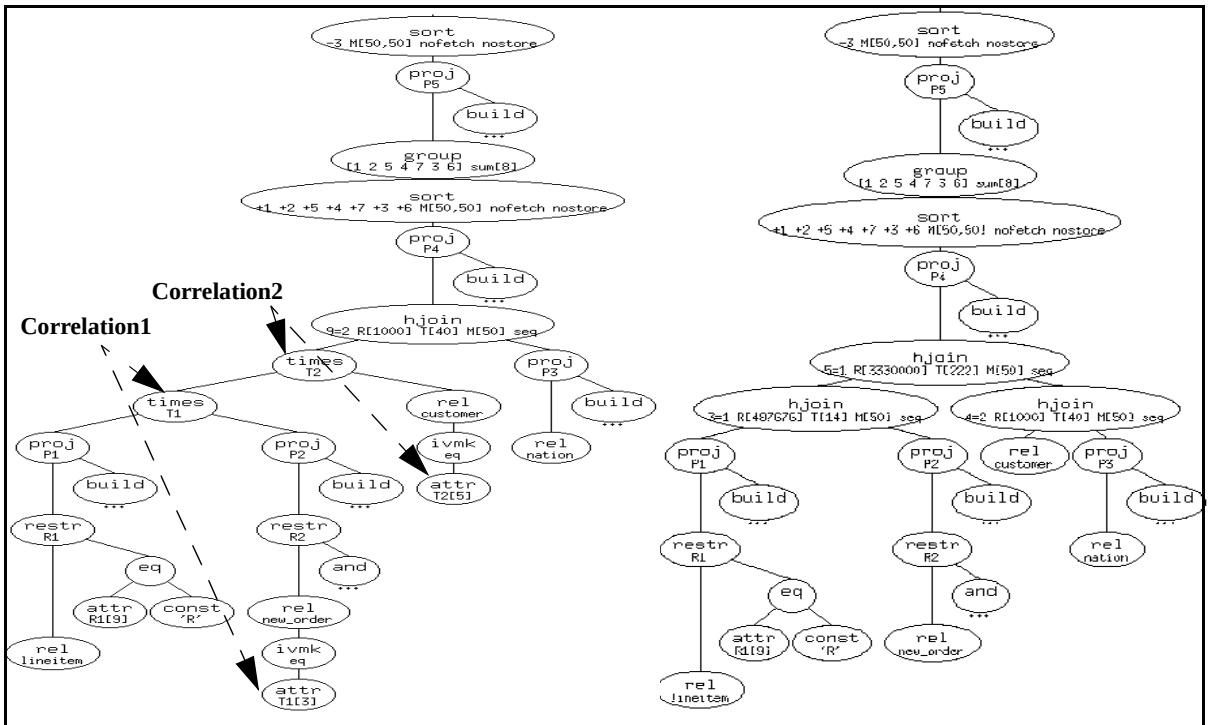
Table 6.1 Average execution times (sec)

	TransBase	Model-M
Sequential	286,13	130,95
Parallel	106,3	49

have used the *seq* cost model for the Model-M optimizer. Thus, both optimizers produce the best sequential plans, i.e. for non-parallel processing scenarios, corresponding to their search strategy.

As shown in Table 6.1 by comparing the execution times, the sequential plans produced by the new optimizer are significantly better that those produced by the *TransBase* optimizer. This applies also for the parallel plans that have been generated by TOPAZ from the respective sequential plans.

Next, we analyze the impact of the *quasi-parallel* cost model on execution times as well as on optimization and parallelization complexity. First, we illustrate this by using a single example query (Q10 from the TPC-D benchmark). Fig. 6.6 shows the sequential plans obtained by optimizing this query first using the *seq* cost model and next the *quasi-parallel* cost model. By comparing Fig. 6.6a and Fig. 6.6b we first observe that the shapes of the resulting plans are different. In the first case a *left-deep* tree has been produced, while in the second case the shape of the plan is *bushy.* As mentioned in Section 6.4.2, this maximizes the effect of independent parallelism. Next, we notice that the physical operators employed are also different. While for the implementation of two joins the *seq* cost model has chosen *index nested-loops joins*[1], implying two



a) *Seq* cost model a) *Quasi-parallel* cost model

Fig 6.6: Sequential plans obtained by optimizing query Q10 using different cost models

Table 6.2 Analysis of query Q10

	Optimization				Parallelization				
Cost Model	**Tasks/ Rules/ Expr**	**Opt. Time (ms)**	**Mem. (KB)**	**Exec. Time Seq. Plan**	**Phases**	**Task/Rules/ Expr**	**Parallel-ization Time**	**Mem. (KB)**	**Exec. Time Parallel Plan**
Seq	945/ 225/ 174	595	324	48,1 s	1	344/39/109	1,127 s	508	42,1 s
					2	305/41/91			
					3	376/57/101			
					4	275/24/87			
					Sum	1300/161/388			
Quasi-Par	989/ 234/ 183	640	356	49 s	1	385/50/123	1,343 s	620	12,8 s
					2	346/54/109			
					3	645/132/161			
					4	291/29/93			
					Sum	1667/265/486			

correlations, the *quasi-parallel* cost model has decided to use for the same operations *hash-joins*. This results from the approach described in Section 6.4.3. Hence, this strategy eliminates the correlations in Fig. 6.6a since for this example and according to the *quasi-parallel* cost calculation they are counterproductive to parallelism.

In order to perform a detailed analysis, in Table 6.2 we have listed different measures related to the optimization, parallelization, as well as execution time of the example query. Thereby, the optimization (respectively parallelization) complexity is expressed in the number of tasks, applied rules and generated expressions similarly to the performance investigation for TOPAZ in Section 4.7. In addition, we have also listed the absolute optimization (parallelization) times and corresponding memory consumptions. For the parallelization the number of tasks, rules and

Table 6.3 Measurement results for applicable TPC-D queries (averages)

Speedup *seq*	2.01
Speedup *quasi-parallel*	3.98
Speedup *quasi-parallel* as compared to *seq*	2,22
Optimization Overhead in Tasks (%)	8,84
Optimization Overhead in Expressions (%)	9
Optimization Overhead in Rules (%)	7,9
Parallelization Overhead in Tasks (%)	8,5
Parallelization Overhead in Expressions (%)	8,3
Parallelization Overhead in Rules (%)	15,21

1. Please note that in the MIDAS engine, the *index_nested_loops* functionality is implemented using the *times* operator as well.

expressions are given separately for each phase as well. By comparing the two cost models, we observe that the execution time of the sequential plan obtained after the optimization phase using the *quasi-parallel* cost model (cf. Fig. 6.6b) is higher than that obtained by using the *seq* cost model (cf. Fig. 6.6a). This is as expected, since the *seq* cost model is in charge of producing the best plan for a non-parallel environment. However, the situation is different for the execution times of the parallelized plans, given in the last column of Table 6.2. Here, the PQEP produced by parallelizing the *quasi-parallel* plan shows a linear speedup, while in the case of the best sequential plan parallelism could not be applied efficiently. Please note that the difference between the execution times of the two parallel plans is more than factor 3. As can be seen by the measures given in Table 6.2, this has been achieved with a relatively low optimization and parallelization overhead by using the *quasi-parallel* cost model.

Similar results apply to the other TPC-D queries as well. In Table 6.3, we have listed the average speedups achieved by parallelizing the queries that have been optimized with the two different cost models. As can be seen in line 2 of Table 6.3, a sequential optimization using the *quasi-parallel* approach followed by a parallelization as proposed in the TOPAZ strategy yields in the average linear speedups. Line 3 shows that the parallel plans resulting from the *quasi-parallel* cost model are twice as fast as those obtained from the *seq* cost model. The corresponding optimization/parallelization overhead is around 10 percent.

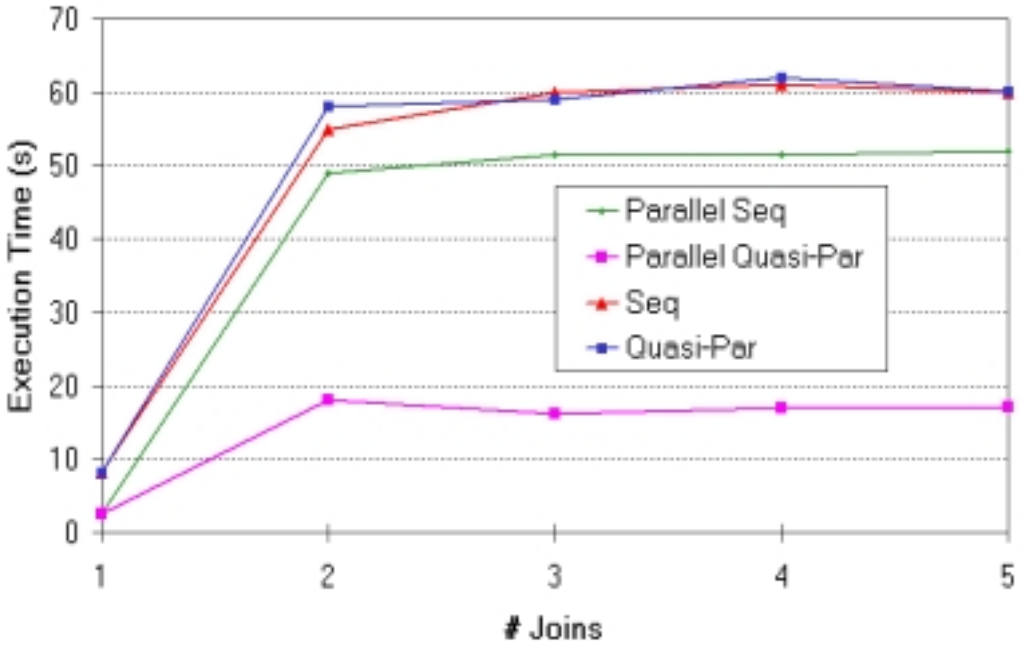
For the second test scenario we used join queries, where the number of joins has been successively increased from 1 to 5. The database has been partitioned on three disks. The measurement results are presented in Fig. 6.7.

By comparing the execution times in Fig. 6.7a we notice that the traditional 2-phase approach, i.e. parallelizing the best sequential plan (denoted *Seq* in the diagram), produces suboptimal results in this case as well. In contrast, by using the *quasi-parallel* cost model, we obtain sequential plans (denoted *Quasi-Par* in the diagram) that have similar or higher execution times as the best sequential plans, but that bear a higher potential for parallelization. The overall approach, i.e. optimizing the input plan with the *quasi-parallel* cost model and parallelizing the resulting plan with TOPAZ yields in this scenario also linear speedups.

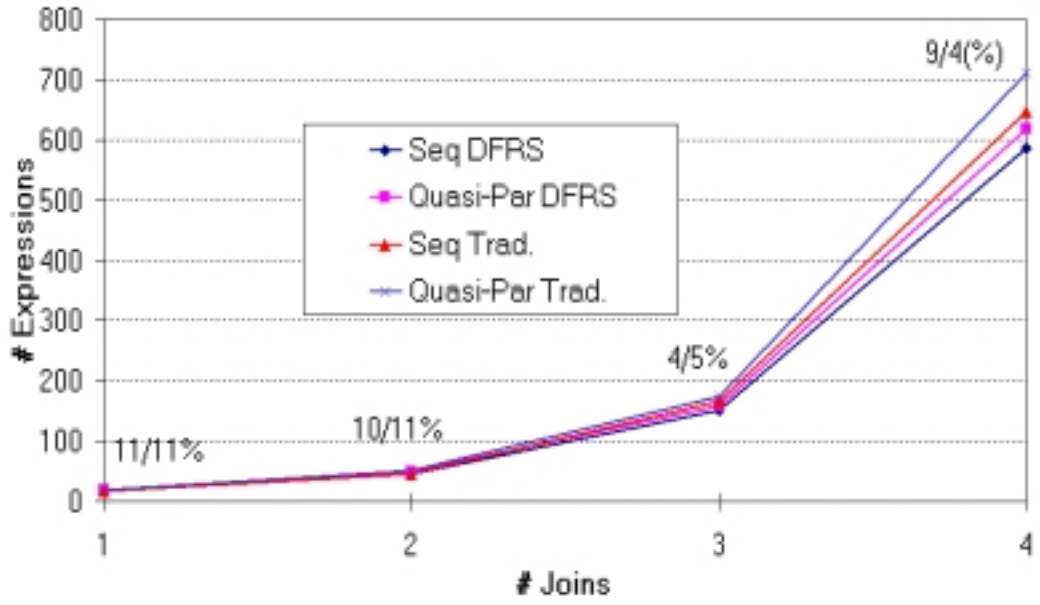
Fig. 6.7b shows the actual number of expressions generated during the optimization phase. Since this measure increases exponentially with the number of joins [OL90], for visibility purposes, we only presented the results for queries containing up to 4 joins. The results are presented for two cases: once for a “traditional” rule set and second for the duplicate-free rule set (marked DFRS in the figure) that has been explained in Section 6.1. The optimization overhead resulting from *quasi-parallel* cost model for these two scenarios are given in percent near the corresponding value coordinates. The results show that this overhead is in this test series only up to ca. 10-15% as well.

By comparing the DFRS and traditional scenarios, we have also shown that our approach can further fully benefit of any improvements made in the area of query optimization. Thus from Fig. 6.7b results that the usage of a duplicate-free rule set has reduced the number of considered expressions in the *quasi-parallel* optimization as well.

In some cases, the *seq* and *quasi-parallel* cost models produce the same plan. This is for instance the case for the query containing a single join in this test scenario. In this situation, the



a)



b)

Fig 6.7: Response times and optimization complexity

increase in optimization complexity (11% as shown in Fig. 6.7b) is not compensated by lower execution times (see Fig. 6.7a). However, this situation arises mostly for simple queries, where the optimization complexity is anyway very low. Hence, the overall performance losses are not noticeable.

In Fig. 6.8a we show the parallelization overhead in expressions generated as compared to parallelizing the optimal sequential plan. As expected, the parallelization complexity increases in almost all cases. The reason for this is that the *quasi-parallel* cost model produces plans with generally higher parallelization possibilities. The biggest difference is achieved in Phase 3 of the parallelization process. The explanation lies in the fact that due to the better parallelization

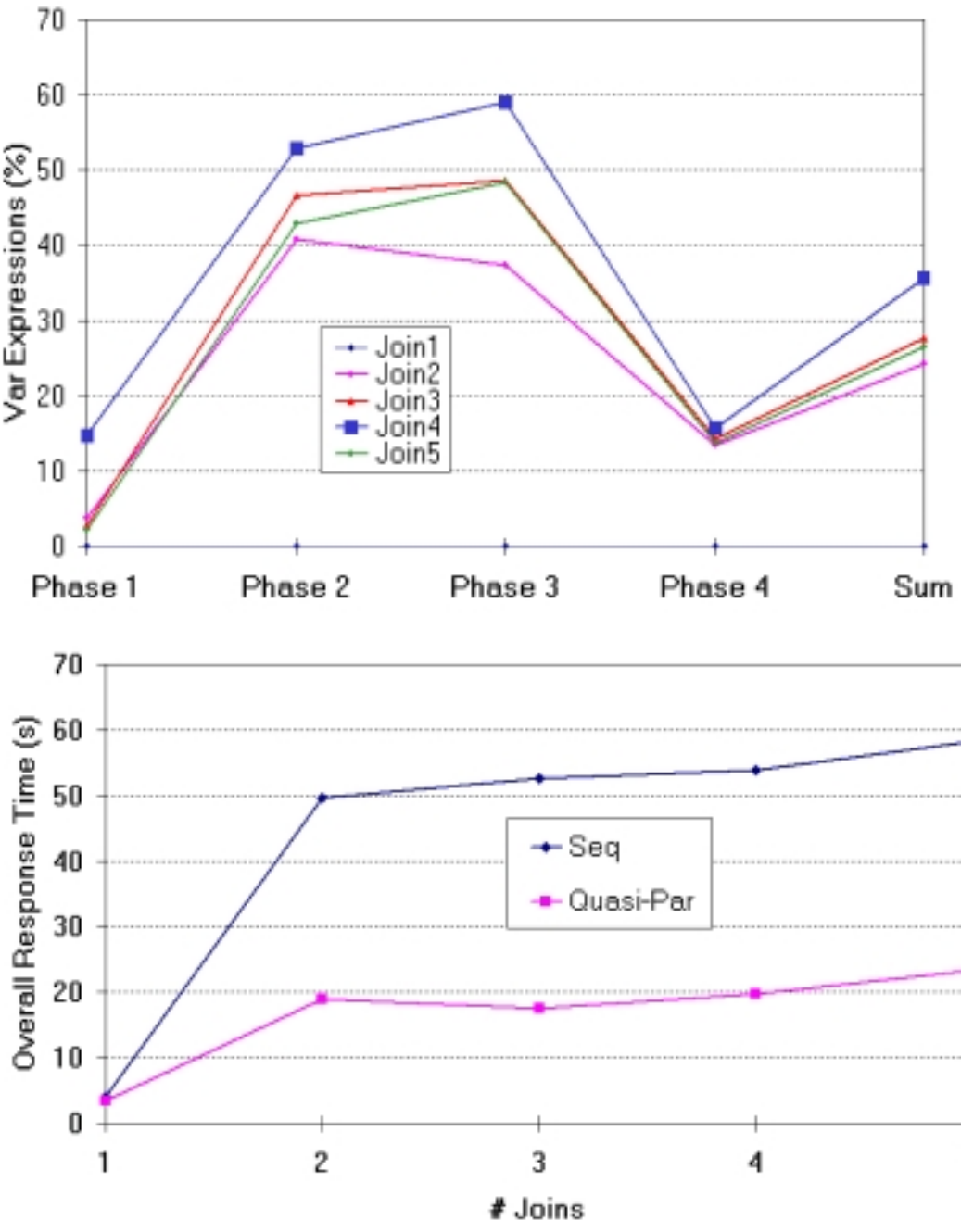


Fig 6.8: Parallelization complexity and overall response times

possibilities, the first two phases could detect more portions in the sequential plan where parallelism is worthwhile to be introduced. As described in Section 4.5.3, it is mainly the task of Phase 3 to carry over these forms of parallelism to the other parts of the query.

However, the increases in optimization, respectively parallelization complexity are fully compensated by the performance gains in execution times. This is shown in Fig. 6.8b, where we compared the overall response times, i.e. the sum of optimization, parallelization and execution times.

6.6 Summary

In this section, we have presented the *Model-M* optimizer as an approach to realize cost-based optimization. This component uses the same search engine as the TOPAZ parallelizer, namely Cascades. However, due to different models, they explore different portions of the search space. For the presented optimizer, we implemented a *quasi-parallel* cost model, in addition to the *sequential* one. In this case, contrary to other approaches, the optimization is grounded on the knowledge that every query will be executed on a parallel processing system. This is done without significant overhead in the optimization, respectively parallelization process.

Thus, the optimizer perfectly fits into the overall parallelization strategy, that reduces the search complexity through phases, each phase concentrating on different portions of the search space. As confirmed by measurement results, this overall parallelization strategy yields parallel plans showing linear speedups.

Chapter 7
Scheduling and Load Balancing

In this section we present the concepts used for the realization of the QEC component in MIDAS. QEC is a run-time system that combines the tasks of load balancing as well as execution scheduling and determines the final resource allocation for the PQEP execution.

7.1 Introduction

In the previous chapters we have presented the strategies that are used by the TOPAZ parallelizer, respectively the Model-M optimizer, to produce parametrized parallel query execution plans. At run-time this PQEP is examined by the *Query Execution Control(QEC)* component. Among the most important requirements posed to this component are the following ones:

- *Scalability*: Scalability is an issue of concern in building parallel database systems because of the ever increasing amount of data, and the complexity and volume of application interaction with the database system. Ensuring scalability of a system requires eliminating potential bottlenecks in the system. This can be only accomplished by choosing a distributed approach for the design of all significant system components. However, in the case of scheduling and load balancing, most related work is based on a centralized approach.
- *Resource management:* In order to provide optimal response times, resource utilization should be maximized as much as possible. Therefore, e.g. idle times due to waiting situations that are inherent to database query processing, should be reduced to a minimum possible.
- *Load balancing:* Load-balancing policy falls into two broad groups: static and dynamic. Static policies use algorithms which operate without regard to run-time loads across a system, while dynamic policies use the run-time performance of various parts of a system in order to make more ‘informed’ decisions about balancing. In order to improve the response time of a single (parallel) query, as well throughput in a multi-query environment, a dynamic load-balancing policy which uses run-time state information in making schedul-

ing decisions should be used.

- *Hybrid heterogeneous architectures:* A simplifying assumption in most research on resource management ([LOT94], [GGS96], [TL96], [DG92], [Va93]) is to consider standard architectures, like shared-everything (SE), shared-disk (SD) or shared-nothing (SN), with identical nodes. However, new trends show a convergence of parallel hardware towards a hybrid architecture, comprising two levels [NZT96]. Moreover, as new applications need more computing power, it is likely that newer and faster computing components will be available that will be added to the system. In this way the nodes in the inner level of these environments may differ in processor speed, amount of memory, number of disks etc. MIDAS is currently working on such a hybrid heterogeneous architecture. Load balancing and resource management for such systems is considered an open problem [HFV96].
- *Adaptability:* The cost estimates provided by the optimizer, respectively parallelizer components are often inaccurate. The reasons for this are various, like e.g. obsolete statistical information, the difficulty to find correct estimation functions (especially for intermediate results), etc. The problem is further aggravated in the case of parallel object-relational systems that allow users to define data types, methods and operators. For instance, selectivity estimation for such user-defined constructs is still an area that is poorly understood [KD98]. Hence, it is important to recognize unforeseen situations and to (re)act correspondingly. In addition, as MIDAS supports real-life applications ([CJ+97], [NJM97]) we have to implement a strategy for the QEC that also keeps track of different workloads in a multi-query environment.
- *Robustness*: An important aim of the QEC component is to provide a faultless execution. For instance, an overload of the system, even in the presence of several resource-intensive applications, should be avoided. This applies also to deadlocks that can come to existence due to waiting situations for common resources. In the same time, the QEC component has to guarantee that the cumulated memory requirements of the concurrently running queries do not exceed the overall available database cache.

In order to satisfy these requirements, we propose a *distributed, two-phas*e approach. Hereby, the information concerning the system state is made on all nodes available. Thus, the scheduling task can be accomplished on any node, i.e. no bottlenecks can occur. We call this first phase *coarse scheduling*. The corresponding decisions take the current global database system state and the provided cost formulas into account. Thereby, the parameters for query execution resources are chosen in a way that matches both resource availability and contention. As a result of this first phase, the constituting subplans of a PQEP are assigned to different processing sites that are favorable form a global point of view. However, the exact schedules for these subplans, i.e. start and wait times, are decided by the local component of the QEC. This *fine scheduling* is based on the actual runtime parameters of the corresponding processing site. It accounts more exactly for the resource consumption, dependency situations among the participating subplans and other related aspects.

The chapter is organized as follows. An overview on related work is given in Section 7.2. Furthermore, Section 7.3 provides a detailed description of the basic QEC strategies. In Section 7.4

the main phases of the scheduling task are presented. Finally, in Section 7.4 a summary completes the chapter.

7.2 Related Work

Numerous techniques have been proposed for resource management and load balancing in parallel database systems.

In XPRS [HS93] the plan fragments are divided in *CPU-bound* and *I/O-bound* tasks and further scheduled accordingly. However, the proposed allocation algorithm is only applicable for shared-memory systems, as it is massively based on global control. To cope with the increasing complexity, some reported work only concentrates on scheduling of joins in a way that maximizes the effect of certain forms of parallelism. Apart of the fact that the linear tree schedulings proposed in [SD90, MD95, ZZS93] risk to utilize the resources inefficiently, the concepts cannot be applied for forthcoming query types, like OLAP, holding also other complex operators.

A memory allocation scheme for multiple-query workloads is addressed in [MSD93]. An adaptive algorithm taking into account different query classes is proposed. However, the study is limited to single join queries on a centralized database system. In [LC+93] processor assignment algorithms for pipelined hash-joins in a shared-disk environment are presented. In this work, communication costs and independent parallelism are ignored.

In [GI96] a multi-dimensional resource scheduling algorithm is proposed. However, it is restricted to identical preemptive resource sites. As memory is not preemptive, it is not considered in this model, which is not realistic w.r.t. multi-query environments. Moreover, the assumptions made by the authors are prohibitive for the usage of this algorithm in a real-life system without changes: no memory limitations, clones of the same operator placed strictly on different sites, as well as non-increasing operator execution times. In [GI97], a more generalized model is presented, that includes also memory consumption as a so-called space-shared resource. Assuming that the output of an operator tree is always repartitioned to serve as input to the next one, the degrees of parallelism for each operator are calculated independently. The results shown in the previous chapters show that the optimal degree of parallelism of a set of operators differs from the optimal degree of parallelism of each stand-alone operator, as in this way repartitioning can be avoided and larger blocks can be constructed, thus saving resources. Apart of this, still identical sites and only simple hash joins are considered, additionally assuming that the build relation fits always into the memory.

In [BFV96], the topic of load balancing in a hierarchical shared-nothing database system is discussed. Apart of the fact that here all nodes are considered to be identical as well, no other resources than CPU is considered. Our measurements for intra-operator parallelism show that scheduling the instances on different machines may result in high execution time skew, even if the data is evenly partitioned (no data skew). This results in a higher overall execution time, because the partitioned subquery takes as long as the slowest of its instances. Moreover, the effects with regard to the resource utilization (idle times) are the same as with data skew.

7.3 QEC Strategies

As presented in Chapter 4, the goal of the TOPAZ parallelizer is to come up with parameterized parallel query execution plans. Some of these parameters are related to resource consumption, such as requested memory, degree(s) of parallelism etc. On the other hand, TOPAZ also transmits the corresponding anticipated cost measures. These measures are calculated by using the cost model described in Chapter 5. It is the task of the QEC to make the final adjustment for resource parameters according to the run-time system state and to initiate and control the execution of the constituting subplans.

For this purpose, it is crucial to use a multi-dimensional resource model as proposed in [GI96] and [GI97]. Above, we have already pointed out some drawbacks of this model with regard to architectural and query processing assumptions. One important critic concerning the model itself is the dimension independence assumption. Our measurements show that this is not confirmed by reality, as for instance memory allocated to a hash-join allows the operator to use a larger hash table, thus reducing disk contention. Thus a resource manager has to take both memory and disk bandwidth into account to balance resource contention [DG95]. We generalize this statement by saying that in order to provide better overall performance for all forms of parallelism in a multi-query environment, the QEC has to use a multi-dimensional resource model extended for heterogeneous hybrid architectures and the cost models and algorithms adapted to keep track of the functional dependencies between dimensions.

In the following, we will describe some of the core strategies of the QEC component towards achieving this aim.

7.3.1 Distributed Approach

Most related work on scheduling parallel query execution plans in PDBMSs [GI96, GI97, SD90, MD95, ZZS93] is based on a *centralized* approach. In this model, the scheduler is a monolithic component, mostly already incorporated within the parallelizer. The advantage of such a strategy, where all subplans are initiated and managed by a single component, is that the knowledge on concurrently running tasks and anticipated costs is also centralized. Hence, better estimations on the global system can be made, that in turn permits a better load balancing. However, in a parallel DBMS supporting real-life applications and multi-query environments, such an approach is not beneficial due to performance and availability reasons.

Hence, we propose a *distributed* approach, where the scheduling task can be performed on any processing site, similar to the optimization and parallelization. Thus, in the MIDAS system architecture presented in Fig. 2.3, the QEC becomes part of the application server. As described in Section 2.2, application servers are assigned exclusively to DBMS clients. Hence, for each new client, a new QEC is started that manages only the queries coming from the respective application. The location for a new application server, respectively new QEC component, within the parallel system is decided by the MIDAS Server taking into account load balancing aspects. That means that in MIDAS the QEC components themselves are also uniformly spread across the constituting processing sites.

However, in order to perform an optimal scheduling of the incoming PQEPs, each QEC component also has to keep track of the global system state. This is determined by the resource characteristics of the constituting nodes of the PDBMS, as well as the queries submitted by other concurrent applications having resource demands on their own. The strategy to derive this information without incurring too much overhead is described in the following section.

7.3.2 QEC Input Information

The decisions taken by the QEC component are based on information gathered from different components.

First, the estimated *cost information* is provided by the parallelizer, respectively optimizer. This is given for each subplan as a multi-dimensional array, corresponding to the accounted resources. As presented in Section 5.5, these cost components for each parallel plan variant are calculated with respect to some given constraints w.r.t. non-preemptive resources, as e.g. memory.

Second, in a hybrid heterogeneous environment as MIDAS, the QEC component also has to keep track of the *configuration characteristics of each processing site* within the parallel DBMS. In MIDAS, the description of each processing site consists of the number of processors, cache size, processor speed and attached disks.

Third, up-to-date information on the *current system state* is equally important w.r.t. load balancing aspects. Thereby, the incurring overhead, in terms of messages etc., should be kept as low as possible.

One possibility is to inquire the current system state for each scheduling task. However, this implies repeated message exchanges with all processing nodes, hence unacceptable overhead. Moreover, in a distributed approach as previously proposed, additional effort is necessary to guarantee that the information received at the beginning of the scheduling task is not obsolete by the time the subplans are actually assigned to processing nodes. This gives rise to similar problems as encountered in the field of transaction management, solvable only by locking schemes or similar complex strategies and thus incurring non-negligible overhead.

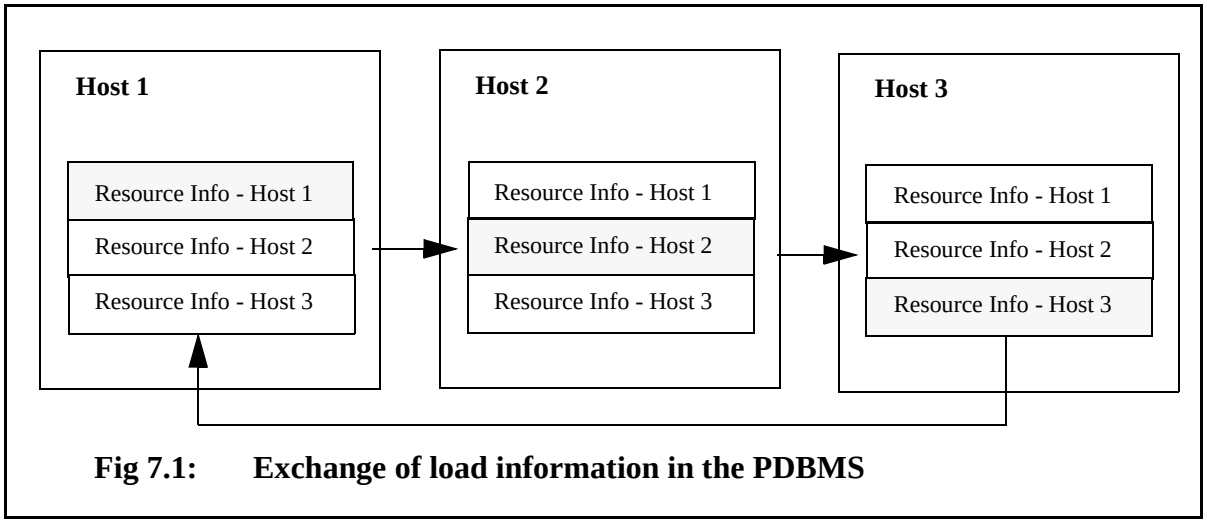


Fig 7.1: Exchange of load information in the PDBMS

Hence, we decided to decouple the information update concerning the system state from the actual scheduling task. The corresponding strategy is depicted in Fig. 7.1. Thus, the exchange of load information among the different processing nodes is provided by a separate subsystem. Thereby, each processing site keeps a shared-memory data structure containing information on the current and planned resource utilization for each processing site of the PDBMS. In the following, we will use the term “planned” for resources that correspond to subplans that have already been assigned to a processing site, but have not yet come to evaluation (due to the inexistence of input tuples, materialized fronts or other fine-scheduling reasons). The information update, based on PVM [Ge94] messages, is cyclic, as shown by the arrows in Fig. 7.1 and independent of any scheduling tasks. Thereby, during each iteration, a processing site first receives the current load information from a neighboring processing site, then updates within the data structure only the information concerning the processing site itself (marked in Fig. 7.1 by a gray background) and finally sends the entire data structure to the next node. The advantage of such an approach lies in the fact that the load information is available to other components as well, as e.g. the transaction manager [Zi99]. In addition, the communication overhead is constant and independent of the number of submitted queries, respectively QEC components.

However, the cyclic update scheme as presented above implies also a delay in the exchange of information among the constituting processing sites. Thus, this information channel is not suitable to provide detailed information, as e.g. start and end times for each subplan etc. Rather than this, more coarse-grained and implicitly less time-sensitive data is exchanged. Examples of such data for each processing site are given in the following: the total number and cumulated costs of the currently running, as well as planned subplans, the total number of already accredited, as well as planned cache pages etc.

7.3.3 Two-Phase Scheduling

The design of QEC component(s) has to support the loose coupling between the scheduling task and the information subsystem, as presented in the previous section. In such an environment, detailed information about the execution of each subplan on every processing site is not available. However, the requirement w.r.t. robustness as mentioned in Section 7.1, is imperative. Hence, the scheduling task is divided into two phases. The corresponding component design is depicted in Fig. 7.2.

The first phase, called *`coarse scheduling`*, is incorporated within the application server and thus is assigned exclusively to a DBMS client. Hence, it belongs to the dynamic part of the QEC. It takes the global load information and system parameters (as described above) into account to assign the incoming subplans to processing sites. Thereby, only the cumulated required resources of a subplan and the cumulated planned, respectively already assigned resources of the processing sites are taken into consideration. Thereby, the plan variant which fits best into the current runtime environment is chosen. Thus, the main task of this phase is *load balancing* and *resource management* on a global level.

The second phase, called *`fine scheduling`*, is assigned to a processing site. Hence, it the static part of the QEC component. It is in charge of elaborating a detailed and accurate schedule

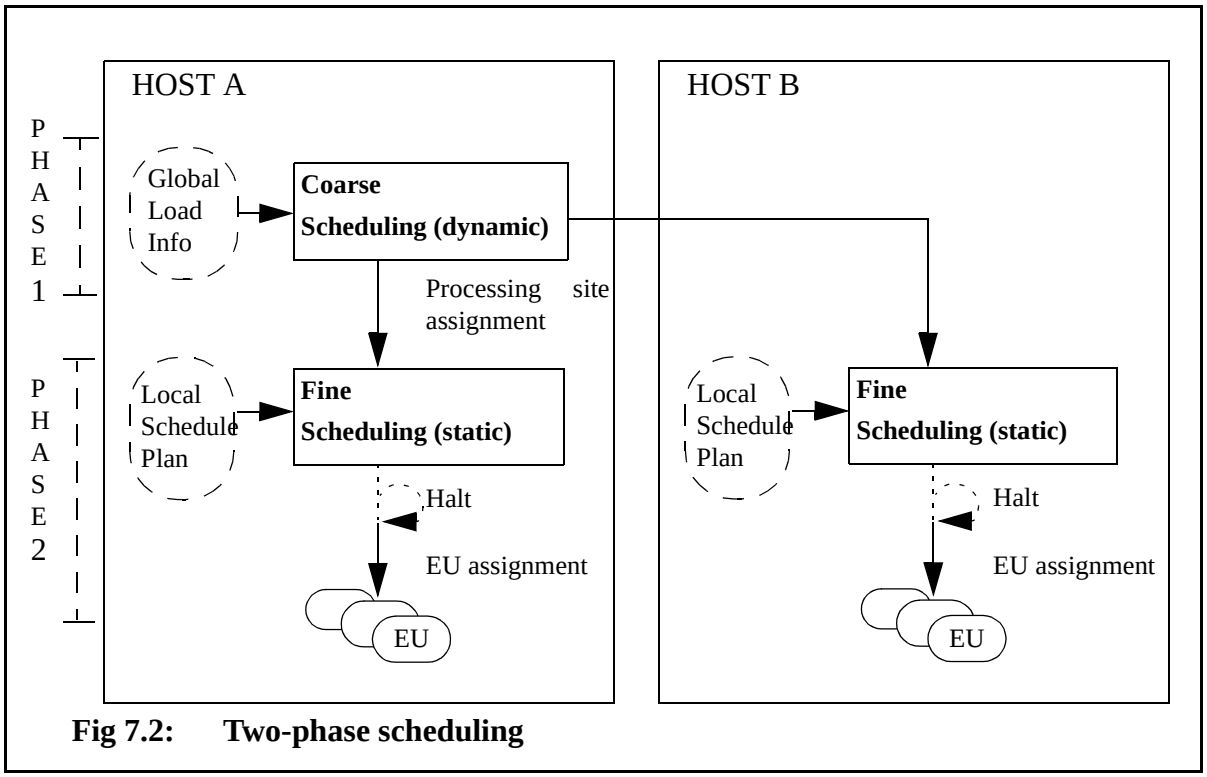


Fig 7.2: Two-phase scheduling

plan for the corresponding site. In contrast to Phase 1, it manages the subplans coming from different applications. Thereby, constraints such as available memory, data dependencies, materialized fronts etc. are also taken into account. If e.g. the memory requirements of a given subplan can not be satisfied, the subplan is halted. Thus, the main task of this phase is to guarantee *robustness* and *adaptability.*

7.3.4 Management of non-preemptive resources

By shifting the exact scheduling of subplans to the corresponding processing sites, more accurate planning is possible. In this way the usage of non-preemptive resources, like memory, can be maximized. In the previous chapters we have shown that the start of subplans can be delayed by *materialized fronts*. Hence, if all subplans of a PQEP are started at the same time, some of them will remain idle until they receive their first input tuple. However, in order to guarantee the necessary robustness, the memory requirement of this subplan is already taken into account and hence cannot be used by other subplans. Thus, it is important to schedule such subplans only when they can start their effective evaluation, i.e. when their first input tuple is available.

The other aspect is related to *non-uniform resource utilization* during the execution of a subplan. Although this is acceptable in the case of preemptive resources, for non-preemptive resources it can cause scheduling delays and thus throughput problems. The non-uniform resource utilization is caused by the fact that operators in a subplan have different resource requirements. This is determined on one hand by the implemented algorithm (i.e. different memory requirements of the *hash-join* vs. the *sort* operator). On the other hand, it is influenced

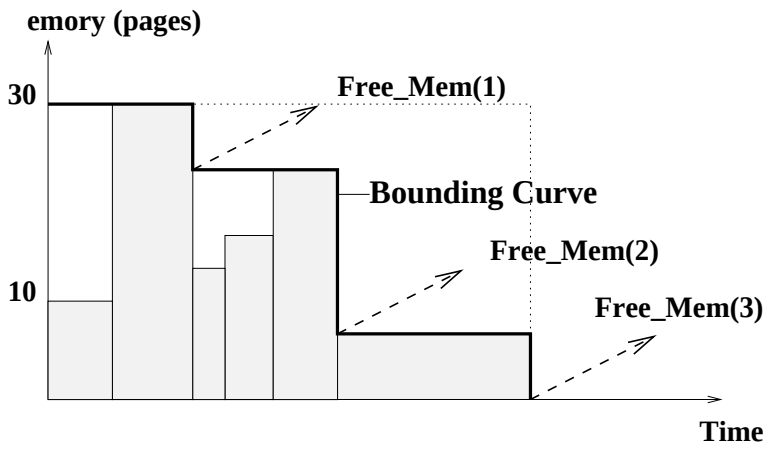


Fig 7.3: Memory distribution

by the input cardinality of the operators. The memory distribution for an example subplan is given in Fig. 7.3. By associating to this distribution a strictly decreasing bounding curve, the reserved memory accredited exclusively to this subplan can also be decreased accordingly. Thus, possible waiting tasks can already be started *during* the execution of the subplan. In Fig. 7.3 this corresponds to the times *Free_Mem(1)* and *Free_Mem(2)*. This is in contrast to other approaches, where the entire reserved memory is made available to other subplans at once, namely only at the end of the execution of this subplan (corresponding to the time *Free_Mem(3)* in Fig. 7.3).

Both aspects, i.e. materialized fronts and non-uniform resource consumption, have not been taken into account even in recently published models [GI97].

7.4 QEC Phases

In the following, we briefly describe the functionality of the two phases of the QEC component in MIDAS.

7.4.1 Phase 1: Coarse scheduling

As described in Section 7.3.1, the first phase, called coarse scheduling, is realized within the dynamic part of the QEC component.

One solution to the problem of adaptability is to dynamically adjust the resource allocation for the subplans and to keep track of the configuration and load of each processing site. The QEC uses the information described in Section 7.3.2 to assign subplans to processing sites that are most suitable from the resource utilization point of view. Instances of the same subplan are executed on similar machines, if possible. If not, the resources accredited to subplans are decremented according to the node's characteristics and the run time of the slowest instance. In this way, these resources are available for other plans or subplans. In high load situations it is even

beneficial to choose a plan variant that has a higher estimated response time, but requires less resources.

Another important aspect in a shared-disk environment is to minimize the overhead introduced by the distributed database buffer, in our case the VDBC. It is essential to reduce the number of multiple page copies in local database buffers at different nodes and to achieve high buffer hit ratios. When considering access costs, data locality is extremely important. This refers not only to disk versus buffer access but also to local and remote buffer access. The costs of remote buffer access comprises both transfer costs for the accessed pages and the costs due to increased resource/data contention between nodes. To support data locality, we introduce a logical assignment of data partitions to nodes. For example, for a relation scan it is desirable to execute it at the node where currently most of the pages to be scanned are buffered. This saves communication, disk I/O, and CPU costs. But sometimes it might be better to assign the scan to another node due to load balancing purposes. Costs related to data contention (e.g. lock waits and deadlocks) have to be considered as well.

With the concept of data locality, the QEC can easily be adapted for shared-nothing environment as well. In such situations, the data locality is treated as a constraint, instead of an option as in a shared-disk environment.

The assignment of subplans to processing sites is done as follows:

The constituting subplans of a PQEP are first sorted in ascending order of their start time. This is determined by their materializing fronts, as described in Section 5.3.3. For subplans having the same start time, the second sorting criteria is the local cost of the respective subplans. This sorted list is then processed starting with the first element, i.e. subplan. For each subplan, the demanded resources, as calculated by the cost model, are matched against the available and planned resources of every processing site. Therefore, the corresponding resource vectors are linearized, i.e. transformed into *cost values*, similarly to the approach presented in Section 5.5.2. Thus, for each subplan, the processing site having the most favorable cost value is chosen. The available resources of this processing site are then decreased accordingly and the process continues in a similar way for the next element, i.e. subplan, of the sorted list.

In this way, first those subplans are taken into account that can immediately start their evaluation, thus providing the necessary input for the subsequent, dependent subplans. On the other hand, the strategy accounts also for the local costs of the subplans, assigning high-cost subplans to the most favorable processing sites. This reduces the overall response time of a query.

When this process is finished for all subplans of a PQEP, the first phase of the QEC is also in charge of transmitting the subplans to the corresponding processing sites.

7.4.2 Phase 2: Fine Scheduling

For each processing site, there exists one (static) QEC component performing the second scheduling phase, called fine scheduling. It receives subplans from different applications, i.e. different dynamic QEC components.

This phase is in charge of elaborating an accurate and efficient schedule plan for the tasks

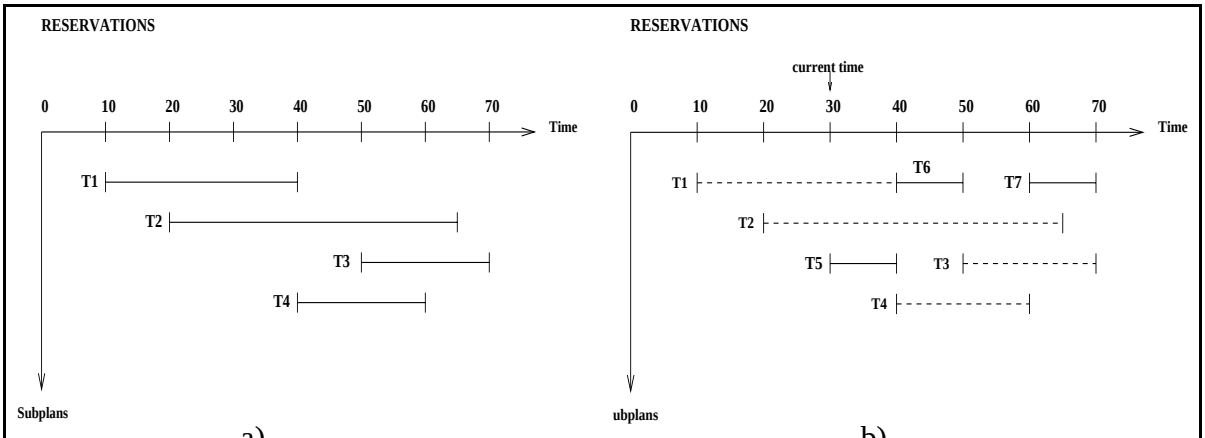


Fig 7.4: Fine scheduling

assigned to the given processing site. Thereby, more detailed information can be taken into account as compared to the first phase: data dependencies between subplans, memory requirements, the existence of input tuples etc. This can be partly achieved by using the cost estimates provided by TOPAZ. For instance, as presented in Chapter 5, the cost model accounts for the materialized front of subplans as well. Hence, subplans having a start time $T_{start}>0$ can already be scheduled corresponding to the estimated delay. However, for the realization of this aim, a detailed schedule, respectively reservation plan is necessary.

The strategy for fine scheduling taking the resource main memory correctly into account is presented by the example given in Fig. 7.4. For simplicity reasons, this example does not deal with any non-uniform resource utilization distribution, as discussed in Section 7.3.4. Assume that on a processing site with a totally available memory of *30* pages, the following 4 subplans have been assigned:

	T_{start}	T_{end}	*requested memory*
1.	10	40	10
2.	20	65	10
3.	50	70	10
4.	40	60	10

The corresponding schedule, respectively reservation list is depicted in Fig. 7.4a. Assume that at the time *30*, *3* additional subplans are assigned to the processing site:

	T_{start}	T_{end}	*requested memory*
5.	0	10	10
7.	10	20	10
3.	0	10	10

Taking into consideration the current time ($T = 30$), this results into the following time spans:

5. (30, 40)
7. (40, 50)
3. (30, 40).

Thus, the reservation list can be completed by adding subplans T_5 and T_6 accordingly. However, for subplan T_7, the total memory of *30* pages is already assigned in the requested time period *(30, 40)*. Hence, in order to assure a correct processing, this subplan must be postponed and can only start at time *60*, as shown in Fig. 7.4b.

Thus, the basis of the second QEC phase are schedule plans. In this way, the usage of available resources is maximized, at the same time guaranteeing the required robustness. However, these plans are constructed by using the cost estimated provided by the parallelizer. As already mentioned, these estimates are often based on simplifying assumptions as well as on statistical information. Hence, the second QEC phase also has to take care of the necessary adaptability by counterbalancing some wrong cost estimates made by the optimization or parallelization.

The two main tasks related to the response to unforeseen situations are the following:

- **Forward correction:**
  With this strategy, the QEC reacts on early terminations and corrects the plan correspondingly. For instance in Fig. 7.4b, if the evaluation of T_4 is completed before time *60*, subplan T_7 can be scheduled earlier as well. However, a necessary condition for this action is that the materialized front of T_7 is already completed, i.e. the subplan can effectively start processing.
- **Backward correction:**
  This strategy applies for the following two situations:
  1. The evaluation of a given subplan takes longer than initially estimated by the parallelizer.
  2. The data rivers that serve as input for a given subplan are not yet filled. As already mentioned, this situation comes to existence if the materialized front of a subplan takes longer than initially estimated.
  The backward correction accounts for the resulting delays, and postpones the evaluation of subsequent subplans.

At the same time, the QEC also guarantees that the result of the two strategies described above still yields a correct schedule plan at any time, i.e. satisfying the imposed constraints, as e.g. the memory limits.

7.5 Summary

In this chapter we have presented our approach to scheduling, load balancing and resource management incorporated within the QEC component. This approach is based on a multidimensional resource model. In contrast to other strategies, the necessary scalability is provided by a distributed design. Adaptability and robustness are combined within a 2-phase strategy. Thus, the first phase focuses on global system parameters while the second phase is in charge of elaborating a schedule plan related to each processing site by taking local aspects into account. Thereby, the incurring overhead, in terms of messages etc., is also minimal. Since the scheduling algorithm accounts also for the individual architectural characteristics of each processing site, it is applicable for forthcoming hybrid heterogeneous architectures as well.

The proposed approach is implemented within the MIDAS system, yielding promising first results. We plan to address a detailed performance evaluation to our future work.

Chapter 8

Parallelization of User-Defined Functionality

In this chapter we validate our approaches towards achieving efficient intra-query parallelism for complex applications, thereby concentrating especially on execution, extensibility and adaptability issues. Therefore, we have chosen a specific class of applications, called in the following *stream-oriented* applications. These are characterized by the fact that their data often takes the form of (time) series, or more generally streams, i.e. an ordered sequence of records. Analysis of this data requires stream processing techniques, which differ in significant ways from what current database query techniques support today. In this chapter we present a new operator, called *StreamJoin*, to solve stream-related problems of various applications, such as data mining, pattern recognition and universal quantification. We show how this operator can efficiently be embedded in the database engine, thus implicitly using the optimization and parallelization capabilities presented in the previous sections for the benefit of the application.

8.1 Introduction

In the past years the importance of efficient stream processing techniques within the database system is continuously increasing. This can be shown by the examples listed next.

- *Data Mining*

  One of the core mechanisms of most data mining algorithms [AM+95] is a phase that evaluates patterns called *frequent itemsets*. A frequent itemset is a set of items appearing together in a number of database records meeting a user-defined threshold, called *support*. The final itemsets are usually derived using a set of *candidate* itemsets. That means, a candidate itemset is established as being a frequent itemset if the number of transactions containing ***all*** items in the candidate itemset exceeds the predefined support.

- *Universal Quantification*

  Forthcoming applications stress the need for efficient implementation of universal quantification concepts. Given e.g. a decision support system (DSS), a frequently formulated query type is the following: "*Find the customers/suppliers/stores that buy/supply/sell* ***all***

items that satisfy a particular condition".

- *Sequences*

  Recently, there is a growing interest in *periodicity search* [HGY98] in time-related databases. Given for instance a stock data database, one frequently formulated query is: *"Find **all** stock items that monotonically fall/rise in a given time period"*.

- *Pattern Recognition*

  In the domain of molecular biology, scientists often match functionally unknown proteins against a protein coding database of known proteins. If a match is found, it is likely that the two proteins are functionally related. Thus, given a sequence and a protein coding database, the problem can be formulated as follows *"Find the item sequences in the database system that contain **all** items of the pattern sequence in the given order"*.

- *Digital Libraries*

  Modern digital libraries offer a *profiling* service to keep track of their user's individual reading interests. A profile usually consists of a set of keywords, that are usually specified in a specific order. One task of the profiling service is to find the documents that contain ***all*** keywords. In some cases, the words in the documents have to fulfill certain additional position requirements [NJM97].

If we analyze in more detail the above mentioned query types, we recognize that a common feature of all application domains is the need for effective processing of a `variable` number of intermediate result tuples, or `streams`, to produce the final result. This number is not known a priori, as it corresponds e.g. to the number of items in an itemset, or the number of search terms in a profile.

However, stream analysis is not effectively supported by current database engines. Our solution to this problem is to extend the database functionality by a new operator, called *StreamJoin*, whose stream processing technique is applicable in various application domains. We will explain the functionality of *StreamJoin* based on an example coming from the data mining area, i.e. frequent itemset generation. In the subsequent sections, we will show the applicability of this operator in other domains as well. For each domain, we present possibilities of further increasing performance by means of intra-query parallelism, obtained according to the concepts presented in the previous chapters.

8.2 Applicability of *StreamJoin* for the Evaluation of Frequent Itemsets

Data mining is an emerging research area, whose goal is to extract significant patterns or interesting rules from databases. High-level inference from large volumes of routine business data can provide valuable information to businesses, such as customer buying patterns, shelving criterion in supermarkets and stock trends. In the following, we will concentrate especially on the issue of performing data mining directly on the data stored in a data warehouse.

As the amount and complexity of data in the warehouse reaches previously unthinkable proportions, it becomes more difficult to identify trends and relationships in the data using simple query and reporting tools. Data mining can provide new insights into the relationships between data elements and provide analysts and decision-makers with new discoveries. Given the above, integrating the data warehouse DBMS with data mining makes sense for many reasons. First, obtaining clean, consistent data to mine is a primary challenge, implicitly provided by data warehouses. Second, traditional mining tools would typically transfer and copy the data, an operation that is prohibitively costly in the case of large-scale warehouse applications. Third, it is advantageous to mine data from multiple sources to discover as many interrelationships as possible. This requirement is also fulfilled by warehouses that contain data from a number of sources. An integrated approach provides also the capability to go back to the data source to ask specific, newly-relevant questions. Finally, the continuously extended functionality of the database engines, including parallelization, can also only be used in an integrated environment.

One of the basic operations in data mining is the discovery of frequent sets. Given a set of transactions, where each transaction refers to a set of items, this operation consists of finding itemsets that occur in the database with certain user-specified frequency, called *minimum support*. The derived itemsets can be used for further processing, such as association rule mining [AY97], time series analysis, cluster analysis etc.

However, special requirements have to be fulfilled when integrating frequent itemset generation with large-scale databases, such as data warehouses. First, due to the large amounts of data stored, multiple database scans cause prohibitive overhead in terms of I/O costs. Second, it is essential to provide adequate pruning techniques to reduce the exponential complexity of search space exploration. Finally, the mining algorithm itself has to prove high efficiency. Most algorithms proposed and integrated into a database engine are variants of *Apriori* [AM+95]. However, this strategy scales exponentially with the longest itemset length, thus reducing its applicability for scenarios involving both high data volumes as well as considerable item domains, such as e.g. data warehouses.

Itemsets that have no superset that is frequent are called *maximal frequent itemsets (MFI)*. The set of all maximal frequent itemsets, also called the *maximal frequent set (MFS)*, implicitly defines the set of all frequent itemsets as well. Based on this observation, we propose a novel methodology to efficiently evaluate the maximal frequent set only, called *MFSSearch*. This strategy is based on a new operator, called *StreamJoin*, that efficiently calculates the support of a candidate itemset, as well as of all of its prefixes. A dynamic pruning technique that combines both top-down as well as bottom-up techniques is used throughout search space exploration. As a result, the complexity of the algorithm does not depend exponentially on the length of the longest MFI and is only proportional to the *MFS volume*, i.e. the sum of the lengths of all MFIs.

The strategy can efficiently be embedded into the database engine, resulting in a uniform processing scheme within a single query execution plan. We show how the approach can easily be expressed in augmented SQL using user-defined table operators [JM99] and common table expressions [SQL99]. The most striking difference to other approaches lies in the drastically reduced I/O overhead. Thus, instead of performing multiple scans of the whole database, only selective disk accesses are necessary, depending on the current search space status. Intermediate

result materializations or preparatory phases are not necessary, either. Moreover, the processing scheme is non-blocking, i.e. first results (MFIs) can be delivered fast before the whole MFS is derived. Furthermore, we point out how this approach can make full profit of the parallelization possibilities of the database engine, as presented in the previous chapters, yielding linear speedup and thus further performance improvements. In contrast to other approaches, we use the existing disk partitioning of the database, since repartitioning or even (selective) replication of the entire data is impracticable for operating data warehouses.

8.2.1 Related work

Integrating data mining with databases has gained growing interest in the research community. However, almost all approaches are based on the bottom-up *Apriori* algorithm, thus bearing also the main drawbacks of this strategy, namely exponential complexity and multiple database scan operations [AS96, AY98]. Additionally, the proposed approaches also involve some kind of intermediate result materializations or preprocessing of data. On the other hand, some of the proposed improvements of the *Apriori* algorithm are not suitable for integration with data warehouses, as they perform modifications on the stored data [PCY97].

When comparing different possibilities of integrating the *Apriori* algorithm within a DBMS, in [STA98] the most promising scenario was found to be one based on a so-called *Vertical* format of the database. However, this format requires also a preparatory phase that creates for each item a BLOB containing all *tids* that contain that item. We discuss this work in more detail later on in this section.

Recently, solutions using also top-down or hybrid search strategies were proposed, such as the *MaxMiner* algorithm [Ba98], *PincerSearch* [LK98] or *MaxClique* and *MaxEclat* [ZP+97]. However, although the pruning strategies of these algorithms reduce the search complexity considerably as compared to *Apriori*, they still involve multiple database passes or even preparatory phases. This has a negative effect on performance, as demonstrated later on in this section. In addition, it is not clear how these approaches can be efficiently integrated in the database engine. Moreover, the algorithms presented in [ZP+97] are also based on a vertical database format, making it impractical for existing large warehouse applications.

Other approaches concentrate on language extensions [MPC96, MPC98, HF+96] that are based on special operators to generate association rules. However, as shown in [Ch98], for the easy development of data mining applications it is important that the constituting operations are unbundled so that they may be shared. Thus, a better alternative is to provide a primitive that can be exploited more generally for different data mining applications. The strategy for MFS calculation proposed in this section follows exactly this recommendation.

8.2.2 Data Mining Scenario

The decision on whether a given candidate itemset is frequent is the performance-critical operation in the MFS calculation. In our approach we want to directly map this primitive operation

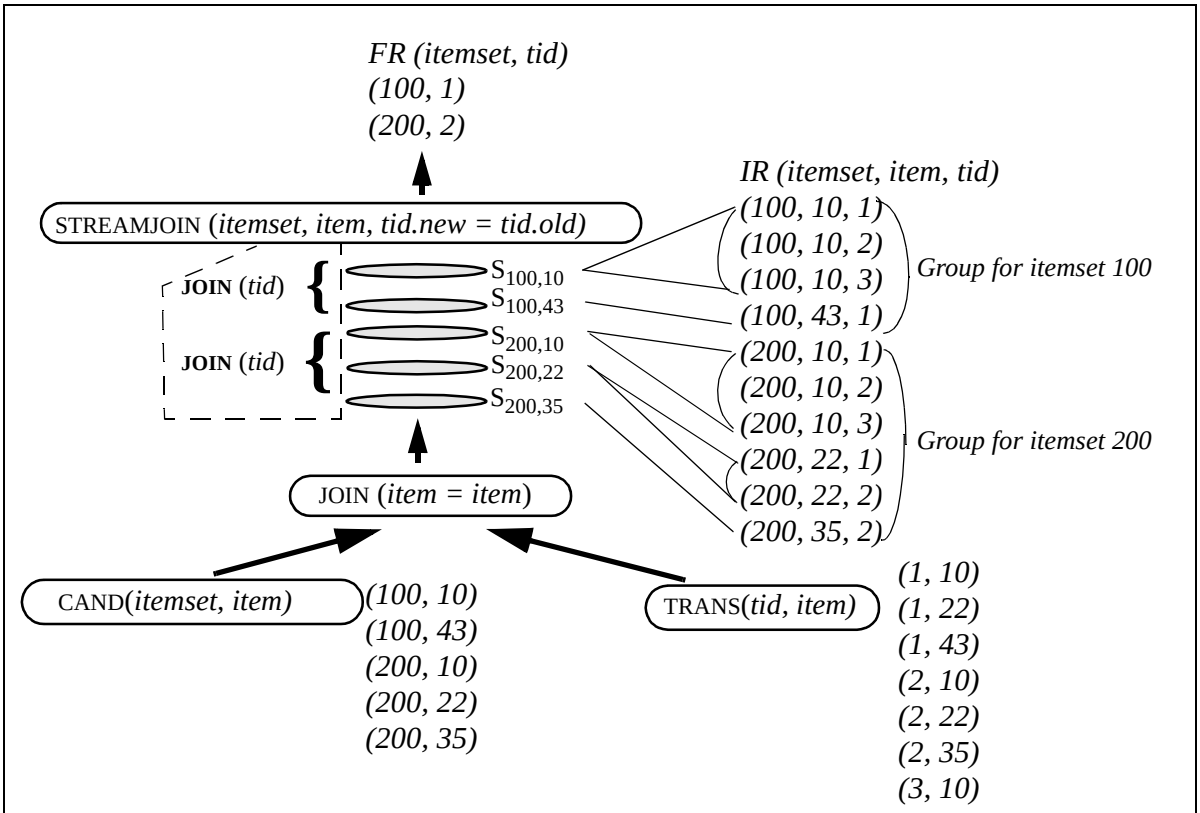


Fig 8.1: Processing scenario for the evaluation of frequent itemsets

to a single database operator. In order to model this scenario we assume the following two tables: TRANS(*tid, item*) giving report of which transactions contain which items and CAND(*itemset, item*) telling which itemsets contain which items, i.e. the potential frequent itemsets. An example of two itemsets (*100* and *200*), three transactions (*1, 2* and *3*) and four items (*10, 22, 35, 43*) is shown in Fig. 8.1. This relational modeling of the data mining scenario supports that both the number of items per *tid*, as well as the number of items per *itemset* is variable and unknown during table creation time. In contrast, other representation alternatives, as e.g. all items of a *tid* appearing as different columns of a single tuple are not useful in practice [STA98]. Given this scenario and a parameter *minsup* defining the minimal support set by the user, the problem of deciding for a potential frequent itemset *IS* in the CAND table whether *IS* is frequent can be formulated as follows:

> "Find (through the TRANS table) those transactions containing **all** items of *IS*. If the sum of the qualifying transactions exceeds *minsup*, then return *IS* as being a frequent itemset."

Evaluating this query involves a join on *item* between the two tables TRANS(*tid, item*) and CAND(*itemset, item*) for itemset = *IS*. This yields a set of tuples *(IS, item, tid)*. We will call the subset of those tuples for a given itemset which contain one specific item *I* a *stream*, denoted by $S_{IS,I}$. The streams for one specific itemset form a *group*. This means in general, every tuple from the CAND table defines one stream.

For the task to find frequent itemsets, the streams only form an intermediate result (*IR* in Fig. 8.1). E.g., for our potential frequent itemset *IS* we must find those transactions which contain **all** the items of IS. This means that for the final result (*FR* in Fig. 8.1) we have to join the

different streams of the itemset *IS* on the *tid* attribute, i.e. we have to join all the streams within a group. In Fig. 8.1 we have an example for two itemsets, *100* and *200*, which contain two resp. three items. Therefore two streams resp. three streams are built for the itemsets. Joining the streams yields for this example into a two-way resp. three-way join. However, in the general case we do not have knowledge on the number of items per itemset, hence the number of joins to be performed on a group is variable as well.

This task is a kind of all-quantification. At the same time, it is a very primitive operation within the processing of frequent itemsets. Hence, any efficient evaluation of this all-quantification directly supports the performance of the frequent itemset processing. Since all-quantification is not yet a frequently occurring operator in database processing, there are rarely implementations of it available [CK+97]. Here we designed our own solution, called *StreamJoin* operator, which perfectly fits into our algorithms. The operator will be introduced in the next section. Thereafter we will describe the particularities of our algorithm for efficient MFI candidate generation.

8.2.3 The *StreamJoin* Operator

We first describe the functionality of the operator: *StreamJoin* basically memorizes the incoming tuples as long as they belong to the same stream. Then, these tuples are joined with the next stream. This procedure continues for all streams of a group (i.e. for all items within a candidate itemset), such that at the end, only those tuples survive that support all streams within a group, i.e., all items within the given itemset. The *StreamJoin* operator has the following signature:

StreamJoin (Group-ID, Stream-ID, pred(Join-ID1,Join-ID2, ...))

Two parameters specify the columns that define a group and the streams within a group; here, these parameters are *itemset* and *item*. The subsequent parameter defines the join predicate.

This join predicate defines condition(s) between attribute values of the current stream and those of the previous stream. This is expressed by the suffixes *new* and *old*. Thus, *Join-ID.new* represents the value of the attribute *Join-ID* for the current stream and *Join-ID.old* represents the value of the same attribute in the previous stream.

For example, in Fig. 8.1 the join predicate *tid.new = tid.old* defines a simple equi-join on the *tid* attribute. Thus the operator joins subsequent streams of the same group on the *tid* column, as explained in the previous section. However, more complex predicates can be used as well to support e.g. pattern matching or sequence analysis, as presented later on in this chapter.

We now come to some implementation issues: The input has to be grouped on the *Group-ID* and *Stream-ID* attributes. Obviously, this requirement can always be fulfilled by adequate sorting techniques. However, explicit sorting can mostly be avoided by adequate pre-processing of the data in the very same query execution plan.

For instance, assume that the CAND table in Fig. 8.1 is sorted on *itemset*. Consider the following evaluation alternatives w.r.t. the join between TRANS and CAND:

- an index-nested-loops join, using an index of the TRANS table on the *item* attribute; this is possible in almost all cases, since in most data warehouse schemas the central table has several indexes on the dimension attributes.

- a hash join, the CAND table being used as the probing table; please note that for an item domain containing *l* items the number of possible candidates is 2^l and thus the size of the CAND table might even exceed the size of the TRANS table.

In these cases, the join result is constituted as follows: for each tuple *(itemset, item)* of the CAND table a set of tuples *(itemset, item, tid)* is generated, yielding exactly a stream, i.e. the transactions that contain that specific item. For instance, the tuple *(100, 10)* has generated the stream $S_{100,10}$ consisting of the tuples *(100, 10, 1), (100, 10, 2)* and *(100, 10, 3)*. Hence, the necessary grouping of the intermediate result *IR* for the *StreamJoin* processing is already satisfied and no additional sort operations are necessary.

Based on this, our preliminary implementation for *StreamJoin* uses a dynamic hash-based approach with two hash tables. The strategy is illustrated in Fig. 8.2 for an example group, representing group *500*, constituted of four streams, derived from four items *1*, *2*, *3* and *4*. The first stream of each group is used to build the first hash table. The next stream is probed against this hash table, the matching tuples being inserted into the second hash table. At the beginning of the next iteration the first hash table is deleted, and the second is used for probing. Similar to the previous iteration, the matching tuples are used to build up the new contents of the first hash table. This process continues until the next group is reached, or either a result of a probing phase or a constituting stream is empty. At the same time, the intermediate result (i.e. hash table) sizes decrease with each iteration, as the tuples which don't match the join condition are eliminated.

Please note that this description of *StreamJoin* shows certain similarities to the evaluation of recursive queries in database systems [CCW93]. Indeed, though the two areas seem radically different because of the approach and formalism used, they have some common features. First, the presented hash-based implementation of *StreamJoin* is similar to the transitive closure algorithm described in [HWF93]. Second, both approaches apply a variable number of consecutive iterations and a stop condition to obtain the final result. Moreover, in both cases the number of iterations is not known a priori, being dependent on the value distribution of the input. However,

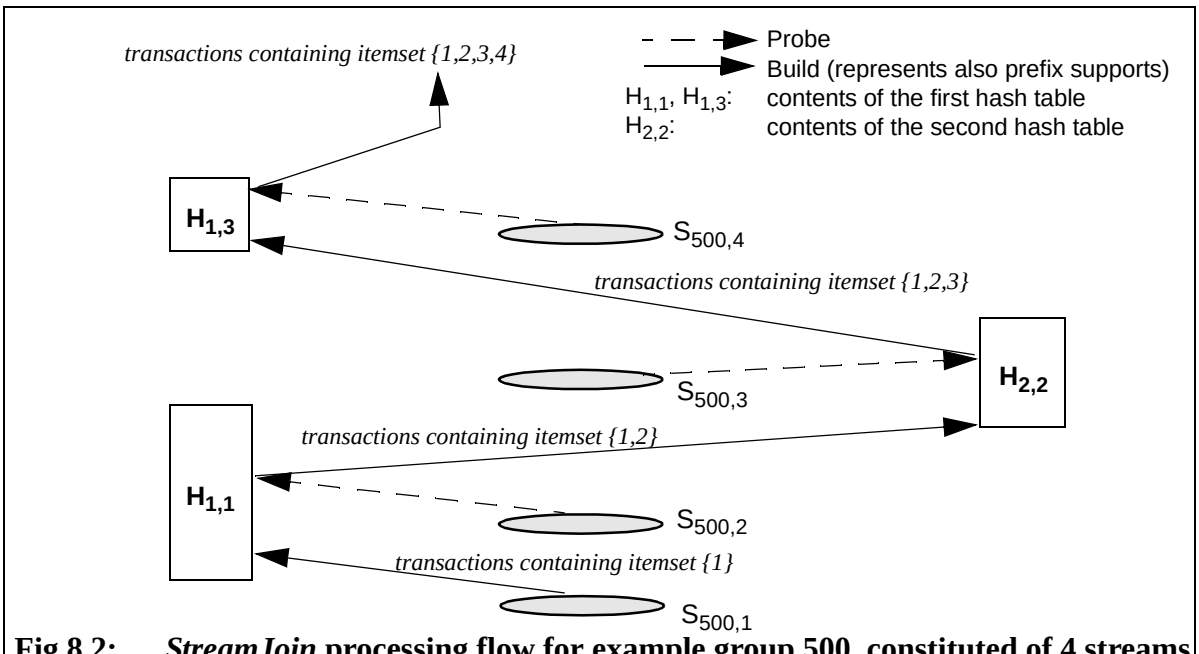


Fig 8.2: *StreamJoin* processing flow for example group 500, constituted of 4 streams

the difference between the two approaches lies in the characteristics of the iterations. As already mentioned, *StreamJoin* processes its input *stream-wise* or *linearly*, i.e. a given input tuple is only considered once. In contrast, the evaluation of transitive closure, or recursion in general, requires a repetitive processing of the input. This *cyclic* processing may consider the same input tuple several times in order to produce the complete result. Another important difference between the two approaches lies in the fact that the *StreamJoin* processing *reduces* the intermediate result with each iteration, as the tuples which don't match the join predicate are eliminated. In contrast, transitive closure produces in each iteration new tuples that are added to the final result. Finally, there is also a major difference between the two stop conditions applied. Thus the transitive closure algorithm stops when no new tuples are produced, i.e. the (intermediate) result set of a given iteration is identical to the one of the previous iteration. In contrast, the *StreamJoin* algorithm joins streams of the same group until the subsequent group is reached. Thus, the stop condition for joining is the fact that the value of the *Group-ID* attribute for the current tuple is different to that of the previous input tuple.

In the following, we point out some characteristics of the *StreamJoin* processing. As indicated in Fig. 8.2, the continuous arrows also represent the transactions that contain the prefixes of the example itemset *{1,2,3,4}*. Hence, the corresponding supports can be easily evaluated by a simple subsequent *count(tid)* operation. Thereby, the frequent itemsets are those for which the calculated support exceeds *minsup*. In general, the following observation is valid:

> ***Observation 1:*** Given an itemset *X = {1,2,..., N-1,N}*, by processing this itemset via the *StreamJoin* operator, we also obtain the supports of all prefixes *{1}, {1,2}... {1,2,...N-1}*. ❒

Hence, given a transaction table TRANS and a table CAND with candidate itemsets, the support of the candidates as well as of their prefixes can be efficiently evaluated within the database, performing a join on the two tables and pipelining the intermediate result *IR* into the *StreamJoin* operator, as already shown in Fig. 8.1.

8.2.4 Definition of the Candidate Itemsets

An open question is how to guide the search space exploration in order to reduce search complexity and expensive database scans. In the hypothetical scenario from Fig. 8.1 we supposed that all candidate itemsets are stored in the CAND table. However, given the exponential complexity of the search space, this is impracticable for real-life applications and item domains. Hence, it is desirable to fill the CAND as much as possible with MFI candidates only. In our solution this sophisticated task of generating suitable candidates for the CAND table is performed by the *MFSSearch* algorithm. The strategy expands the search space gradually, starting with the itemset containing all items. The *StreamJoin* processing is employed for the calculation of individual candidate supports. The produced results are used for the generation of further candidates. Thus, *MFSSearch* guides the search space exploration dynamically by already derived intermediate results. In the following, we describe this strategy in detail. For simplification purposes, theorem proofs as well as detailed algorithms can be found in Appendix C.

Given an infrequent itemset *X = {1,2,...,N-1,N}*, in a top-down search it is necessary to test all of its subsets of level *N-1*. This can be done by successively eliminating the items *N-1, N-2,...1*

from X. It is not necessary to do this with item N, since X -$\{N\}$ is a prefix whose support is implicitly evaluated together with the support of X by the *StreamJoin* operator (see *Observation 1*). In the following, we will call the set of items needed to generate all unexplored subsets of level N-1 for a given itemset X the *ElimList* of X or E_X.

In this case $E_X = \{1,2,...,N-1\}$. The subsets situated on the same level will be called *siblings*. However, if this procedure of generating subsets by eliminating the first N-1 elements is applied recursively, duplicates are generated. This follows from the following observation:

> ***Observation 2:*** If X_1 and X_2 are two subsets of itemset X, obtained by eliminating two different items from X, then X_1 and X_2 differ by only one item position. ❒

Suppose $X_1 = \{1,2,...,A,Z,B,...,N-1\}$, $X_2 = \{1,2,...,A,Y,B,...,N-1\}$.

If this process is done recursively, one subset of X_1 will be obtained by eliminating item Z: $X_{1Z} = \{1,2,...,A,B,...,N-1\}$. Similarly $X_{2Y} = \{1,2,...,A,B,...,N-1\}$ and $X_{1Z} = X_{2Y}$.

In order to avoid duplicate generation, if X_2 is expanded after X_1, the *ElimList* of X_2 must not contain Y. More generally, if the siblings $X_1, X_2...,X_{N-1}$ are expanded successively, the *ElimList* of a given itemset X_i must not contain any positions which differentiate this itemset from any of its siblings $X_1,..., X_{i-1}$. This can be done easily, if we decrease the original *ElimList* of X by one element in order to get the appropriate *ElimList* for each subset generated.

Thus for $X = \{1,2,..., N-1, N\}$ and $E_X = \{1,2,...,N-1\}$ we have the following sibling subsets and *ElimLists*:

$X_1 = \{1,2,...,N-2,N\}$, $E_1 = \{1,2,...,N-2\}$,
$X_2 = \{1,2,...,N-3, N-1,N\}$, $E_2 = \{1,2,...,N-3\}$,
....
$X_{N-1} = \{2,...,N-1, N\}$, $E_{N-1} = \varnothing$.

Fig. 8.3 shows a search space generated by the *MFSSearch* algorithm after employing only the *ElimList* technique described above, i.e. without any additional pruning as described later. We will call the subsets expanded by an itemset itself *direct subsets*, while subsets expanded by a sibling are called *cross subsets*. For instance $\{36\}$ is a direct subset of $\{136\}$, while $\{16\}$ is a cross subset of $\{136\}$, expanded by $\{126\}$.

Based on the information given so far, we can now derive the following two basic properties.

> **Theorem 1:** The *ElimList* method guarantees a full expansion of the search space without duplicate generation.

As a conclusion, given a superset X on level N with the *ElimList* E_x and a subset X_i generated by eliminating item N-i from X, the *ElimList* of X_i is $\{1, 2,..., N-i-1\}$. This method of successively decreasing the *ElimList* for sibling subsets guarantees a complete and duplicate-free search space expansion.

Obviously, any subset of a frequent itemset is also frequent. Hence it is only necessary to expand the direct subsets of a given itemset if this itemset is infrequent. We call this form of pruning *Direct Top-Down Pruning (DTDP).*

> **Theorem 2:** The upper bound for the number of itemsets considered by the *MFS-Search* algorithm for a finite item domain $1,2,..., N$ is 2^{N-1}.

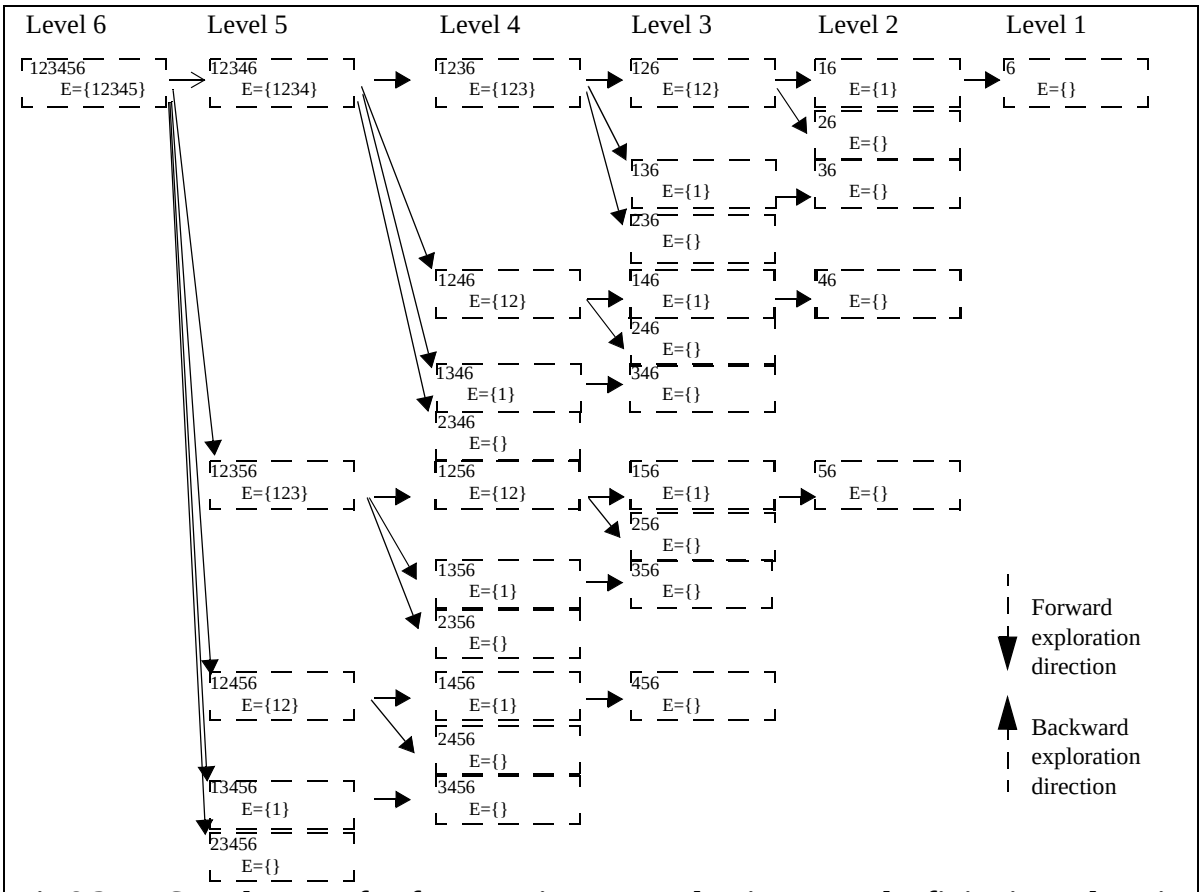


Fig 8.3: Search space for frequent itemset evaluation over the finite item domain: 1,2,...,6.

Thus, by employing this basic version of *MFSSearch* (*ElimList* technique and DTDP) the search complexity has already been reduced by a magnitude of 2.

Sibling subsets can be explored either *backward* or *forward*, as marked in Fig. 8.3 by arrows. The following section will detail on that and a summary of all possible and meaningful combinations of the various pruning and search strategies is given in Section 8.2.7.

8.2.5 Backward Exploration (BE) of Itemsets

An important property for all backward-oriented strategies to be discussed below is given by the following theorem.

Theorem 3: Given two itemsets X_1 and X_2, s.t. $X_1 \subseteq X_2$. In the BE scenario X_1 will be processed *after* X_2.

8.2.5.1 Cross Top-Down Pruning

In this section, we are interested in finding out how a given frequent itemset can prune cross subsets as well. In Fig. 8.3, if e.g. *{1456}* is frequent, direct top-down pruning eliminates *{456}*, but it is desirable to prune the cross subsets *{156}* and *{146}*, expanded later on by *{1256}*, respectively *{1246}*, as well. We call this *Cross Top-Down Pruning (CTDP).*

Theorem 4: Given a finite item domain *1,...,N*. In the backward exploration any item-

set except itemset $Z=\{N\}$ is a superset of at least one itemset that is not expanded yet.

Hence, once an itemset is found frequent, it can prune its (direct and cross) subsets from exploration. However, at a given moment, the search space consists of itemsets that are either a) expanded and explored, b) expanded or c) unexpanded. The fully explored search space is given in Fig. 8.3. Evidently, pruning can only affect category b) and c), i.e. not yet explored itemsets. Itemsets of category b) can be pruned together with their direct subsets as soon as a frequent superset is found. However, it is not clear how to prune itemsets of category c). In Fig. 8.3, if e.g. itemset $X=\{456\}$ is found to be frequent, it can prune itemsets *{56}, {46}* and *{6}*. However, these are itemsets on Level 2 and 1, none of which have been expanded yet at the moment when X is explored. Thus, it is necessary to memorize X for itemsets that have not yet been explored, if these itemsets can produce subsets of X. In this case, these are the itemsets *{12356}* and *{12346}*. Such a set of *relevant frequent itemsets,* called *FrequentSet,* is logically assigned to each itemset element, similarly to its *ElimList*.

Theorem 5: The set of frequent itemsets F assigned to any candidate itemset contains only maximal frequent itemsets.

Given a frequent itemset X and an expanded but unexplored itemset Y (category b), with Y having a direct subset Z that is not expanded yet (i.e. of category c), s.t. Z is also a cross subset of X. Now the question is how to prune the search space in order to avoid the evaluation of Z.

Theorem 6: Given a frequent itemset X and an expanded but unexplored itemset Y, $X \not\subset Y$ and the *ElimList* of Y being E. Y will expand a subset of X if $\{y | y \in Y, y \notin X\} \subset E$. This will be further on referred to as ***Condition (1).***

A relevant frequent itemset will be propagated to lower levels only if this condition is fulfilled.

Example 8.1: If the itemset $X=\{456\}$ is frequent, it will be included in the *FrequentSet* of the expanded but unexplored itemsets *{12356}* and *{12346}* that also satisfy *Condition (1)*. When these itemsets are explored, they will further propagate X only to the subsets *{1256}, {1246}* and *{1236}* on Level 4. This process continues and leads finally to the pruning of the itemsets *{56}, {46}* and *{6}*.

8.2.5.2 Bottom-Up Pruning

Bottom-up pruning (BUP) uses the property that if a subset of an itemset in the search space is found infrequent, it is no longer necessary to explore that itemset as it is infrequent anyway. According to Theorem 3, in the BE scenario an itemset can never be a subset of an itemset that is expanded later. Thus, an entire itemset can never be used for BUP. However, according to *Observation 1*, by processing an itemset via the *StreamJoin* operator, the supports of all prefixes are implicitly calculated as well. Thus, we can use infrequent prefixes for BUP.

Definition 1: Given an itemset $X = \{1,2,...,N-1,N\}$ with prefixes $X_1 = \{1\},...,X_N = \{1,2,...,N-1,N\}$

The *maximal infrequent prefix (MIP) of X* is $\begin{cases} \emptyset, \text{ if } X \text{ is frequent} \\ X_i, \text{ s.t. } X_i \text{ infrequent and } X_j \text{ frequent, } j < i. \end{cases}$

Thus, an early termination condition for the processing of an itemset X via the *StreamJoin* operator is finding the maximal infrequent prefix of X.

Example 8.2: Given a minimal support of *10* and $X=\{2,3,4,5,6\}$. Assume that the following

supports have been calculated: *sup{2} = 88, sup{2,3} = 51, sup{2,3,4} = 9, sup{2,3,4,5} = 7, sup{2,3,4,5,6} = 1*. Thus the *MIP* of *X* is *{2,3,4}* with the corresponding support *9*. Once this prefix is found, it is not necessary to probe the remaining elements of *X*, namely *5* and *6*, as at this point it is known that *X* is infrequent.

With top-down pruning as described in the previous section, once a superset of an itemset *X* has been found frequent, we could prune *X* together with all its direct subsets, as they are also all frequent. This is not always possible in BUP. More precisely, if a subset *Y* of an itemset *X* is found infrequent, we can prune *X*, but not all direct subsets of *X*, as they might not include *Y*.

Example 8.3: Assuming that in the backward exploration from Fig. 8.3, the *MIP* of itemset *{23456}* is found to be *{23}*, also *{12356}* is infrequent and can be pruned. However, from its direct subsets only *{2356}* contains *{23}*, while the others still have to be explored.

Theorem 7: In bottom-up pruning, a maximal infrequent prefix *X* can prune an itemset *Y* in the search space together with its direct subsets if $X \subset Y$ and $ElimList_Y$ doesn't contain any items from *X*. We will further refer to this as ***Condition (2)***.

Example 8.4: Suppose that in the backward exploration from Fig. 8.3, the *MIP* of itemset *{456}* is found to be *{4}*. In this case, *{4}* can prune the whole branch rooted at *{1246}* since the *ElimList* of this item is *{12}* and thus every direct subset also includes *{4}*.

Similar to top-down pruning, in order to incorporate also unexpanded itemsets in the BUP, it is necessary to keep a *set of infrequent itemsets* that are relevant for the direct subsets of a given itemset *X*, called IF_X.

A given prefix can be evaluated multiple times, within different itemsets.

Theorem 8: Given a finite item domain *1,2,...,N*. In the BE scenario the last time a prefix $P=\{P_0, P_1,..., P_n\}$ is evaluated is within the itemset $X=\{P_0, P_1,..., P_n, N\}$.

We will further refer to the prefix $X \setminus \{N\}$ of an itemset *X* as $PMax_X$. From Theorem 8 results that given an infrequent itemset *X*, its maximal infrequent prefix *MIP* can only be a subset of an itemset that is not explored yet if $|MIP| < |X| - 1$. We will further call this formula ***Condition (3)***. Indeed if $|MIP| = |X| - 1$, then $MIP = PMax_X$ and according to Theorem 8 this prefix will not be expanded further on.

Example 8.5: If the *MIP* of itemset *X={456}* is *{45}*, there is no sense to perform bottom-up pruning with this prefix, as it is not included in any itemset still to be explored.

Another important result of Theorem 8 is that if $|MIP| = |X|$, the prefix $PMax_X$ is also a maximal frequent itemset. We will this formula ***Condition (4***) in the algorithm description given in the appendix C.2.

Indeed, from *X = {1, 2,..., N}* and *X* is infrequent and $|MIP| = |X|$, i.e. *MIP = X*, results that $PMax_X$*= {1, 2,...,N-1}* is frequent. According to Theorem 8, there is no other itemset in the search space still to be explored that includes $PMax_X$. On the other hand, there is also no other superset of $PMax_X$ explored earlier that has been found frequent, as in this case the itemset

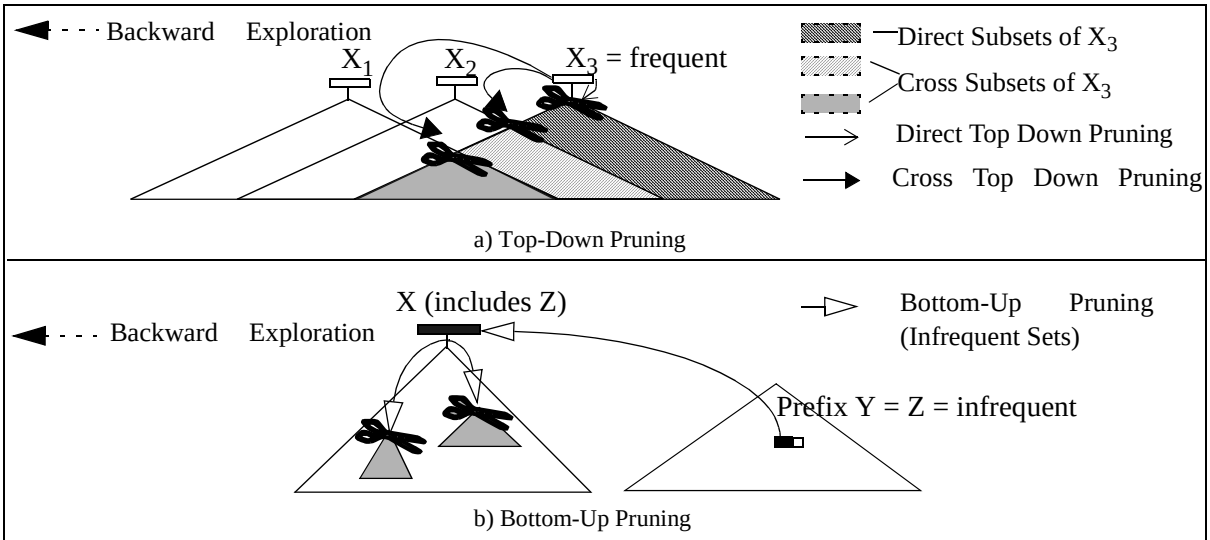


Fig 8.4: Pruning strategies in backward exploration scenario

would have been eliminated by top-down pruning. From this results that $PMax_X$ is a maximal frequent itemset.

Fig. 8.4 shows the reduction of the search space through the pruning techniques presented so far. In Fig. 8.4a the frequent set X_3 prunes its direct subsets as well as its cross subsets expanded by X_1 and X_2. In Fig. 8.4b the infrequent prefix Z prunes both itemset X as well as subsets of X that satisfy *Condition 2*.

8.2.6 Forward Exploration (FE) of Itemsets

From Theorem 3 results that in the FE scenario, an itemset A can be a subset of an itemset B that will be explored later. If both A and B are found frequent, A cannot be a MFI. Thus, contrary to BE, in the FE scenario it is possible to generate also frequent itemsets that are not maximal. Hence it is necessary to have some filter mechanisms that return only MFIs. This can be realized by e.g. explicitly maintaining a *set of maximal frequent itemsets* throughout the exploration. Once a frequent itemset X is found, it is added to this list and eventual subsets of X have to be eliminated, if existing. Thus, at the end of the algorithm the set contains the MFS only.

Similar techniques are used also in [Ba98, LK98]. The disadvantage of this approach is that in this way the maximal frequent itemsets can only be returned when the whole search space exploration is finished. More precisely, if FE is realized within the database engine, this would yield a blocking boundary, as all input has to be processed before the first output tuple, i.e. MFI, is delivered. Please note that in the BE scenario this is not necessary, since once an itemset is found to be frequent, it can immediately be returned, as Theorem 3 guarantees that it is also maximal.

We will further concentrate on pruning possibilities for the FE scenario. DTDP can be realized in the same way as described for BE. However, according to Theorem 3, in the FE scenario no itemset explored at a given time has cross subsets that are expanded later. Hence, CTDP is not applicable at all.

8.2.6.1 Bottom-Up Pruning

In the FE scenario, either entire itemsets or maximal infrequent prefixes can be used for BUP. However, contrary to BE, if we use entire itemsets for pruning, we cannot simply discard an itemset from exploration if one of its subsets is found infrequent. As shown in Section 8.2.5.2, each itemset X in the search space stands in reality for two itemsets, namely X and $PMax_X = X \setminus \{N\}$. If we find an itemset $Y = \{Y_0, Y_1, ..., N\}$ to be infrequent and $Y \subset X$, we can discard X from evaluation, but we still have to evaluate $PMax_X$, as this itemset is not a superset of Y. Only if $N \notin Y$, X can be totally discarded from evaluation.

Example 8.6: If in the forward exploration from Fig. 8.3 itemset *{246}* is infrequent, it can prune *{12456}* from evaluation, but it is still necessary to evaluate its prefix *{1245}*. However if the *MIP* of *{246}* is *{24}*, *{1245}* is infrequent and can be pruned as well.

In the backward evaluation scenario, we don't have to consider this problem, because as shown in Section 8.2.5.2, only maximal infrequent prefixes can be used for BUP.

Please note that the *MFSSearch* algorithm for both the backward and forward exploration scenarios is given in the appendices C.2, respectively C.3.

8.2.7 Summary of Pruning Techniques

As detailed in the previous sections and shown in Fig. 8.4, the following pruning techniques have been developed for efficient generation of maximal frequent itemsets:

- **Direct Top-Down Pruning (DTDP)** prunes the direct subsets of an itemset X. The technique ensures that these subsets will only be expanded and explored if X is infrequent. DTDP is applicable to both backward and forward exploration strategies.
- **Cross Top-Down Pruning (CTDP)** ensures that unexplored cross subsets of frequent itemsets are eliminated from exploration. CTDP makes use of a list of relevant frequent itemsets assigned to each expanded itemset, called *FrequentSet*, that is propagated selectively towards not yet explored subsets. It is only applicable to the BE scenario.
- **Bottom-Up Pruning (BUP)** eliminates supersets of infrequent itemsets from exploration. Analogously to CTDP, BUP makes use of a list of relevant infrequent itemsets. BUP is applicable to both exploration strategies. However, a tailoring to the associated strategy has to be provided.

A summarization of the pruning techniques and their application to BE and FE is given in Tab. 8.1.

Please note that the terms "bottom-up" and "top-down" have been also used in combination with the optimization strategies presented in Chapter 4. However, the difference lies in the fact that w.r.t. query optimization, they denote the way in which the search space is expanded. In contrast, for the *MFSSearch* strategy the terms "top-down" and "bottom-up" describe only the applied pruning techniques, while the way in which the search space is expanded is described by the terms "backward" and "forward".

Table 8.1 Summarizing of the pruning techniques employed by the *MFSSearch* algorithm

	Backward Exploration (BE)	Forward Exploration (FE)
DTDP (*X=Frequent*)	• prunes direct subsets of *X*	• prunes direct subsets of *X*
CTDP (*X=Frequent*)	• adds *X* to *FrequentSet(Z)*, if *Z* expands a subset of *X (Cond. 1)* • prunes *Y* and its direct subsets, if $Y \subset X$	• not applicable
BUP (*X=Infrequent*)	• only possible if *X* is not an entire itemset *(Cond. 3)* • if $X \subset Y$, prunes *Y*; direct subsets of *Y* are pruned only if *Cond. 2* satisfied; else adds *X* to *InfrequentSet(Y)*	• if $X \subset Y$, but $X \not\subset PMax_Y$, prunes only *Y* direct subsets of *Y* are pruned only if *Cond. 2* satisfied; else adds *X* to *InfrequentSet(Y)* • if $X \subset PMax_Y$, prunes also $PMax_y$

Despite of this difference, the *MFSSearch* strategy still bears certain similarities with the search space exploration for database query optimization, as implemented e.g. in Cascades (see Section 4.3). In both cases, the search space is expanded gradually, depending on the current status and possible pruning techniques. Furthermore, the knowledge on already explored search space regions has to be managed by means of appropriate data structures. In this context, the proposed "frequent" and "infrequent" sets in *MFSSearch* have the same purpose as the "memo" structure in the Cascades Optimizer Framework, as described in Section 4.3.1.2. Finally, an important aim of both optimization strategies is to reduce the search space in order to improve optimization performance. This is achieved in both cases by appropriate pruning techniques, reduction or elimination of redundant work as well as of duplicates. Please note that this latter aim is realized in the *MFSSearch* algorithm by the *ElimList* technique, while in Cascades it is achieved by a duplicate-free rule set as described in Section 6.3.

8.2.8 Integration with the DBMS

In Fig. 8.1 we visualized our approach to evaluate the supports of itemsets and prefixes within the database by using the *StreamJoin* operator. In this scenario, the candidates are given by the (static) CAND table. Hence, in order to find the MFS, this table must contain all possible candidate itemsets, determined e.g. during a preprocessing step. However, as already mentioned, this approach is prohibitively costly in terms of time and disk space for real-life item domains.

Hence, in our solution the input for the *StreamJoin* operator is provided by the *MFSSearch* algorithm introduced in Section 8.2.3. In order to obtain an efficient and comprehensive integration of data mining with the data warehouse DBMS this task has to be performed in the database engine as well. In this section we present a strategy to efficiently map *MFSSearch* to database operators. The necessary flexibility will be provided by user-defined functions [SQL99] and user-defined table operators (UDTOs) [JM99]. UDTOs permit the definition of set-oriented operations within the database engine. They operate on one or more tables and possibly some additional scalar parameters and return a tuple or a table. The arguments (i.e. input tables) can

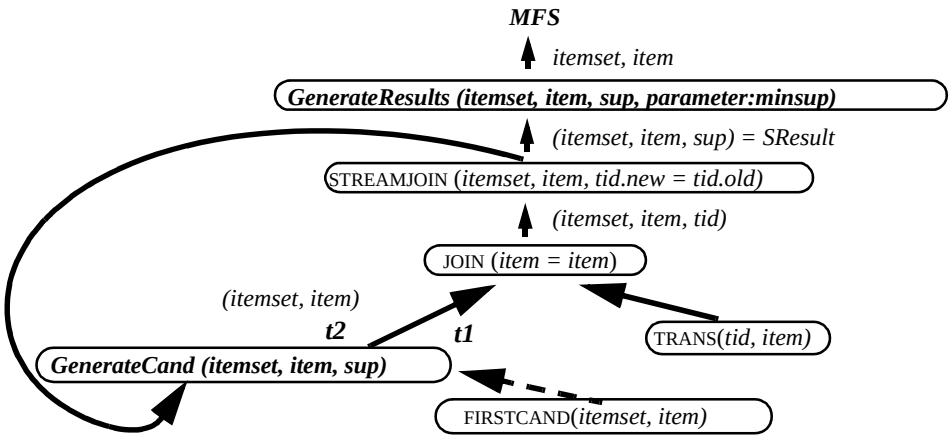


Fig 8.5: Mapping of *MFSSearch* on database operators

be intermediate results of a query, i.e. they are not restricted to base tables only. Thus the *StreamJoin* operator itself can be implemented as a UDTO as well.

In addition, we assume that the candidate generation algorithm is realized as a UDTO as well, called *GenerateCand*. As already mentioned, this algorithm starts with the itemset holding all items. Thus, this is the first itemset produced by the *GenerateCand* operator. Later on, since dynamic pruning is employed, the generation of further candidates depends on the results of processing the current candidate in the search space via the *StreamJoin* operator. This results into a cycle in the overall MFS generation scheme, as depicted in Fig. 8.5.

As the *GenerateCand* UDTO is incorporated within a cycle, candidate and result generation must be split up. In Fig. 8.5, the functionality of the *StreamJoin* operator has already been expanded to calculate also the aggregation on *itemset* as explained in Section 8.2.3, thus returning the support of each itemset as well as of all its prefixes. The resulting output stream is called *SResult(itemset, item, sup).*

This output stream is consumed by two operators: the *GenerateCand* UDTO and the *GenerateResults* UDF. The *GenerateCand* UDTO is only responsible for candidate generation. For easier understanding, we assume that it initially reads the first candidate, namely the itemset incorporating all items, from the table FIRSTCAND(*itemset, item*). This first input is used for internal initializations and transmitted unchanged to the subsequent operators *Join*, respectively *StreamJoin*. These calculate the corresponding (prefix) supports in a similar way as already presented in Section 8.2.2. In all subsequent iterations, the input of the *GenerateCand* UDTO is provided by the output of the *StreamJoin* processing, i.e. the *SResult* data stream. This intermediate result is used by *GenerateCand* to perform pruning as presented in the previous section and further span the search space. The resulting subsequent candidate itemsets are added to the output stream, thus starting new iterations. The process continues until no further candidate itemsets are available, i.e. the entire MFS is calculated.

The functionality for generating the final result is taken over by the *GenerateResults* UDF. This gets as input the result of the *StreamJoin* operator in the form *(itemset, item, sup).* The minimal support defined by the user is provided by means of a scalar parameter *minsup*. Thus, *GenerateResults* has the possibility to identify frequent itemsets, i.e. to perform a filtering functionality. As already explained, by employing the backward exploration for the *MFSSearch* candidate

```
SET minsup = myminsup;

WITH SResult(itemset, item, sup) AS
((SELECT StreamJoin(itemset, item, tid)
  FROM TRANS,
     (GenerateCand(SELECT itemset, item, -1 FROM FIRSTCAND)) AS t1
  WHERE TRANS.item = t1.item)
UNION ALL
(SELECT StreamJoin(itemset, item, tid)
 FROM TRANS,
     (GenerateCand(SELECT itemset, item, sup FROM SResult)) AS t2
 WHERE TRANS.item = t2.item))

SELECT GenerateResults FROM SResult
```

Fig 8.6: SQL representation using common table expressions, UDFs and UDTOs

generation, a frequent itemset found is also a MFI. Thus, *GenerateResults* can immediately add it to the output stream, yielding a maximal frequent itemset that is returned to the user. This results in fast response times and continuous input for further processing, such as e.g. association rule generation.

In the following, we focus on how the QEP from Fig. 8.5 can be expressed in (augmented) SQL. As already mentioned, *GenerateResults* can be realized by a UDF, as currently supported by most database vendors. However, UDFs can not be used for the *StreamJoin* and *GenerateCand* approaches, since they both deliver sets of tuples. This problem of expressing set-orientation can be solved by UDTOs [JM99], as presented in the following. As for the cycle within the QEP, this can be resolved in a similar way as recursion [SQL99], using e.g. common table expressions [Ch96].

By adequately using the above mentioned concepts, we obtain a single statement, as depicted in Fig. 8.6. Hereby, we have used a simplified syntax for a better understanding. The common table expression corresponds to the *SResult* stream. The first "SELECT *StreamJoin*" clause corresponds to the first iteration, where *GenerateCand* receives its input from the FIRSTCAND table, i.e. the itemset containing all items. This first input is used to initialize the search space, hence the value of the *sup* parameter is irrelevant (e.g. *-1* in Fig. 8.6). After initializing the search space, the *GenerateCand* operator transmits this first candidate unchanged to the *StreamJoin* operator. The corresponding output stream is called *t1* in Fig. 8.5 and Fig. 8.6. The subsequent iterations are expressed by the second input of the UNION operator. In this "SELECT *StreamJoin*" clause, the input of *GenerateCand* is already provided by the result of the *StreamJoin* operator, i.e.the *SResult* stream. This input is used to further explore the search space. The subsequent candidate itemset is added to the output stream *t2*, thus starting a new iteration.

In this way, the MFS calculation can be comprehensively expressed in (augmented) SQL. Thus the entire processing scheme or constituting parts of it can be referenced for other mining tasks as well. Please note that in contrast to other approaches [STA98], this strategy avoids intermediate table constructions as well as the formulation of separate SQL statements for each processing phase. Instead, as presented in Fig. 8.6, the entire MFS calculation can be expressed in a compact way by a single statement, thus query optimization and parallelization can be applied as usual for sake of increasing efficiency.

8.2.9 Parallelization Potential

In this section we concentrate on the parallelization possibilities of our approach. A critical aspect of parallelization is that the strategies work well with the existing physical data partitioning. This is especially important for an efficient integration of data mining with data warehouses. In contrast, most related work [HKK97, AS96] propose solutions that are based on proprietary partitioning strategies or even data replications, rendering these approaches inadequate for large-scale operating databases. In addition, for such applications communication and I/O overhead should be avoided as much as possible. However, most previous parallelization strategies [SK96, AS96] make repeated passes over the disk-resident database partition, thus incurring high I/O overhead. Moreover, in many cases they exchange their counts of candidates or remote database partitions during each iteration, resulting also in high communication overhead. Additionally, some of the approaches replicate complicated memory structures, thus inefficiently utilizing main memory.

In the following, we present parallelization approaches that maximize data locality, thus reducing communication as well as I/O overhead. Moreover, different kinds of physical disk partitionings of the data warehouse are taken into account.

Suppose that the TRANS table in Fig. 8.5 is the central FACTS table in a data warehouse star schema, holding also other attributes like *supplier, customer, time* etc. We differentiate two scenarios w.r.t. possible physical partitionings on disk, as discussed in the following.

8.2.9.1 Collocated Transaction Items

According to different application scenarios, the FACTS table can be partitioned in multiple ways [Schn97]. In Fig. 8.7 we propose a solution which is compatible with a partitioning strategy of

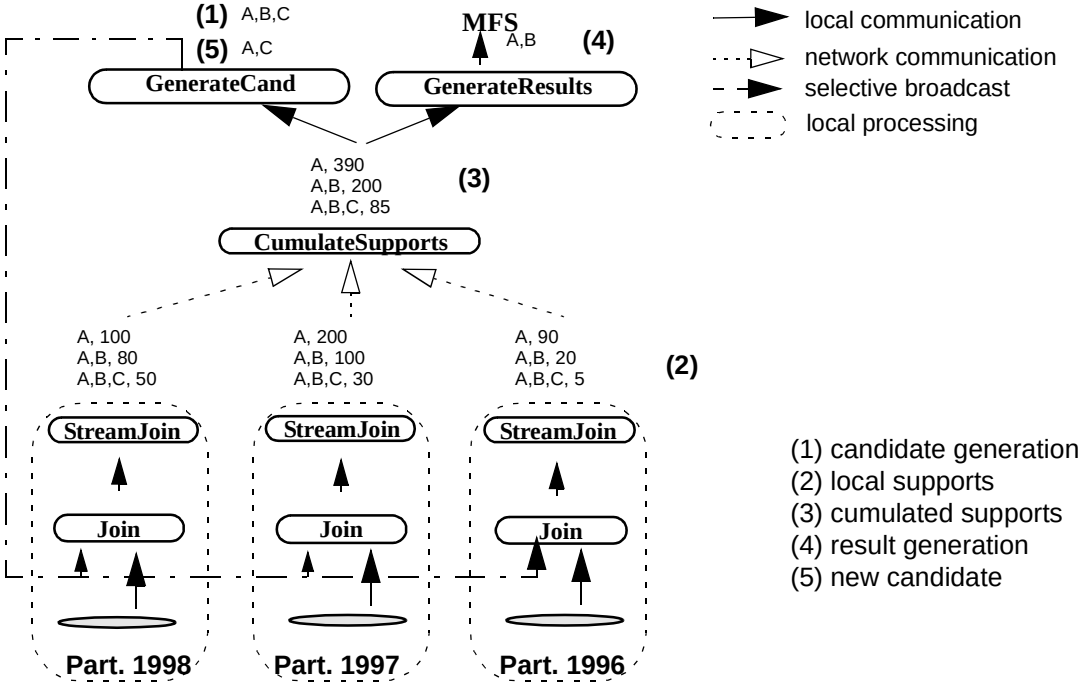


Fig 8.7: Parallel processing for the evaluation of maximal frequent itemsets; items belonging to a transaction reside on the same partition

the central FACTS table so that all tuples belonging to a single transaction are on the same partition. This is the case for instance if the partitioning is done on *time*, *tid*, or *customer* attributes.

In this case, each partition can calculate the local supports of the candidate itemsets by using the *StreamJoin* processing scheme. In Fig. 8.7, the first candidate itemset is *{A, B, C}*. Only the supports of the prefixes need to be communicated to a central merge operator, called *CumulateSupports*, that evaluates the final supports by adding up the local supports of the prefixes. This cumulated result is the input of both the central *GenerateCand* as well as *GenerateResults* operators. As described in Section 8.2.8, the *GenerateCand* operator decides on the next candidate itemset. This itemset, e.g. *{A, C}* in Fig. 8.7, is broadcasted to all participating nodes, thus starting a new iteration. The *GenerateResults* operator produces the final results holding maximal frequent itemsets as already shown in Section 8.2.8.

Hence, only candidate itemsets and computed supports need to be communicated over the network, producing only minimal communication overhead. This is similar to the *Count Distribution* algorithm [AS96]. However, in contrast to this strategy, performance is improved by avoiding multiple database passes and replicated memory structures.

8.2.9.2 Distributed Transaction Items

In the second possible scenario the FACTS table is partitioned in a way that doesn't guarantee that all items belonging to a transaction reside on the same partition. This is the case if the partitioning is done for instance on the *item* attribute.

A small modification of the *StreamJoin* operator allowing it to read streams from different inputs can also prevent from repartitioning. This is shown in Fig. 8.8, where the data warehouse is partitioned on the *item* attribute. When computing the support for candidate itemset *{A, B, C}*, the *StreamJoin* operator receives its input streams from different nodes, corresponding to the constituting items. Thereby, the *StreamJoin* operator can reside on any of the processing nodes. Please note the communication overhead is increased by the fact that the *tids* belonging to each

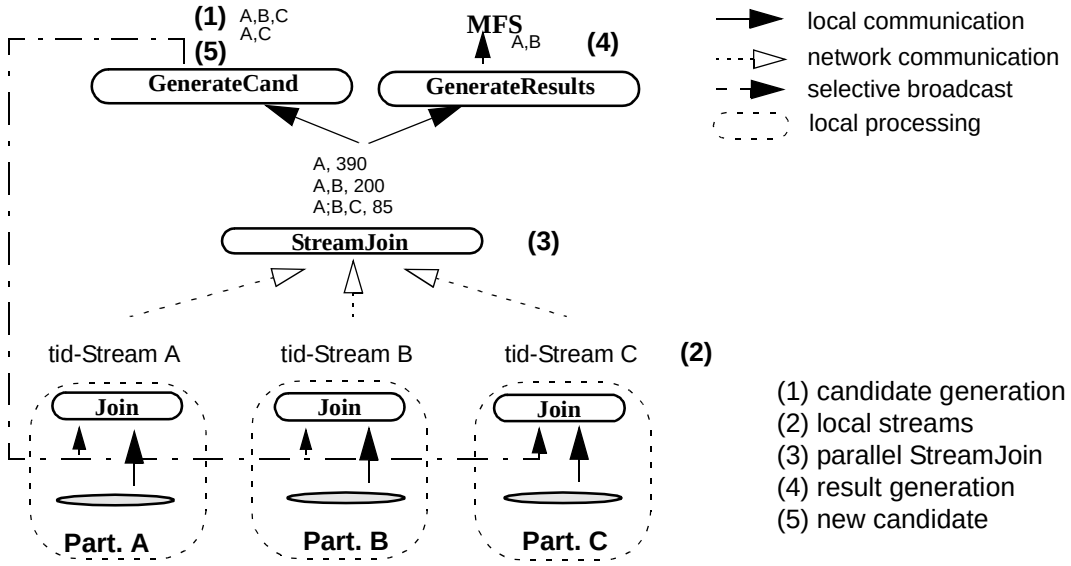


Fig 8.8: Parallel processing for the evaluation of maximal frequent itemsets; items belonging to a transaction reside on different partitions

item need to be communicated over the network. This can be reduced by executing the *StreamJoin* processing on the node corresponding to the item with the highest support. In this case, the most voluminous *tid*-lists don't have to be communicated over the network.

It is not necessary to broadcast the candidate itemsets to all partitions, either. If the partitioning function on the *item* attribute is known, a candidate itemset only has to be sent to the partitions that contain that item. For instance, in step (5) from Fig. 8.8, the new candidate *{A, C}* only has to be sent to partitions *A* and *C*. We do not know of any other parallelization strategy in the data mining area which would incur less communication overhead for this scenario without repartitioning or (selectively) replicating the database, as e.g. in [HKK97].

8.2.10 Implementation Aspects

In both scenarios, local and network communication as well as the (selective) broadcast can be realized by means of the data river concept presented in Chapter 3. Thus, in case of collocated transaction items, the merging of local supports (resulting from step (2)) can be realized using a *receive* operator adopting the MERGE strategy. In contrast, when transaction items do not reside on the same partition, the resulting *tid* streams have to be merged using the SEQ merge strategy. Cf. Section 3.3.4, this strategy guarantees that one entire data stream is consumed before the next is worked on.

Since the *StreamJoin* operator is realized according to the iterator concept, it can be incorporated both into the *TOPAZ* parallelizer as well as the *Model-M* optimizer. Thus, efficient intra-query parallelism can be provided to any execution plan that contains this operator.

In order to achieve this aim, *StreamJoin* first has to be incorporated into the cost model as a logical and physical operator in a similar way as the MIDAS operators presented in Chapter 5. This can be done in a straightforward way, since the operator is neither blocking nor necessitating any special treatment w.r.t. its cost calculation. Hence all cost formulas provided in Section 5.4.1.2 for regular unary operators apply for the *StreamJoin* operator as well. In this context, the base component T_{lbegin} is determined by the cost that is necessary to construct the first hash table of the *StreamJoin* operator. The local processing costs $T_{lprocess}$ can be derived from statistics by estimating the total number of groups as well as the (average) number of iterations that are necessary to process a group.

Second, appropriate rules have to be provided in order to make TOPAZ cognizant of the parallelization possibilities of the *StreamJoin* operator. These parallelization rules have to account for possible physical disk partitionings of the database, as discussed in Sections 8.2.9.1 and 8.2.9.2. These partitionings can be defined as properties and treated by TOPAZ in an analog way as the underlying system architecture, i.e. shared-nothing, shared-disk or hybrid (discussed in Section 4.4). Finally, the routines that convert MIDAS operator trees into logical operator trees as input for *Model-M* and *TOPAZ* and that convert the optimized, respectively parallelized physical trees back into a MIDAS operator tree have to be adjusted.

Hence, our parallelization strategy presented in the previous chapters can easily be extended to provide comprehensive and efficient parallelism for query execution plans that contain user-

defined constructs as well. However, please note the resulting performance is influenced in a significant way by the effectiveness of the applied estimation functions, that are especially needed for the determination of the $T_{lprocess}$ component. Providing accurate selectivity and execution cost estimates for user-defined extensions is still an open problem [Ja99] and is beyond the scope of this thesis.

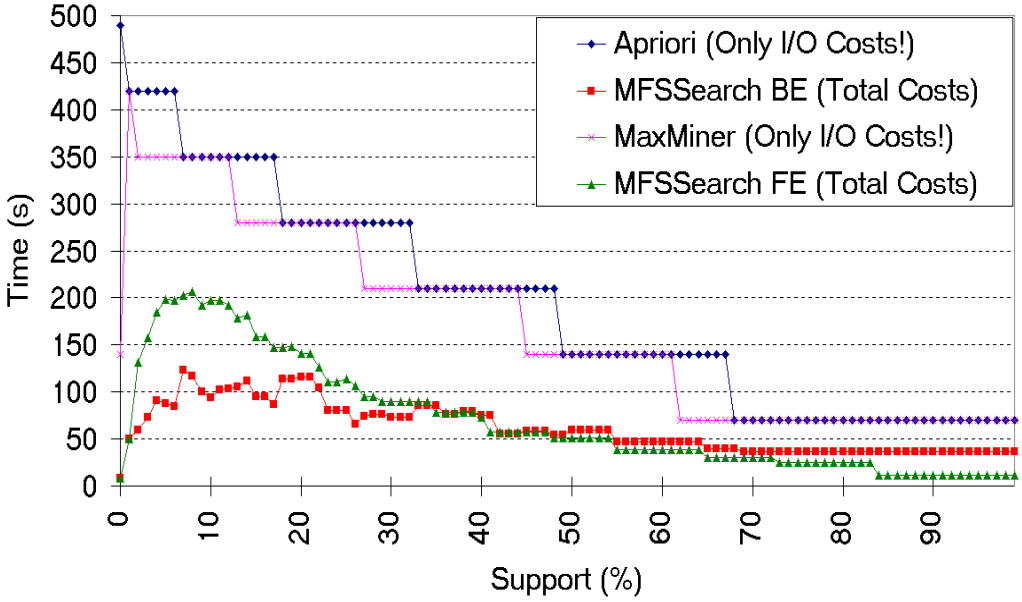
8.2.11 Performance evaluation

In order to evaluate the performance of our processing scheme *MFSSearch* for maximal frequent itemsets via the *StreamJoin* operator, we have integrated this operator into the MIDAS system. We have validated our approach using a 100 MB TPC-D database [TPC95], running on a SUN-ULTRA1 workstation with a 143 MHz Ultra Sparc processor. For the parallel scenarios, we used a cluster of up to 4 workstations. The database contains 150.000 transactions comprising orders on 20.000 different parts. For a detailed evaluation, it was important to consider a column having a limited value domain. Thus, we have performed our measurements on the LINEITEM table, where the place of the *item* column is taken over by *linenumber* and the pair *partkey, suppkey* is considered as being the *tid* attribute. The domain of the *linenumber* column is from 1 to 7, and the attributes *partkey, suppkey* define 67.806 transactions.

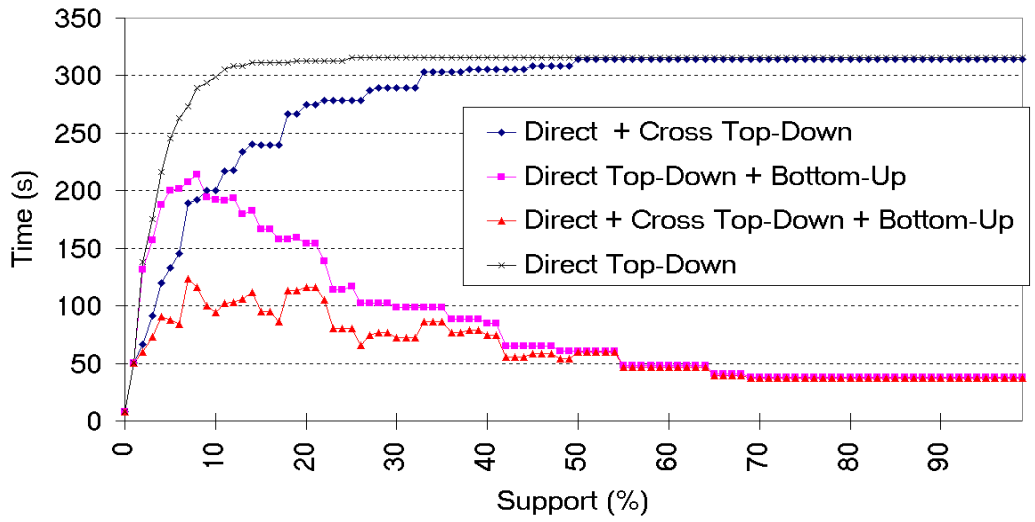
In the following we would like to point out the difference of this modeling to traditional market basket analysis. For simplification purposes, assume that the *linenumber* attribute represents some kind of timestamp: weekdays, months etc. In a traditional market basket analysis, if an itemset *{A, B}* is found frequent, a possible resulting rule might be: *"If a customer buys item A at a given time it is likely that he/she buys also item B"*. In contrast, a possible interpretation of a frequent itemset *{1, 2}* in our modeling is the following: *"If a part is sold at timestamp 1, it is likely that the same part will be sold at timestamp 2 as well"*. Hence, this kind of modeling is particularly suitable for e.g. event analysis.

In almost all top-down approaches item ordering plays an important role [Ba98, ZP+97]. The rationale is that items with high frequency are most likely to appear in frequent itemsets, thus increasing the effectiveness of top-down pruning. Therefore, items are ordered during a pre-processing phase in increasing order of their supports. We have evaluated the influence of item ordering on *MFSSearch* as well, obtaining also the maximum efficiency for the decreasing order of the item frequencies. Hence, in the following we report on the performance of *MFSSearch* using this ordering strategy.

In order to compare the performance of *MFSSearch* with the *Apriori* algorithm that is the basis of most bottom-up approaches, in Fig. 8.9a we have presented the time that is necessary to perform the multiple database scans specific to this algorithm. Please note that this curve doesn't comprise any CPU costs that are also inherent to the *Apriori* algorithm. As can be seen in Fig. 8.9a, *MFSSearch* shows both for the forward (FE) as well as for the backward evaluation (BE) a performance that is orders of magnitude better than the *Apriori* algorithm. At the same time, we have listed the I/O costs that would result from processing the items using *MaxMiner* [Ba98]. This roughly corresponds also to the I/O necessary for the *PincerSearch* algorithm [LK98]. As can be seen, although both approaches have proven to be more efficient than the



a)



b)

Fig 8.9: Effectiveness of pruning

Apriori algorithm in terms of CPU costs, the repetitive database scans still cause significant I/O costs. The results in [Ba98] also show that the increased efficiency of *MaxMiner* doesn't result primarily from the reduction of database passes, but from the consideration of less candidates. However, as can be seen in Fig. 8.9a, the resulting I/O costs of these algorithms already exceed the total costs of *MFSSearch.*

As results from Fig. 8.9a, BE is more efficient for lower supports. The reason for this is that in this case we have a large number of long MFIs that can be used for top-down pruning. However, in the FE scenario, CTDP is not possible. In contrast, in the domain of higher supports and

implicitly large number of infrequent itemsets BUP is most effective. In this case FE is slightly better than BE, resulting from the fact that in the BE scenario only prefixes can be used for BUP.

A detailed analysis on the effectiveness of the different pruning techniques for the BE scenario is given in Fig. 8.9b. Obviously, top-down pruning is most effective for lower supports, where large maximal frequent itemsets can prune several subsets, these being also frequent. Starting with a support of 20-25%, DTDP does not come to application at all. As for CTDP this point is reached with a support of ca. 50%. BUP is most effective with higher supports. The reason for this is that the higher the support, the more infrequent itemsets are found, that in turn can prune their supersets. The bottom curve in Fig. 8.9b shows that the best performance is achieved by the combination of all three pruning techniques. This leads to overall response times that are only proportional to the volume of maximal frequent itemsets.

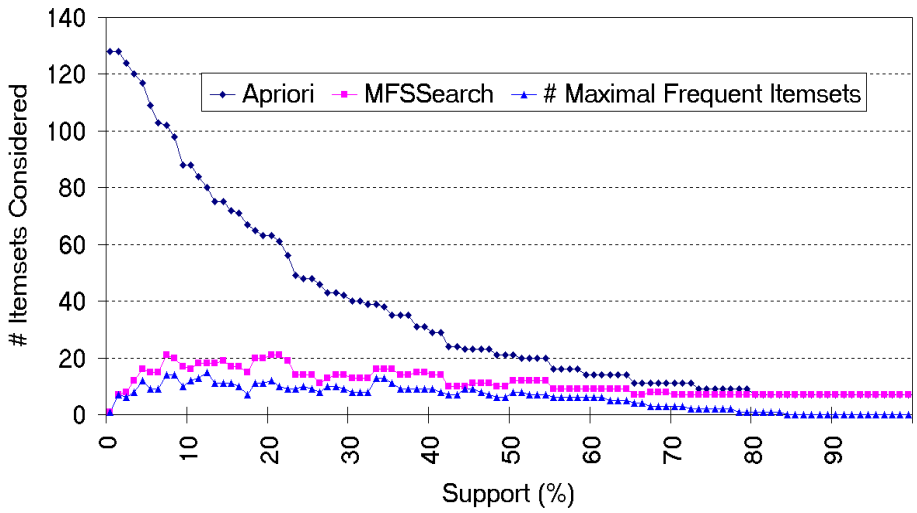
Given the above, we expect that the BE scenario achieves generally the best performance when integrated into the database engine. This results on one hand from the fact that it yields a non-blocking processing, hence rapid response times. On the other hand, this strategy combines both pruning strategies to achieve an efficient reduction of the search space for any support values. Hence, all further performance evaluation refers to the *MFSSearch* BE scenario.

The most striking difference between *Apriori* and *MFSSearch* is in the number of candidate itemsets considered to produce the set of maximal frequent itemsets. As can be seen in Fig. 8.10a, while in the *MFSSearch* processing scheme this number is proportional to the actual number of maximal frequent itemsets, the itemsets considered by the *Apriori* algorithm increases exponentially with decreasing supports. As shown in [LK98], the number of maximum frequent itemsets is a non-monotone function w.r.t. the minimal support. This result shows that, unlike most algorithms, *MFSSearch* can fully benefit from this property. Thus, it is also suitable for the exploration of large item domains.

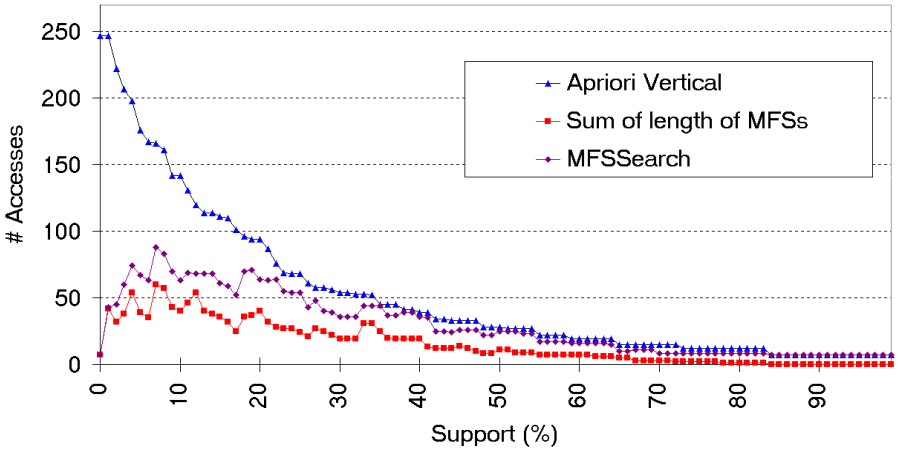
As mentioned in Section 8.2.1, in [STA98] the most promising improvement of the *Apriori* algorithm that is also suitable for database integration was found to be one based on a *Vertical* format. This format requires a preparatory phase that determines for each item a list comprising all *tids* that contain that item. Because of the variable length of these lists, in [STA98] they are stored in BLOBs. To compare this optimized variant of the *Apriori* algorithm with our approach, we have kept track of the data accesses necessary in each scenario. In this case, by a single data access we mean the access to all *tids* corresponding to a single item, i.e. in the implementation proposed by [STA98] reading in a single BLOB. In Fig. 8.10b, we have compared these numbers with the MFS volume, i.e. the sum of the lengths of all maximal frequent itemsets. As can be seen, the data accesses related to *MFSSearch* are proportional to this volume. In contrast, the data accesses needed for the *Vertical* approach increased exponentially with decreasing supports. Comparing Fig. 8.9a and Fig. 8.10b, we can also see that the complexity of *MFSSearch* scales linearly with the maximal frequent itemset volume.

As for parallelization, we have partitioned the LINEITEM table on *partkey, suppkey.* This corresponds to a partitioning in which all tuples belonging to a transaction reside on the same partition (cf. Section 8.2.9.1). We have used a degree of parallelism varying from 1 (sequential) to 4. The results presented in Fig. 8.10c show a linear speedup. This demonstrates that the data river concept presented in Chapter 3 provides efficient intra-query parallelism for plans containing user-defined constructs as well, thus validating the extensibility of the concept.

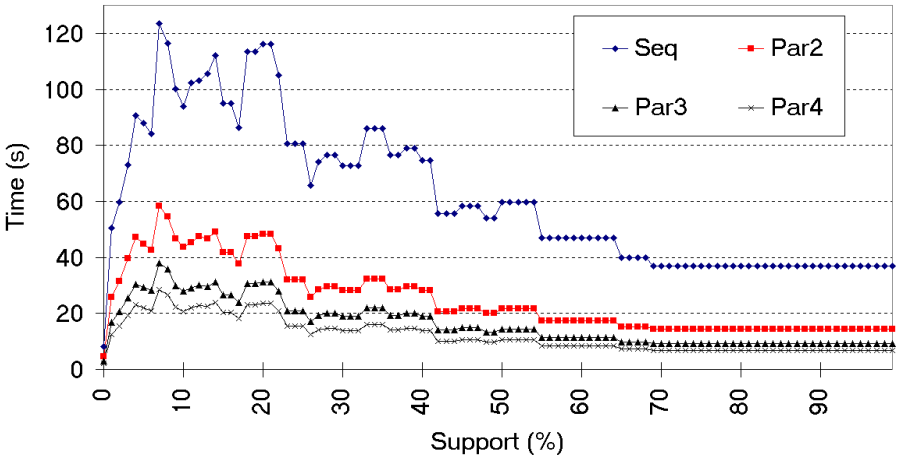
.



a) Candidate itemsets considered



b) Data accesses



c) Parallelization performance

Fig 8.10: Performance evaluation for MFS calculation

8.2.12 Summary

In this section we have presented a processing scheme for the generation of maximal frequent itemsets that employs a new operator, called *StreamJoin* that is combined with a highly effective search strategy. This scheme meets all requirements posed to modern data mining approaches:

- The data accesses are kept minimal, only being proportional to the sum of the lengths of the maximal frequent itemsets (called the volume of maximal frequent itemsets).
- Efficient pruning can be applied. Only itemsets that are neither supersets of any known infrequent itemsets nor subsets of any known frequent itemsets are considered. As a result, the number of candidate itemsets considered is proportional to the actual number of MFIs. Hence, the algorithm is also applicable for large item domains.
- The complexity of the algorithm scales linearly in the volume of maximal frequent itemsets. The measurement results demonstrate that our approach can fully benefit of the non-monotone property of the number of maximal frequent itemsets w.r.t. minimal support.
- Non-blocking execution and short response times are achieved, especially for the first result tuples. This is due to the fact that by using our candidate generation algorithm with a backward exploration of itemsets, any frequent itemset found is also a MFI.
- The processing scheme can be efficiently integrated with a database engine, thus being able to make profit of all forms of query execution optimizations, including parallelization. Moreover, unlike most related work where itemset generation is still performed by multiple phases, involving also insertions into intermediate tables, our approach yields a uniform processing within a single query execution plan, without any additional disk spoolings, blocking of intermediate results, or preparatory phases.

Compared to previous work, our measurements indicate that *MFSSearch* has a better performance than the *Apriori*-like bottom-up approaches. Compared to most top-down approaches, our strategy avoids multiple database passes and is truly independent of the length of the longest MFI. Thus, the *MFSSearch* strategy is applicable also for (ad-hoc) mining in large-scale DBMSs, such as data warehouses, as well as for hybrid solutions, where the considered item domain is restricted by means of e.g. sampling [FS+98].

MFSSearch represents a processing primitive for the calculation of the maximal frequent set, thus facilitating the modularization of similar problems, e.g. pattern recognition, and the reuse of this primitive. We have presented a solution towards deeply integrating *MFSSearch* with the DBMS engine. Furthermore, we have shown that at this level intra-query parallelism is applicable in a straightforward way using the concepts presented in the previous chapters, resulting into linear speedup and thus increased efficiency. This shows the extensibility and effectiveness of our approach for complex applications as well, holding also user-defined constructs.

Finally, our implementation concept based on user-defined table operators and user-defined functions is available on the (SQL) language level, thus permitting an easy reuse of implementation as well.

8.3 Applicability of StreamJoin for Universal Quantification

Quantified queries become increasingly important in forthcoming applications, such as DSS, OLAP etc. However, relational database systems do not adequately support such queries. Effective support is needed at both the language level as well as in the underlying query processing system. As far as the first issue is concerned, quantified queries are expressed in SQL using the GROUP BY, (NOT) EXISTS and (NOT) IN clauses, as well as counting.

We first consider the example given in [GC95], a university database with two relations, COURSE(*course-no, title*) and TRANSCRIPT(*student-id, course-no,...*). The goal is to find the students who have taken all courses offered by the university. Two examples of expressing this query in SQL are presented below.
Example 8.7:

```
SELECT DISTINCT t1.student-id FROM TRANSCRIPT t1
WHERE NOT EXISTS
   (SELECT * FROM COURSE C
   WHERE NOT EXISTS(
      SELECT * FROM TRANSCRIPT t2
      WHERE t2.student-id = t1.student-id
      AND t2.course-no = c.course-no))
```

Example 8.8:

```
SELECT t.student-id FROM TRANSCRIPT t
GROUP BY t.student-id
HAVING COUNT (t.course-no) = (SELECT COUNT (course-no) FROM COURSE)
```

Obviously, these formulations are not natural and in addition difficult to optimize. Language extensions have already been proposed in the literature [HP95, RBG96] and are being considered as additional predicates in the upcoming SQL99 standardization as well [SQL99]. In this section, we concentrate on the support provided by relational query processors.

8.3.1 Related work

[CK+97] presents a comprehensive treatment of universal quantification for object-oriented and object-relational databases from the query level to the evaluation. According to this analysis, plans implementing the all-quantification with an anti-semijoin are superior to all other alternatives. However, as we will present in this section, this approach is only applicable in an object-oriented model. Thus it is still an open problem how to deal with universal quantifiers in OLAP or DSS applications, that are mostly based on a relational star or snowflake schema. Because of high data volumes, especially in these environments effective support is needed, and data reorganization has to be avoided.

Universal quantification is evaluated in [GC95] by a hash-based division algorithm. However, as shown in [CK+97] and later on in this section, this approach applies only for a special class of queries, namely those for which the quantifier's range constitutes a closed formula. In [Da87] generalized join and aggregation operators are presented. However, the scope of the paper is restricted only to traditional aggregation operations over groups, such as *average*, *max*, *count* etc., and is not applicable for all-quantification. [RBG96] propose a generalized quantifier

framework that defines a completely new query subsystem. Thus, it requires significant changes to the query execution system, as special indexes and multi-dimensional structures have to be built for all relations. Moreover, since only small test databases have been used, the results are not directly applicable for large-scale applications, such as e.g. OLAP.

In the following, in order to assess the applicability of the different approaches, we will consider two example scenarios employing universal quantification.

8.3.2 Example Scenario 1: The University Database

We first consider the university database example given in [GC95]. As already mentioned, this database contains two relations, COURSE(*course-no, title*) and TRANSCRIPT(*student-id, course-no,...*).

8.3.2.1 The *Hash-division* Algorithm

In [GC95] the TRANSCRIPT relation is called the *dividend*, the COURSE table the *divisor and* the division result is called *quotient*. The hash-division algorithm uses two hash tables, one for the divisor and one for the quotient. For each tuple in the quotient table a bitmap is kept with one bit for each divisor tuple. First, all divisor tuples are inserted into the divisor table. Next, the algorithm consumes the dividend relation. If a matching tuple is found in the divisor table, the dividend tuple is either newly inserted as a candidate into the quotient table, or, if it is already present, the associated bitmap is modified by turning the bit corresponding to the matching divisor tuple to 1. When all tuples of the dividend relation are consumed, the quotient consists of those tuples in the quotient table for which the corresponding bitmap contains no zeros.

8.3.2.2 The *Anti-semijoin* Approach

As described in [CK+97], in an object-oriented model the *N:M* relationship *enrolled* between students and courses is typically modeled by a set-valued attribute *enrolledCourses* for each student. Thus the all-quantification can be resolved by an anti-semijoin on the 2 tables TRANSCRIPT and COURSE using the condition $course-no \notin enrolledCourses$. The anti-semijoin adds to the output stream only those tuples, i.e. students, for which no join partner has been found. This is equivalent to the fact that all courses are included into the *enrolledCourses* attribute of the given student item, hence the student has attended all courses.

However, in a relational schema this approach cannot be applied. Obviously, the anti-semijoin on TRANSCRIPT and COURSE, using the condition $course-no \neq course-no$, yields as a result an empty set. Thus, the anti-semijoin strategy cannot be applied for the evaluation of universal quantification in e.g. data warehouses that adopt a relational star or snowflake schema.

8.3.2.3 The *StreamJoin* Approach

Fig. 8.11 presents the QEP for the evaluation of the all-quantification via the *StreamJoin* operator. By joining the 2 tables TRANSCRIPT and COURSE using e.g. an index on *course-no* for the TRANSCRIPT table, the intermediate result *IR* consists of separate streams for each course, containing the students that have taken that course. These streams are called in the example from

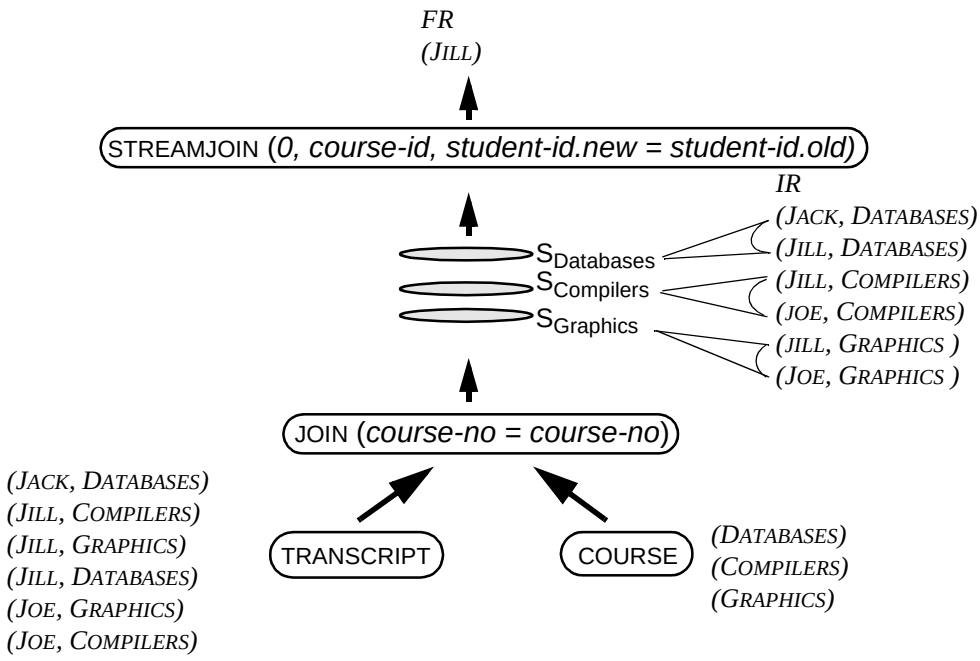


Fig 8.11: Evaluation of universal quantification with the *StreamJoin* approach

Fig. 8.11 $S_{Databases}$, $S_{Compilers}$ and $S_{Graphics}$. In order to obtain the students that have participated in all courses, a join of these streams on the *student-id* attribute is necessary. This operation is performed by the *StreamJoin* operator, yielding for the example from Fig. 8.11 one tuple for the final result *FR*. Please note that the parameter corresponding to the *Group-ID* is set to 0 (or any arbitrary constant value), as there is only one divisor set to be tested.

As compared to the hash-division algorithm, this approach needs less buffer space, because it has as an upper bound the size of the first stream (in this example the 2 tuples of the stream $S_{Databases}$). As shown in Section 8.2.3, the intermediate result (i.e. hash table) sizes decrease with each iteration, as the tuples which don't match the join condition are eliminated. In contrast, the hash division algorithm has to keep the whole divisor in memory, i.e. the COURSE table (3 tuples), together with all quotient candidates and the corresponding bitmaps (3 tuples + bitmaps). This results for this small example to altogether 6 tuples + 3 bitmaps that have to be kept permanently in memory for the hash division algorithm, compared to maximal 2 tuples that are kept simultaneously in memory for the *StreamJoin* approach.

8.3.3 Example Scenario 2: Data Warehouse

Consider a data warehouse with a central FACTS table, describing the sales in a given time period, and several dimension tables, e.g. NATION being one of them:

FACTS(*partkey, suppkey, nationkey,...*)
NATION(*regionkey, nationkey, nationname,...*) ...

Suppose that for marketing purposes, a user is interested in the following query:

Find the suppliers who supply a part that is being sold in all nations of a region.

In the following, we analyze how this query can be solved by the various approaches.

8.3.3.1 The *Hash-division* Algorithm

In this case, the divisor table is NATION, or, more precisely, it is constituted of several parts of this table, each part corresponding to a region. It results already from this aspect that the hash-division algorithm cannot be applied in a straightforward way, as it has to be extended to keep separate bitmaps for every divisor, i.e. every region. In addition, each <*partkey, suppkey*> combination has to be considered as a possible quotient candidate. Thus, the quotient table is constituted of all such combinations, each combination having in addition several bitmaps corresponding to the different regions. It results already from these memory requirements that the algorithm is not competitive for this case.

8.3.3.2 The *Anti-semijoin* Approach

Similar to Section 8.3.3.2, the approach is not applicable for this relational schema, since the anti-semijoin between the 2 tables on the *nationkey* attributes yields an empty set.

8.3.3.3 The *StreamJoin* Approach

Since *nationkey* is one of the dimensions of the central FACTS table, suppose for simplicity purposes that there exists an index on this attribute. We further assume that the NATION table is sorted on the *regionkey* attribute. Please note that if these conditions are not fulfilled by the physical databases design, they can be accomplished by corresponding operators. By joining the NATION and FACTS table, each region defines a group, containing the parts that have been sold in that region. In the example from Fig. 8.12 the groups are defined by the attributes *AFRICA* and *AMERICA*. Each group contains several streams, corresponding to the different countries of that region. For instance, in Fig. 8.12 the group *AFRICA* is constituted of the streams *ALGERIA* and *KENYA*. This intermediate result constitutes the input for the *StreamJoin* operator. Thus the *Group-ID* parameter of the *StreamJoin* operator as defined in Section 8.2.3 corresponds to the *regionkey* attribute, while the *Stream-ID* is set to *nationkey*. If we join the streams on the *suppkey* attribute, we obtain the suppliers whose different parts are sold in all countries of a region. However, the query contains as an additional constraint that it must be the *same* part that is sold

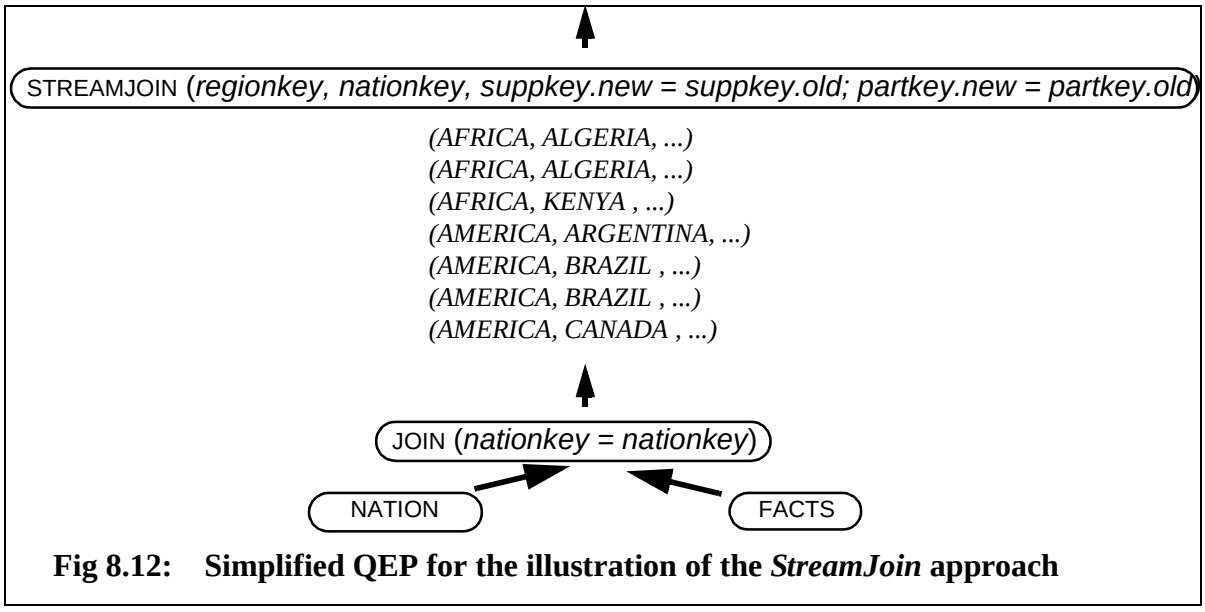


Fig 8.12: Simplified QEP for the illustration of the *StreamJoin* approach

in all countries. Thus, the join is defined on two attributes, namely *suppkey* and *partkey.* For both attributes, the join condition is equality, expressed by *suppkey.old = suppkey.new*, respectively *partkey.old = partkey.new.*

This example shows that the proposed approach solves even complex queries involving universal quantification, yielding a uniform database processing with low memory consumption. Intermediate result materializations are not necessary, either. This results in reduced I/O costs as well. Finally, savings in communication costs result from the fact that only the final result tuples need to be transmitted back to the application.

8.3.4 Performance Evaluation

For the performance evaluation, we have used the same 100 MB TPC-D database as in Section 8.2.11. For this database, we have evaluated different possibilities of evaluating the query presented in Section 8.3.3, namely *"Find the suppliers who supply a part that is being sold in all nations of a region"*.

We first compared the memory requirements of the *StreamJoin* and hash division approaches, on condition that the latter is extended to handle also complex divisors, by e.g. keeping several bitmaps for each quotient candidate. In this database the number of distinct <*partkey, suppkey*> combinations is 79.947. As mentioned, each such combination has to keep a separate bitmap corresponding to each region. In the TPC-D database, the number of regions is 5, each region being constituted of 5 nations. Thus, the memory requirement for the quotient table of the hash division algorithm is as follows:

79.947 (tuples) x 16 bits (*partkey, suppkey* attributes) x
5 (bitmaps corresponding to each *region*) x 5 bits (*nations* per *region*) = 3,2 MB.

This table is permanently needed by the algorithm, thus, if it doesn't fit into memory, both divisor and quotient tables have to be partitioned, resulting into considerable disk I/O costs.Please remind that the size of the hash tables used by the *StreamJoin* approach decreases in each step. During the measurements, the memory consumption of this algorithm averaged to ca. 280 KB. The peek of the memory consumption has been 700 KB. However, this has been measured only for a small time period, corresponding to a single stream.

The hash division algorithm has to consume all of its input before it produces the first output tuple. An additional advantage of the *StreamJoin* algorithm is that it forwards the result tuples region by region, thus making this approach also more attractive for pipelining.

As presented in Section 8.3.3, the anti-semijoin approach is not suitable for this scenario. The other possibilities in [CK+97] for evaluating the all-quantificator are based on counting or set difference. Hence, we compared the *StreamJoin* evaluation with these two strategies as well.

In Fig. 8.13 different variants of the *StreamJoin* approach are shown, all using a DOP of 4. Please note that the schema of the TPC-D database is more complicated than the classic star schema. Since the LINEITEM table doesn't contain the *nationkey*, this attribute has to be derived by joining the *region*, *nation*, *customer* and *(new-)order* tables, the final join with the central LINEITEM table being done on the *orderkey* attribute.

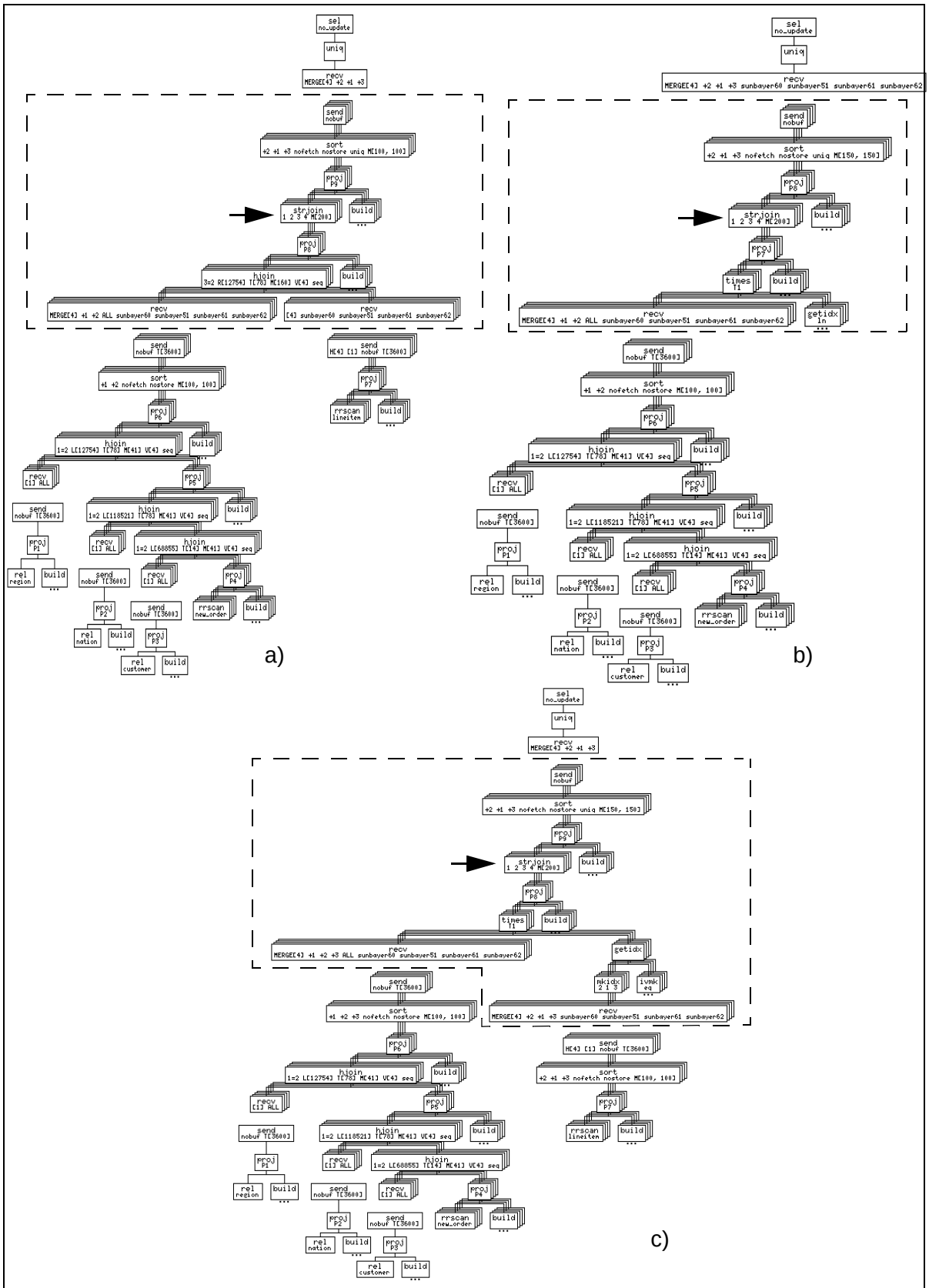


Fig 8.13: Query execution plans using the *StreamJoin* approach

The variants shown in Fig. 8.13 are identical in the way they employ the *StreamJoin* operator (marked by dark arrows), but differ in the way they perform the join with the LINEITEM table. As mentioned before, this has to be done in a way that guarantees the correct ordering for the *StreamJoin* operator. This grouping has to be done on *Group-ID* that in this example is identical with the *regionkey* attribute, and *Stream-ID*, which is set to *nationkey* (as explained Section 8.3.3.3). The grouping requirement can of course always be accomplished by *sort* operators. However, as shown by these examples, this is not always necessary, since the required ordering can also be accomplished by using an adequate algorithm for the operators delivering the *StreamJoin* input. Thus Variant a) uses a hash join and full table scan on LINEITEM. Variant b) employs an nested loops index join using a previously built index of the LINEITEM table on the *orderkey* attribute. Additionally, in variant c) this index is built on the fly. Thus, sorting can be entirely avoided if such possibilities are taken into account when optimizing queries that contain a *StreamJoin* operator.

The PQEPs shown in Fig. 8.13 also demonstrate the various parallelization possibilities of a query execution plan containing the *StreamJoin* operator. Please note that the most convenient variant for the given database configuration can be generated by a flexible, cost-based and rule-driven parallelizer, such as TOPAZ, that accounts also for the existing physical properties of the database, such as indexes, partitioning strategies etc.

In Fig. 8.14 we present the results of our performance evaluation for the approach based on counting as well as for the *StreamJoin* variants. The approach based on set difference took more than 10 hours for only one region, hence it is obviously not competitive. As already mentioned, the queries have been performed sequentially, as well as in parallel, using different degrees of parallelism. When employing parallelism, the part of the query that is in charge of deriving the *orderkey* attribute for the join with the LINEITEM table has been kept constant, while modifying the DOP of the relevant subplan, e.g. for the *StreamJoin* variants the subplan containing the *StreamJoin* operator. In the plans from Fig. 8.13, the subplans whose DOP has been changed is marked by a bounding box. In order to asses the speedups correctly, the constant costs of the base subplans have to be considered as well. Thus, they have been listed separately in Fig. 8.14.

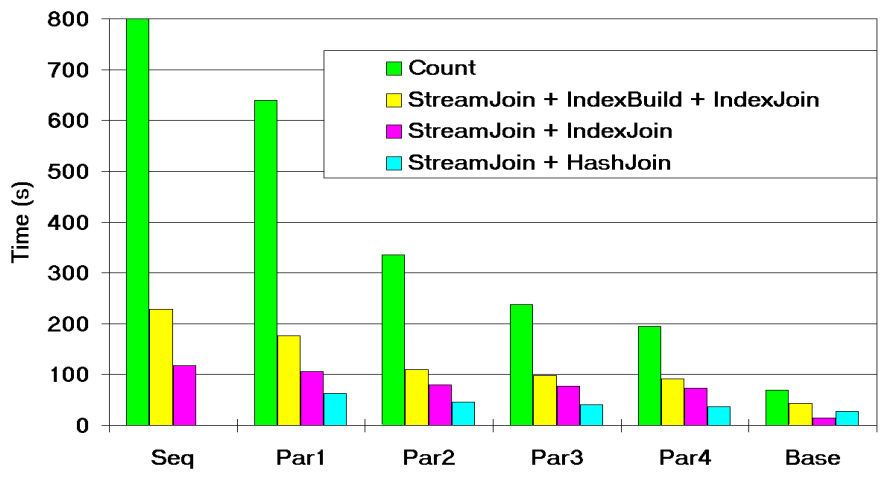


Fig 8.14: Performance evaluation

The performance evaluation shows that the *StreamJoin* approach outperforms the variant based on counting by factors, even if the index of the LINEITEM table is built on the fly. The best results have been achieved for the hash join variant, as this approach allows a uniform evaluation with minimal I/O costs after the hash table has been built. However, this variant can only be used if the database cache is large enough to hold the tables of both the hash join as well as *StreamJoin* operators. As can be seen, the only situation where this requirement couldn't be accomplished was the sequential case, where the whole QEP is evaluated on the same processing site.

As shown in Fig. 8.14, after substracting the constant base costs, quasi-linear speedups have been achieved. This demonstrates that our approach for achieving efficient intra-query parallelism is effective also in case of complex query execution plans as those presented in Fig. 8.13, holding user-defined extensions as well.

8.4 Sequences

Recently, patterns and sequences, especially time sequences appear in various application domains. Typical examples are scientific experiments such as temperature reading generated by sensors, business applications such as stock price indexes or bank account histories and medical data such as cardiology data. Sequence processing is a challenging task for data mining purposes as well [Za98]. The corresponding (temporal) databases tend to be voluminous, thus forcing specific algorithms to reduce processing overhead, such as communication costs etc. However, related work treats sequence analysis mostly on top of a database. In contrast, the *StreamJoin* operator can significantly contribute towards performing sequence processing in an integrated fashion. This will be demonstrated in the following using as an example an application operating on financial data.

One interesting scenario in finance is to identify pairs of stocks whose prices track one another [MC98]. Suppose a database for stock analysis containing the following table:

STOCKINFO(*week, day, stockkey, price*).

Without loss of generality, we assume that this table is sorted on the *week* and *day* attributes. Hence, the query *"Which stocks have had continuously rising prices during an entire week?"* can be simply evaluated by a scan of the STOCKINFO table followed by the *StreamJoin* operator. The corresponding signature is expressed as follows:

StreamJoin (week, day, stockkey.new = stockkey.old; price.new > price.old).

Hence, the *Group-ID* is set to *week*, the *Stream-ID* set to *day* and the join is performed on the *stockkey* and *price* attributes. In this way only those tuples of a new stream, i.e. a new day, qualify, that satisfy the condition that for the same *stockkey* (*stockkey.new = stockkey.old)* the price is raising (*price.new > price.old).* The result of this simple execution plan are tuples of the form *(stockkey, week),* representing the stocks that have had continuously rising prices during an entire week.

Assume that the user is further interested in stocks that are chasing one another. In a simplified

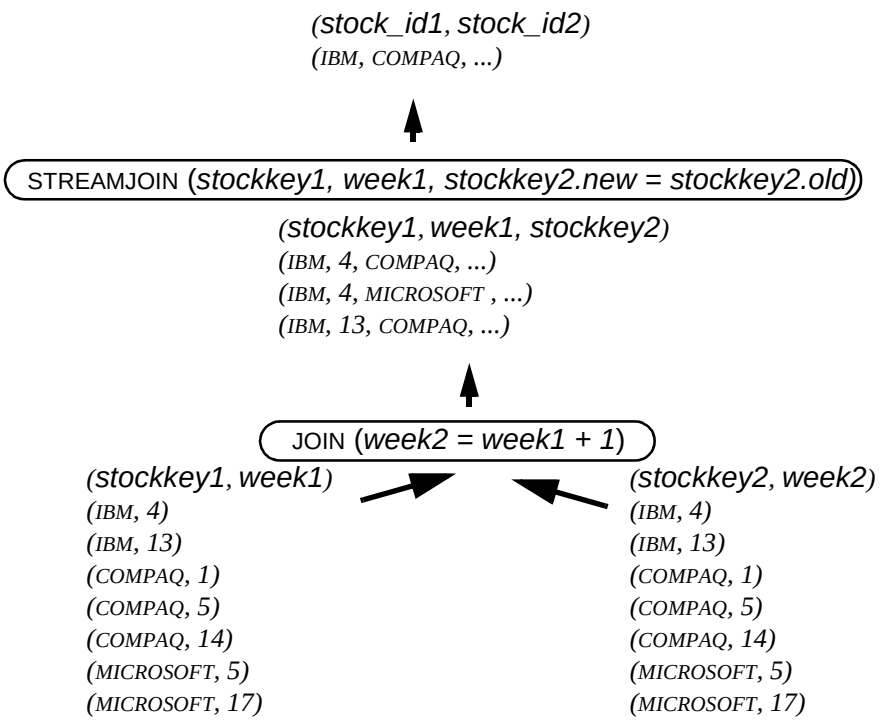


Fig 8.15: QEP for the usage of the *StreamJoin* operator in financial time series

model, this can be defined by the fact that every time the price of the first stock is rising during an entire week, the price of the second stock is also rising during the subsequent week as well. Thus, the query can be answered by a self join followed by the *StreamJoin* operator as shown by the QEP depicted in Fig. 8.15. The self join is on condition that the attribute value *week2* is subsequent to *week1,* given by the expression *week2 = week1 + 1*. The intermediate result *(stockkey1, week1, stockkey2)* expresses pairs of stocks that chase one another in 2 subsequent weeks. However, the condition for chasing stock prices is that *every time* the first stock is rising, the second is also rising in the subsequent week. This condition is evaluated by the *StreamJoin* operator. The input is delivered by the intermediate result described above, by setting the parameters *Group-ID* to *stockkey1*, *Stream-ID* to *week1* and the join predicate is defined on *stockkey2.*

In Fig. 8.15, we have depicted the group defined by the stock *IBM*. This group is constituted of two streams, corresponding to the two weeks (*4* and *13*) with rising prices for this stock. By joining the streams on *stockkey2*, we obtain the stocks that chase *IBM* in *all* subsequent weeks. In this example, the price of *Compaq* is rising every time when the price of *IBM* is rising, while *Microsoft* is chasing *IBM* only in the 4th week, thus it is eliminated by *StreamJoin* for the final result.

This example shows once again that the *StreamJoin* operator interfaces in a natural way the other query processing capabilities of the database engine, thus being able to make use also of all existing facilities and optimizations, such as indexes, sorting etc., including intra-query parallelism.

8.5 Pattern Discovery in Genomic Databases

Genomic databases assist molecular biologists in understanding the biochemical function, chemical structure, and evolutionary history of organisms. Popular systems for searching genomic databases match queries to answers by comparing a query to each of the sequences in the database. Efficiency in such exhaustive systems is crucial, since some servers process over 40,000 queries per day [BLO93], and several queries require comparison to over one gigabyte of genomic sequence data. A genomic database contains sequence records that are continuous strings drawn from a specific alphabet, varying from a few characters to several hundred thousand characters in length. During each query task, a new string, further called *pattern*, has to be matched against the old strings. Thereby, the strategy must be able to find statistically significant similarity in the presence of not only varying sequence lengths, but also repetitive subsequences. Contrary to most related work [WZ96, TFT99], our approach to discovering patterns in a database of genetic sequences is realized within the database engine.

We consider the patterns as being regular expressions of the form $*X_1*X_2*...$, where X_1, X_2 are segments of a sequence made up of consecutive letters, and * represents a variable length of intermediate letters. We treat the maximal value of this variable length as a parameter, called *int_length*. The user is interested in the locations (positions) where a pattern is contained in a given sequence. Suppose that the information corresponding to sequences is stored in a table *SEQUENCE(pos, letter)* and the pattern is stored in a table *PATTERN(letter)*. Please note that this is only a simplified representation. Thus if e.g. there are several patterns to be analyzed, the *PATTERN* table has to be extended by a *patternkey* attribute. Similarly, in the general case, the *SEQUENCE* table contains a *sequencekey* attribute as well. Assume that in this example the pattern has the form *A*B*C*, with *int_length* being 1, i.e. one intermediate letter is allowed for matching subsequences.

This query can be evaluated in an integrated fashion by using the *StreamJoin* operator as shown in Fig. 8.16. In this case each element of the pattern defines a stream, containing all positions of the sequence where this element occurs. As already mentioned, we are interested in the portions

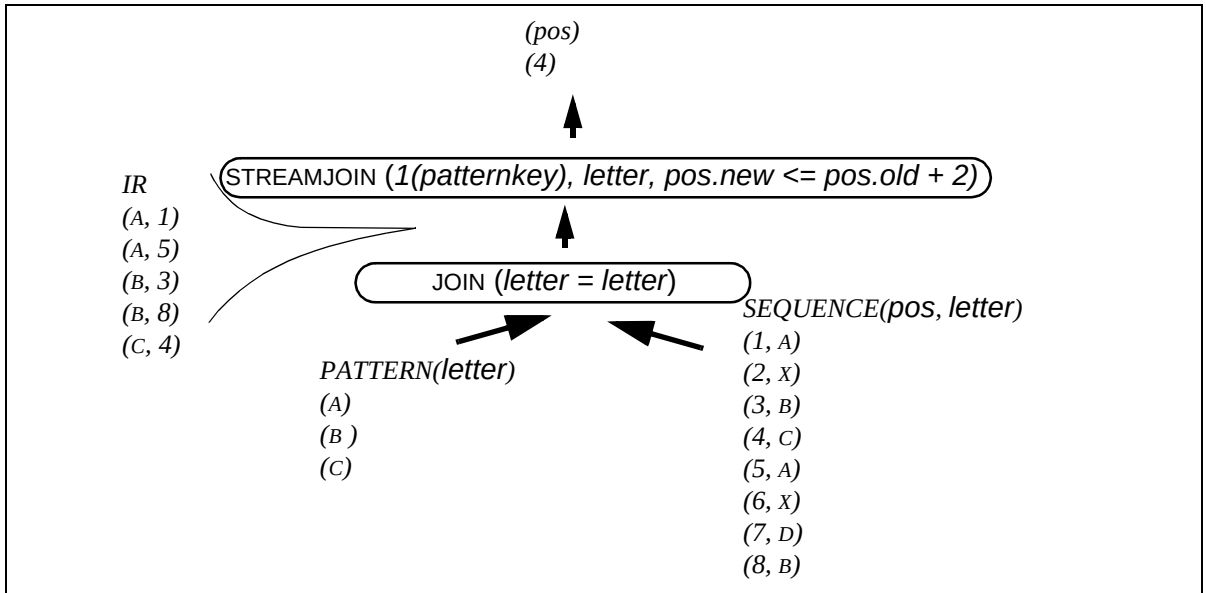


Fig 8.16: QEP for the usage of the *StreamJoin* operator in pattern recognition

of the sequence that contain *all* elements of the pattern *in the given order with the imposed position requirements*. Hence, the join condition for the StreamJoin operator is on the *pos* attribute, expressed by the following parameter:

pos.new <= pos.old + int_length + 1.

In the example from Fig. 8.16, by substituting *int_length* with *1*, we obtain:

pos.new <= pos.old + 2.

The *Group-ID* parameter of the *StreamJoin* operator is set to *1* (or any constant value), since a group is defined by a pattern sequence and in this simple example there is only one pattern involved. In the generalized case the *Group-ID* is set to the *patternkey* attribute. The *Stream-ID* is set to the *letter* attribute. In the example, we have three streams, corresponding to the letters *A, B* and *C* of the pattern. As mentioned, only the sequences satisfying the imposed position requirements qualify. For instance, in the example from Fig. 8.16, the tuples *(A, 5)* and *(B, 8)* of the intermediate result *IR* don't survive, since the distance between the letters is greater than 1. The result contains positions in the sequence that mark the end of a location where the pattern is integrally included. In this example, the pattern is included in the *SEQUENCE* table from position *1* to *4*, hence the result contains the position *4*.

The presented examples illustrate that the *StreamJoin* operator can be employed to solve queries concerning sequences within the database as well. The resulting plans guarantee a natural and uniform data flow among the operators involved. Moreover, no data transfer between the database and the application is necessary. Taking into account the volume of the involved applications, this aspect alone already results in substantial performance improvements.

Moreover, the capabilities of the database engine to provide efficient intra-query parallelism can be fully applied. The resulting parallel plan is similar to those presented in Section 8.3, hence we have not depicted it here separately.

8.6 Applicability of StreamJoin in Profiling Services

Nowadays enormous quantities of data are being accumulated by both the commercial and the scientific communities and thus use and spread of digital libraries enhances drastically. Additionally, an increasing number of users accessing these databases through web interfaces poses even harder performance requirements on the digital libraries; in these environments system response times of hours or even days for sequential evaluation have been reported despite of well-optimized execution plans.

For some years now, we experienced all this, while employing our *O*MNIS *D*ocument Management *S*ystem (short ODS) in various application scenarios, especially library applications. ODS has been developed at our university to administer all kinds of scientific libraries with documents in paper form or electronic form ranging from books and journals to technical reports and articles from newsgroups. ODS is developed as a layered system divided into three major components. The *service layer* comprises system services for archiving, retrieval, lending, as well

as profiling. The *full-text and document management layer* uses the *database layer* to manage the documents given in several representations and stored in multiple formats. Broad accessibility is guaranteed by support of different platforms and operating systems as well as a WWW gateway. More information on ODS can be found in [CV+95], [BO+95].

8.6.1 The Boolean Retrieval Model

ODS provides a boolean retrieval model answering queries on the existence of word patterns, words, phrases, and boolean expressions of them in documents. Syntax and semantics of the model can be informally described as follows:

<word>	*Exact match of <word>*
.	*Distance operator, used to combine terms to phrases*
&	*Boolean AND of phrases*
|	*Boolean OR of phrases.*
%	*Wild card character usable in a <word>, allowing to specify simple pattern matching*

A *term* is defined as a word or a word pattern. Terms connected by distance operators form *phrases*. This simple model defines the basis of the retrieval service, as for example, the phrase “deduct% . database” asks for all documents in which there is an occurrence of a word starting with the string ‘deduct’ together with a word ‘database’ and having at most one other arbitrary word in between as indicated by the distance operator.

8.6.2 The Profiling Service

Basically, a profile represents the user’s reading interests and can be expressed as a single full-text retrieval query in ODS. The profiling service is the (batch) execution of a large set of profiles, which can be launched in fixed time intervals, for example once per night. A sample profile, also used in the following, may look like this:

‘deduct%. database | multimedia database’)

Please observe that there are some noteworthy differences between complex retrieval queries and profiles:

- Profiles aren’t interactive but batched, which relieves execution timing requirements and thereby offers more optimization possibilities; note that for interactive queries there is no other possibility than optimizing them one at a time.
- Profiling provides opportunities for multi-query optimization, i.e. to process a set of queries in an integrated, tightly coupled and optimized fashion.
- Profiles tend to be stable over some period of time so that they are launched in the same form many times against the same database (holding more and more documents). Again, this advocates for rigorous and perhaps even exhaustive optimization, since optimization time is separate from runtime; thus optimization efforts pay off by repeated execution.

8.6.3 Mapping to the Database Layer

In the following we provide some insight into the mapping of the document retrieval queries as well as the profiles and profile processing to the database layer.

8.6.3.1 Mapping of Full-Text Retrieval

As already mentioned before, our document retrieval model is based on full-text indexing that, applied to ODS, refers to plain relational tables, which are organized in such a way that document archiving and especially document retrieval can be done quite efficiently. There are two important tables, a table *WORDS* (*word*, *word-ID*), which maps each word which appears in at least one document to a word number *word-ID*, and a table *DOCUMENTS* (*word-ID, Doc-ID, Pos*), which maps word numbers to document numbers *Doc-ID* and position numbers *Pos* depending on where the word appears in the documents; primary keys are underlined. To give an idea on how an ODS query looks at the database level, we have shown in Fig. 8.17 the query execution plan (QEP) generated by the optimizer for a sample query. This QEP be easily understood by simply applying the information previously given on our full-text retrieval model, its database representation, as well as the common knowledge on SQL and its query processing. Please note that a detailed analysis of this query type can be found in [BJ+96].

8.6.3.2 Mapping of Profiles

Looking at ODS users' behavior, we expect on the average 15 to 20 search phrases per profile, and for example 2800 profiles if we want to serve all students and staff members of our computer science department. In addition, we register an increasing amount of external users access-

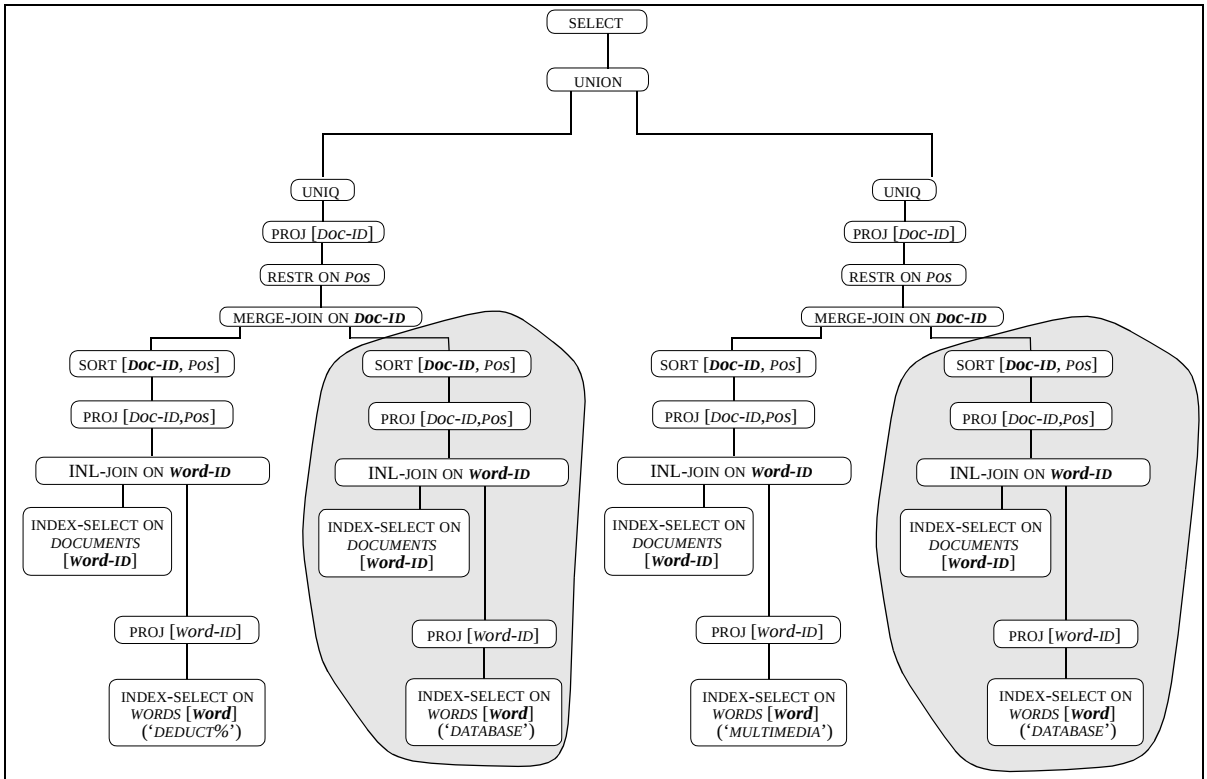


Fig 8.17: The structure of an ODS full-text retrieval QEP ('deduct% . database | multimedia database')

ing the system through the WWW interface, who might also be interested in the new service. Considering the query from Fig. 8.17, which can be seen as a very simple profile containing only 4 words, we can get an idea of the complexity of plans covering regular profiles. More accurately, the query complexity (in terms of number of joins, unions and intersections) is linear in the number of terms in the query predicate. In the average, we have to cope with about 200 unary and 80 binary operations per profile. At the same time, there is a great redundancy both within query plans as well as between them (see the shaded QEP areas for term 'database' in Fig. 8.17 and the discussion below).

In order to cope with this query complexity and to maximize overall throughput, we can conceive two solutions, both quite attractive to us:

- *Multiple Query Optimization (MQO)*
  Some related work can be found in [Se88] and [RC88], presenting algorithms for identification of structurally identical subqueries and elimination of common sub-expressions. Unfortunately, the techniques presented are not applicable for large queries.
- *Parallel Evaluation*
  We can achieve some speedup by using parallel evaluation strategies for single ODS queries according to the concepts presented in the previous chapters. Nevertheless, if we continue processing these queries one at a time, performance needs will not be met: in addition to the processing costs, there is considerable overhead due to (often redundant) plan generation and startup costs.

Hence, our overall strategy has to be a combination of both parallel query evaluation and multiple query optimization. In doing so, we extend the DBMS with a *profiling* component, which accepts as input not only a single profile in the form of a full-text query, but a set of such profiles, i.e., a set of queries. The aim is to combine and to process *all* evaluation phases for *all* profiles on *all* processing nodes in parallel.

Maximizing throughput using batch scheduling in parallel database systems is also dealt with in [MSD93], but only single join queries were discussed. In [TL96] algorithms have been proposed that merge single query plans to produce a pipelined multi-query plan, i.e. separate plan generation is still the case.

In Section 8.6.3.3 we show how separate complex but similar queries can be represented by *one* common QEP, thus reducing both the overall number of operations (by means of elimination of common query fragments) and QEP generation time. In Section 8.6.3.4 we analyze profile evaluation and its inherent parallelization potential.

8.6.3.3 Multiple Query Optimization and Normalized Representation of Profiles

MQO for profiles results into a so-called normalized profile set that is reflected by our internal representation of profiles. This normalized representation consists of three tables as depicted in Fig. 8.18. Each primitive term, i.e. each word or word pattern, is assigned a term identifier *(Term-ID)* and stored in the *TERMS* table; for efficient processing there is an index on the *Term* column additionally providing a lexicographic order. Terms connected by distance operators and boolean AND operators form *subprofiles*. In our example, there are 3 subprofiles. As the

Table *TERMS*

Term	Term-ID
architect%	6
data%	4
database	2
deduct%	1
multimedia	3
system	5

Table *SUBPROFILES*

SubPro-ID	Term-ID	Distance	ProfilePos
1	1	1	1
1	2	-2	2
2	3	0	1
2	2	-2	2
3	4	0	1
3	5	0	2
3	6	-2	3

Table *PROFILES*

Profile-ID	SubPro-ID
1	1
1	2
2	2
2	3

Fig 8.18: Normalized profile representation for two sample profiles primary keys are underlined, common expressions are shaded): Profile 1: 'deduct% . database | multimedia database' Profile 2: 'multimedia database | data% system architect%'

number of terms in a subprofile is variable, we have chosen to describe a subprofile in the *SUBPROFILES* table in the following way: for each term of a subprofile a corresponding tuple is inserted; the last column, *ProfilePos*, holds the position of the term within its subprofile; the *Distance* column expresses the number of words between the actual term and the next term in the subprofile as is defined by the distance operator used in that subprofile. The special value -2 illustrates that there are no further terms in the subprofile. Finally, the *PROFILES* table represents the profiles in disjunctive normal form over subprofiles.

First analyses of the WWW queries to ODS show that there is a significant number of common sub-expressions, as for example common terms (in Fig. 8.18 the shaded terms 2 and 3) and common subprofiles (in Fig. 8.18 the shaded subprofile 2). The elimination of these can be done very efficiently during the normalization of profiles. Thus, superfluous word and word pattern search efforts can be avoided.

8.6.3.4 Parallel Processing of Normalized Profiles

The profiling service can basically be regarded as the execution of a large number of full-text retrieval queries that are combined to a single QEP as explained above. The parallel processing of profiles can be decomposed into the following four steps.

8.6.3.5 Term Mapping

Term mapping, step 1, basically joins the *TERMS* and *WORDS* tables, thus getting the word numbers that correspond to a term. Both relations are clustered in sorted order on the join attribute, so a merge algorithm is best suited for this join.

As can be seen from Fig. 8.19, the resulting QEP can be parallelized by using the concepts provided by our *TOPAZ* parallelizer, as well as the *Model-M* optimizer. Obviously, the declustering of the base relations that supports data parallelism best depends on the concrete architecture

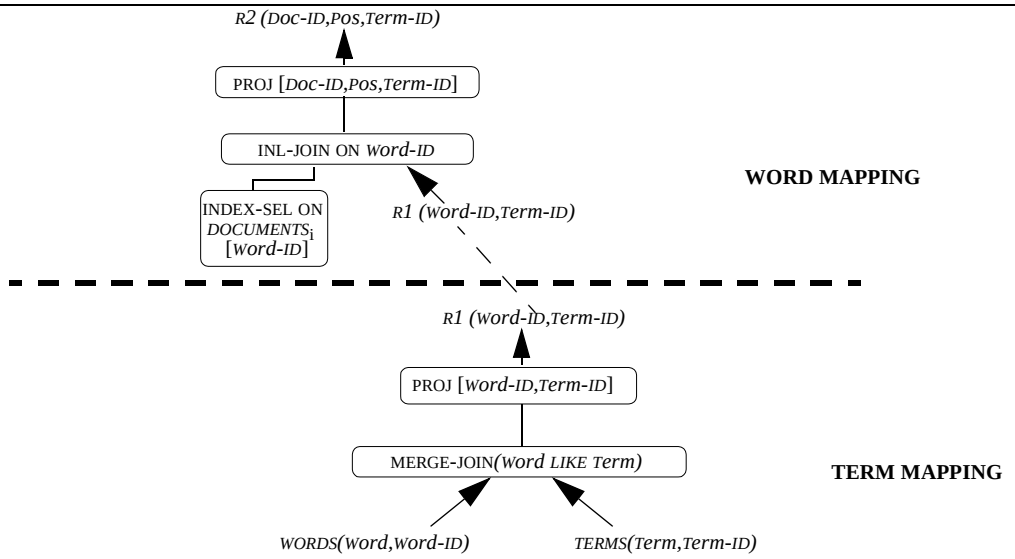


Fig 8.19: Term mapping and word mapping on a single processing node

employed and can be taken in consideration accordingly during optimization by means of physical properties. In addition to using data parallelism for processing, the resulting tuples of this step can be immediately used as input for the next step, thereby allowing for pipeline parallelism.

8.6.3.6 Word Mapping

Word mapping, i.e. step 2, determines documents and document positions corresponding to a term. For this reason, the intermediate result of step 1 (*R1*) is joined with the *DOCUMENTS* table resulting in the intermediate result *R2*. Fig. 8.19 shows the first two phases. An index nested-loops-join is used for this step. The intermediate results *R2* are used to build hash-tables on the *Term-ID* attribute. This is a kind of preprocessing in order to speed up the subsequent step 3.

Again appropriate techniques presented in the previous chapters apply and can be used to achieve a scalable parallel execution of this step, e.g. by using data parallelism.

8.6.3.7 Subprofile Mapping

Step 3 determines all the documents satisfying any given subprofile. The word distances between terms have to be obeyed and only those documents qualify that contain all terms of a subprofile. In order to do this, the partial result of step 2 (*R2*) must be joined with the *SUBPROFILES* table on *Term-ID*. Remember that we have a previously built hash-table on *Term-ID* which can be used to perform efficient hash-joins. Since the number of terms in a subprofile is variable, this join produces a variable number of result streams, each stream holding document positions that contain a certain term. To obtain the documents that contain all terms, these streams have to be joined on the *Doc-ID* attribute. This operation cannot be processed by traditional database operators (as detailed below). Hence, the only possibility is to perform this operation outside the DBMS and within the (profiling) application. But this leads to immense performance losses, as all streams have to be transmitted to the application, merged, and, as far as further operations are needed for post processing, these have to be transmitted back. The profiling service is a good example where pushing semantics down into the DBMS is vital, as this would reduce high com-

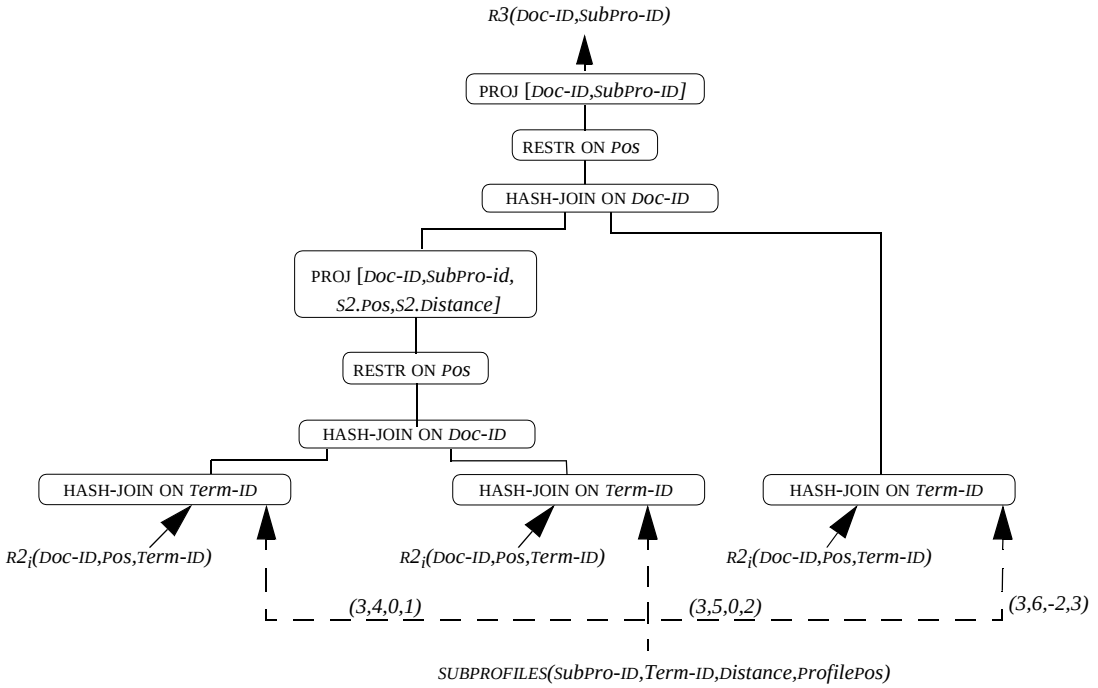


Fig 8.20: Subprofile mapping using the tuple-stream evaluation

munication costs between the application and the database. In the following, we exemplify how this can be achieved for our application.

Approach 1: Tuple-Stream Evaluation

One possibility is to use a so-called “Tuple-Stream Evaluation”. The *SUBPROFILES* table is sorted on *SubPro-ID* and *ProfilePos*. This guarantees that the terms of the same subprofile arrive one after another in the same order as specified in the subprofile. These tuples are probed in parallel against the *R2* hash table. In this way separate streams of document numbers get produced in parallel. Next, these streams are joined two at a time and the result is restricted according to the distance operator specified between two consecutive terms. In Fig. 8.20., this approach is exemplified for subprofile 3 of our sample schema (that holds terms 4, 5 and 6, cf. tables in Fig. 8.18). The difficulty of this approach lies in the fact that the number of terms varies from subprofile to subprofile, so that the number of operators is not known at plan generation time and the height of the QEP varies accordingly. Thus this approach provides no suitable solution to our problem.

Approach 2: StreamJoin

The other possibility is to extend the database functionality by means of the *StreamJoin* operator as introduced in the previous sections. *StreamJoin* combines exactly the “variable part” of the QEP for the Tuple-Stream Evaluation discussed above. Note that every tuple of the *SUBPROFILES* table produces a stream *docstr(SubPro-ID, ProfilePos, Doc-ID, ...)* when probed against the R2 relation (expressing all documents that contain that term). Thus, a subprofile defines a *group* whose streams have to be joined to obtain the documents that contain all terms of this subprofile. The parameters that define a group and the streams within a group are in this case *SubPro-ID* and *ProfilePos*. The join predicate *PRED* in this case is complex, expressing both equality on the *Doc-ID* attribute (*Doc-ID.new = Doc-ID. old*), as well as the conformity with the position requirements, expressed by the distance operator. Please note that the expression *Pos.new = Pos.new +*

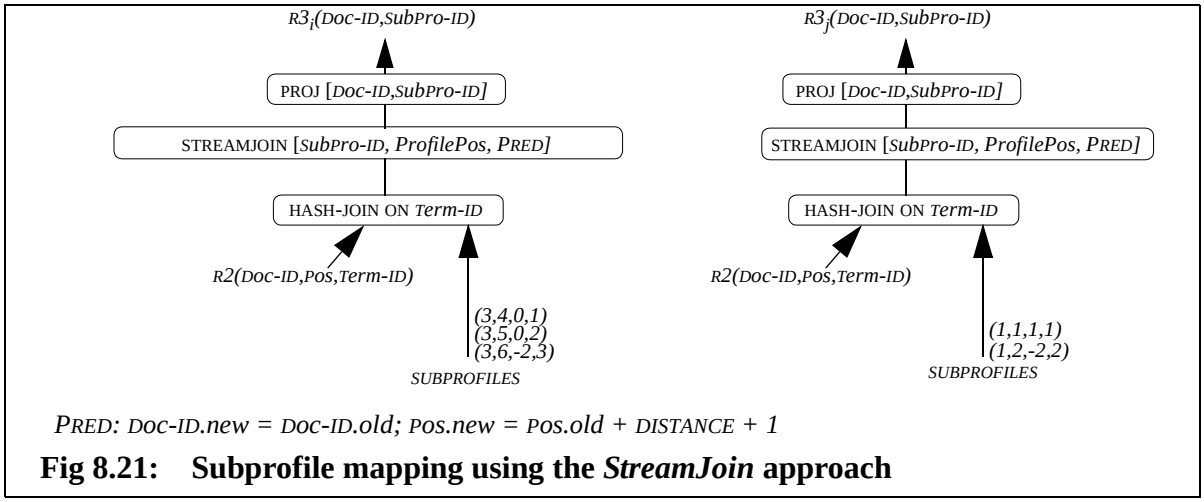


Fig 8.21: Subprofile mapping using the *StreamJoin* approach

Distance + 1 is given only for illustration purposes. Obviously, in a real-life application more complex position requirements can be envisioned and expressed within the *StreamJoin* predicate.

Though the tuples of one subprofile are processed within one operator, the tuples of different subprofiles can be processed by different *StreamJoin* operations, thus allowing data parallelism for the *SUBPROFILES* table as shown in Fig. 8.21. Here, the first subplan instance processes subprofile *3*, while the second one evaluates in parallel subprofile *1*. Please note that data parallelism is also applicable w.r.t. the other input of this step (R2).

The parallelization can be efficiently realized by means of TOPAZ in this case as well. Therefore, the *StreamJoin* operator has to be incorporated into the cost model as discussed in Section 8.2.8. The fact that the input has to be partitioned on the *SubPro-ID* attribute (i.e. the *Group-ID* parameter cf. the definition given in Section 8.2.3), can be specified as a required physical property that has to be satisfied in the final, parallel plan.

8.6.3.8 Profile Mapping

The evaluation of profiles, i.e. step 4, is now trivial. The result of the last step has to be joined on the *SubPro-ID* columns with the *PROFILES* by probing the tuples of this table against the hash-tables built one step before. Finally the duplicates resulting from documents matching more than one subprofile of a given profile have to be eliminated in the result for each profile. This step can also use data parallelism for the operators involved.

8.6.4 Performance Evaluation

For the performance evaluation we have used our 1.2 GB department library, running on a SUN SPARC20 workstation with 4*100 MHz SPARC processors. The data has been partitioned onto 4 disks. We have evaluated about 600 user profiles, each of them being constituted of 10 to 15 search terms. The results are summarized in Table 8.2.

First, we evaluated each profile sequentially. Next, we parallelized each query separately and executed them one after the other on our workstation. As can be seen in line 2 of Table 8.2, the speedup in this case is not considerable, as the structure of the QEPs for full-text retrieval as

Table 8.2 Performance evaluation for profile calculation

Execution Mode		Time (min)
Sequential Batch Execution		56
Parallel Batch Execution		41
4-Step, Sequential		32
4-Step, Parallel	DOP = 1	29
	DOP = 2	22
	DOP = 3	18
	DOP = 4	15

presented in Fig. 8.17 contains several correlations. As shown in Chapter 6, this prohibits an efficient parallelization. Moreover, as already mentioned in Section 8.6.3.2, there is a considerable overhead due to the redundant plan generation and startup costs.

In the following, we used our 4-step processing scheme, without employing any forms of parallelism. As shown in the measurement results, the elimination of redundant work alone resulted already into a considerable performance improvement. By employing parallelism for subprofile mapping as discussed in Section 8.6.3.7, the performance could be further improved as shown in lines 4 to 7 of Table 8.2. Here, the results refer to different degrees of parallelism, that has been varied from 1 (thus employing only pipelining parallelism) to 4. We stress once again that for this performance evaluation we have only used parallelism for subprofile mapping: Thus the results of Table 8.2 corresponding to different DOPs incorporate also the (constant) costs of the sequential (unparallelized) parts.

The results show that intra-query parallelism provided by the concepts presented in the previous chapters of this thesis results into increased efficiency also in the case of batch queries that represent user profiles. Please note that all four steps can employ data parallelism in a scalable way. This means that the profiling service is now bounded only by the limits the hardware poses on parallelism. Moreover, the resources needed for evaluation have been drastically reduced by multiple query optimization.

8.7 Conclusions

In this chapter we have discussed different applications involving stream processing. As presented, this class includes such important applications as data mining, time series, DNA analysis, universal quantification and digital libraries. We have proposed a new operator, called *StreamJoin*, to efficiently process streams on the database engine level. Contrary to related work, the *StreamJoin* approach provides a resource-effective and efficient strategy to solve the problem of stream analysis, avoiding expensive data transfer manipulations. Stream analysis like most data analysis, is best done in a way that permits interactive exploration. The *StreamJoin* approach presented in this chapter is a novel strategy towards efficiently satisfying ‘ad hoc’

queries as well.

Moreover, we have shown that this new functionality can be efficiently integrated with the database engine, both on the execution as well as on the parallelization level. We have shown that extending the database functionality is certainly worthwhile, when, as in most applications presented in this chapter, the communication between the database and the application would become a considerable part of the overall processing costs. By means of integration into the DBMS, we achieve a robust and scalable environment for parallel execution of these queries. This results in efficient execution, reliability, and portability of the corresponding applications. The non-blocking feature of *StreamJoin* can be used for pipelining purposes as well.

As presented by relevant examples in this chapter, adaptability to diverse application domains is provided by means of appropriate parameter settings. Furthermore, we have shown that the *StreamJoin* operator can easily be integrated into our parallelization framework, implicitly yielding increased efficiency to the diverse QEPs containing this operator. The linear and near-linear speedups obtained prove the extensibility and effectiveness of the concepts presented in the previous chapters for truly complex, real-life applications as well.

Chapter 9
Conclusions and Future Work

In this chapter first a brief summarization of the main contributions is given. Thereby, we also discuss the applicability of the proposed techniques for different scenarios. Finally, the chapter concludes with some remarks concerning future work.

9.1 Summary of Contributions

In this thesis we have addressed the problem of inter-query parallelism in parallel object-relational database systems. The presented contributions cover a wide domain in the query processing architecture, starting with parallel query execution, parallelization, optimization as well as scheduling and load balancing. They conclude with the discussion on specific optimization and parallelization techniques for a given application class. In the following, we group our conclusions and discussion on efficiency and applicability around the main topics introduced in the previous chapters.

9.1.1 Data Rivers

First, we presented a *modularization* and *parametrization* approach for the implementation of data rivers that serves as basis for most parallel query execution engines. Our approach to parameterize a PQEP allows to adapt important performance indicating parameters to the existing run-time situation.

We have presented the concept of *block building* in order to achieve coarse-grained parallelism and thus to reduce the number and size of data rivers. Furthermore, measures for reducing the number of data streams and execution units as well as for resolving deadlock situations have been addressed as well.

Since our data river implementation perfectly fits into state-of-the-art parallel database engines, any functional extensions to relational data processing, as e.g. recursion, (set-oriented) triggers, user-defined abstract data types, user-defined functions and user-defined tables, as being discussed for example in the object-relational context can still benefit from our approach in case of parallel processing. Even if the herein mentioned communication patterns do not suffice for

some particular extension, we are still confident that new communication patterns can be easily reflected within our data river approach, because of its modularization and extensibility issues.

9.1.2 The TOPAZ parallelizer

All insights previously gained from the analysis and implementation of the data river paradigm have been incorporated into our parallelizer, called *TOPAZ*. In addition, this component bears further important characteristics that are imperative in order to satisfy the requirements posed by upcoming applications.

Its *'rule-driven'* property guarantees for the necessary extensibility. Both language extensions and extensions to the database engine itself, as well as changes to the parallel system architecture can be accomplished by means of respective rules. Its *'multi-phase'* property realizes an overall strategy that considers all forms of parallelism. It splits the parallelization task into subsequent phases, with each phase concentrating on particular aspects of an efficient parallel execution. In addition, this property turned out to be a major concept to handle the inherent complexity of parallelization. Its *'cost-based'* property guarantees that all decisions w.r.t. investigating the parallel search space are cost-based. Hence, promising search space regions are explored to derive the best parallel plan.

A prerequisite to optimization performance is pruning. The strategy developed for TOPAZ, called *ParPrune*, is based on a global plan pre-analysis and exploited throughout the parallelization phases, within each focusing on valuable search space regions. Since the complex parallelization strategy of TOPAZ is mainly suitable for large and costly queries, *ParPrune* is also in charge of preventing unnecessary parallelization overhead. Thus, if the global costs of a plan resulting from the pre-analysis are below a given threshold, the actual parallelization task does not come to application at all.

The concept of blocks incorporated within the cost model enables (coarse-grain) parallelism to low-cost as well as high-cost operators. Moreover, by the deadlock-preventing facility of TOPAZ cyclic data dependencies can be detected and eliminated already on the parallelization level. It further guarantees economical and efficient resource consumption. As a result efficient parallel plans are generated that show linear speedup even for complex queries as can be found in data warehouse and object-relational environments.

9.1.3 Model-M Optimizer

The integration of optimization and parallelization has been an open problem in most related research activities. In this respect, the one- and two-phase approaches lie at the two different ends of a spectrum, incorporating either detailed or no knowledge of the parallel environment. In our opinion, a combined approach, i.e. an optimizer taking into account some parallel aspects, followed by a detailed parallelization as realized within the TOPAZ approach, is the most suit-

able for forthcoming query scenarios. This is reflected within our new optimizer component, called *Model-M*.

Here, the subsequent parallelization task is considered already at sequential plan generation time by means of a so-called *quasi-parallel cost model*. In order to avoid excessive optimization complexity this model accounts only for the most important aspects and deterrents of parallel execution. These aspects include blocking boundaries, correlations as well as the shape of the resulting plan. As a result sequential plans with a high potential for intra-query parallelism are produced.

We have demonstrated experimental results to support our claim that, in contrast to other strategies, the incurred overhead is not significant. As our approach uses the same top-down search engine for the optimizer and the parallelizer, Model-M can be viewed as a (first) phase of an overall multi-phase parallelization strategy where each phase concentrates on different, gradually refined aspects of the parallel search space. Hence, this concept lends itself predominantly to the area of complex query optimization and parallelization.

9.1.4 The QEC component

One important reason for sub-optimality of query execution plans is that a lot of information about the run-time system is not available at query optimization, respectively parallelization time. In addition, there are a number of reasons why estimating the cost of execution for complex queries becomes increasingly difficult. This observation applies especially for the area of parallel object-relational database systems. Our approach to tackle this problem is to produce parametric plans and to adapt the final values of these parameters to the run-time environment. This task is accomplished by the *QEC* component.

Load balancing and resource allocation are realized within a *two-phase* scheduling scheme. Thus the first phase concentrates on global aspects and assigns adapted (sub)plans to processing sites, while the second phase is in charge of elaborating a detailed schedule plan for the given processing site.

The loose coupling between the information subsystem that provides the necessary load information and the actual scheduling task guarantees that the incurring overhead, in terms of messages etc., is also minimal.

In contrast to most related work, this *distributed* approach supports both scalability and multi-query scenarios, as well as upcoming hybrid heterogeneous architectures.

9.1.5 The StreamJoin Operator

Finally, we have proposed the *StreamJoin* operator as an efficient method to support a specific application class that deals with *sets of items*. This class includes various applications such as data mining w.r.t. the generation of association rules, pattern matching, universal quantification,

time series as well as digital libraries w.r.t. profile evaluation. We have shown how this approach can be efficiently integrated with the database engine, thus being able to make profit of all forms of query execution optimizations, including parallelization. In this way, we have performed a first validation of some of the concepts presented above w.r.t. extensibility and the ability to efficiently deal with the complexity of the parallel search space.

9.2 Future Work

The primary focus of further activities constitute a comprehensive validation and consequent improvement of the presented concepts w.r.t. other application domains as well, thereby especially concentrating on object-relational extensions. In the following we only point out the main topics of future work.

9.2.1 Query Execution

On the execution level, the data river paradigm with its modularization and parametrization approach already proved to be suitable for the support of user-defined functions and predicates as well as of user-defined table operators as described in [Ja99].

However, the parallel evaluation of specific constructs still deserves a further careful study. Herein, given the inherent inaccuracies of cost estimations in the presence of user-defined functionality, especially possibilities for *run-time adjustment of operators* to the current processing environment (similar to dynamic query execution plans [GC95]) should be analyzed.

9.2.2 Query Optimization and Parallelization

As with the implementation and validation of TOPAZ and Model-M the first phase of elaborating suitable parallelization strategies is finalized, we will further concentrate on extending them to other scenarios as well. A possible alternative is to use TOPAZ for *hybrid* optimizer solutions, e.g. to map logical trees, obtained by a bottom-up search strategy, to physical ones, similar to the NEATO optimizer [McB+96]. Here, the bottom-up search strategy is used to enumerate all join orders and the top-down strategy is used to perform the mapping from logical to physical operators in a parallel environment. Although only joins have been considered, optimization time was dominated by the mapping phase, due to the high number of possible mappings from logical operators to physical solutions in a parallel DBMS. We believe that the mapping problem becomes even more complex when new operator types, as e.g. UDFs, have to be considered as well.

Another research direction constitute performance improvement measures as well. Thus, a *tighter coupling between optimization and parallelization* can be envisioned to further eliminate parallelization overhead.

Furthermore, possible modifications in the search engine of the Cascades Optimizer Framework itself should be explored. For instance, the *independency assumption* that is the basis of most sequential optimizers, including Cascades, is no longer valid in the parallel search space. In the running first version of TOPAZ the treatment of this issue is realized solely by the *ParPrune* strategy. However, making the search engine cognizant of this aspect is also an interesting path to follow. Generally, a concept for the accurate modeling of dependencies within a query execution plan should be elaborated, since this aspect is equally important in the sequential search space with respect to e.g. correlations and subqueries.

Furthermore, our performance evaluation showed that for the Cascades version currently employed the memory usage and optimization time can become prohibitive when optimizing very large queries. Thus, it is vital to integrate further *pruning techniques* as used by the successor of Cascades, Columbia [Ju99, SM+98]. We are confident our approach can make profit of other *engineering improvements* as well, as described in [Xu98].

Finally, we intend to *extend the existing set of parallelization, respectively optimization rules* for object-relational environments as well. Thereby, the interaction of rules bear a special interest. Provided its beforementioned characteristics, we hope that our parallelization approach can easily be extended while still retaining its high efficiency.

9.2.3 Resource Allocation, Scheduling and Load Balancing

Another important issue is the further improvement of *run-time adaptability.* In our approach, the query optimizer/parallelizer tries to anticipate the most common cases that might arise at run-time and produce a parameterized plan that covers these possibilities. If a situation arises at run-time that is not covered by the common cases anticipated, a *hybrid* approach involving dynamic re-optimization, as presented e.g. in [KD98], can be envisioned.

As emerging new applications force databases to support complex decision support queries, complex data types and user-defined methods, it will become more and more difficult for query optimizers to statically produce good query execution plans. Run-time adjustment will become imperative in such cases. We believe that the techniques we have presented, possibly in combination with some forms of re-optimization and dynamic query execution plans will form the basis for the future development of query optimizers to meet this challenge.

Appendix A
References

A.1 References

AY97 C. C. Aggarwal, P. S. Yu: Mining Large Itemsets for Association Rules, TCDE Bull., 21(1), March 1997

AY98 C. C. Aggarwal, P. S. Yu: Online Generation of Association Rules, In: DE Conf., Orlando, Florida, 1998

AM+95 R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo: Fast Discovery of Association Rules, Advances in Knowledge Discovery and Data Mining, Chapter 12, AAAI/MIT Press, 1995

AS96 R. Agrawal, J. C. Shafer: Parallel Mining of Association Rules, In: TKDE 8(6): 962-969, 1996

BF97 C. Ballinger, R. Fryer: Born to be Parallel, In: Bulletin of the IEEE Computer Society TCDE, Vol. 20, No. 2, 1997

BKK88 J. Banerjee, W. Kim, K.-C. Kim: “Queries in Object-Oriented Databases, Proc. of the 4th Int. Conf. on Data Engineering, 1988, S. 31-38

BF+95 C. K. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G. P. Copeland, W.G. Wilson: DB2 Parallel Edition, In: IBM Sytems Journal, Vol 34, No 2, 1995

Ba98 R. Bayardo: Efficiently Mining Long Patterns from Databases, In: Proc. SIGMOD Conf., Seattle, 1998

BLO93 D. Benson, D. Lipman, and J. Ostell: GenBank, In: Nucleic Acids Research, 21(13):2963--2965, 1993.

Bi97 Billings, K.: A TPC-D Model for Database Query Optimization in Cascades, Master Thesis, Portland State University, CS Department, 1997

Blak97 J.A. Blakeley: Universal Data Access with OLE DB, In: Proc. of the IEEE COMPCON’97, 1997,S. 2-7

BO+95 C. Böhm, A. Oppitz, P. Vogel, S. Wiesener: Prints of the 17th Century in a Distributed Digital Library System, DEXA ‘95 (Database and Expert Systems Applications), 6th International Conference, LNCS 978, Springer, 1995

BDV96 L. Bouganim, B. Dageville, P. Valduriez: Adaptive Parallel Query Execution in DBS3, In: EDBT Conf., Avignon, 1996

BFV96 L. Bouganim, D. Florescu, P. Valduriez: Dynamic Load Balancing in Hierarchical Parallel Database Systems, In: Proc. of the 22nd VLDB Conference, Bombay, India, 1996

Boz98 G. Bozas: Scalability in Parallel Database Systems, PhD Thesis, Fakultät für Informatik, Technische Universität München, 1998

BFZ97 G. Bozas, M. Fleischhauer, S. Zimmermann: PVM Experiences in Developing the MIDAS Parallel Database System, Proc. of the 4th European PVM User Group Meeting, Krakow, Poland, 1997

BJ+96 G. Bozas, M. Jaedicke et al: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project, In: Proceedings of the EURO-PAR Conf., 1996

Br97 F. Brandmayer: Parallel Query Execution in MIDAS (in German), Master Thesis, Fakultät für Informatik, Technische Universität München, 1997

BM+97 S. Brin, R. Motwani, J. Ullmann, S. Tsur: Dynamic Itemset Counting and Implication Rules for Market Basket Data, In: Proc. ACM SIGMOD Conf., 1997

BV98 S. Brobst, B. Vecchione: DB2 UDB: Starburst, In: Database Programming & Design, Feb. 1998

CCW93 F. Cacace, S. Ceri, M. A. W. Houtsma: A Survey of Parallel Execution Strategies for Transitive Closure and Logic Programs. Distributed and Parallel Databases 1(4): 337-382 (1993)

Ce96 P. Celis: The Query Optimizer in Tandem's new ServerWare SQL Product, In: Proc. VLDB Conf., India, 1996

Ch96 D. Chamberlin: Using the New DB2, Morgan Kaufman Publishers, San Francisco, 1996

Ch98 S. Chaudhuri: Data Mining and Database Systems: Where is the Intersection?, In: Bulletin of the TCDE, 21(1), March 1998

CYW96 M.-S. Chen, P. Yu, K.-L. Wu: Optimization of Parallel Execution for Multi-Join Queries, IEEE Transactions on Knowledge and Data Engineering, Vol. 8, No. 3, June 1996, p. 416-428

CJ+97 A. Clausnitzer, M. Jaedicke, B. Mitschang, C. Nippl, A. Reiser, S. Zimmermann: On the Application of Parallel Database Technology for Large Scale Document Management Systems, Proc. IDEAS Conf., Montreal, 1997

CV+95 A. Clausnitzer, P. Vogel, S. Wiesener: A WWW interface to the OMNIS/Myriad literature retrieval engine, In: COMPUTER NETWORKS and ISDN SYSTEMS 27 (1995), p. 1017-1027, ELSEVIER, 1995.

CK+97 J. Claussen, A. Kemper, G. Moerkotte, K. Peithner: Optimizing Queries with Universal Quantification in Object-Oriented and Object-Relational Databases, In: Proc. VLDB Conf, Athens, Greece, 1997

DG95 D. Davidson, G. Graefe: Dynamic Resource Brokering for Multi-User Query Execution, In: Proceedings of the 1995 ACM SIGMOD Intl. Conf. on Management of Data, San Jose, 1995

Da87 U. Dayal: Of nests and trees: A unified approach to processing queries that contain nested subqueries, In: Proc. VLDB Conf., Brighton, 1987.

DeWi90 D. DeWitt et al: The Gamma Database Machine Project, In: IEEE Transactions on Knowledge and Data Engineering, March 1990.

DG92 D. DeWitt, J. Gray: Parallel Database Systems: The Future of High Performance Database Systems, In: CACM, Vol.35, No.6, pp.85-98, 1992

FS+98 M. Fang, N. Shivakumar et al: Computing Iceberg Queries Efficiently, Proc. VLDB Conf., New York, 1998.

Fl97 M. Fleischhauer: Parallelization of Relational Database Queries in MIDAS (in German), Master Thesis, Fakultät für Informatik, Technische Universität München, 1997

GGS96 S. Ganguly, A. Goel, A. Silberschatz: Efficient and Accurate Cost Models for Parallel Query Optimization, In: Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems,, Montreal, 1996

GHK92 S. Ganguly, W. Hasan, R. Krishnamurty: Query Optimization for Parallel Execution, In: Proc. SIGMOD Conf., San Diego, California, USA, 1992

GGS96 S. Ganguly, A. Goel, A. Silberschatz: Efficient and Accurate Cost Models for Parallel Query Optimization, In: Proc. SIGACT-SIGMOD-SIGART Symp. on Principles of DB Systems, Montreal, 1996

GI96 M. Garofalakis, Y. Ioannidis: Multi-dimensional Resource Scheduling for Parallel Queries, in: Proceedings of the 1996 ACM SIGMOD, Vol. 25, No. 2, 1996

GI97 M. Garofalakis, Y. Ioannidis: Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources, In: Proceedings of the 23rd VLDB Conference, Athens, Greece, 1997

Ge94 Geist, A. et al.: PVM 3 User‘s Guide and Reference Manual. TR ORNL/TM12187, Oak Ridge Nat. Lab., 1994

Gh90 S. Ghandeharizadeh, Physical Database Design in Multiprocessor Systems, PhD thesis, University of Wisconsin - Madison, 1990.

Gr94 G. Graefe: Volcano-An Extensible and Parallel Query Execution System, In: TKDE, 6(1), 1994

Gr95 G. Graefe: The Cascades Framework for Query Optimization, In: DE Bulletin, 18(3), 1995

Gr96 G. Graefe: Relational Engine and Query Processing in Microsoft SQL Server, In: Proc. of the Intl. Conf. on Data Engineering, New Orleans, 1996

GC94 G. Graefe, R. L. Cole: Optimization of Dynamic Query Evaluation Plans, In: Proc. of the 1994 ACM-SIGMOD Conf., 1994

GC95 G. Graefe, R. L. Cole: Fast Algorithms for Universal Quantification in Large Databases, In: TODS 20(2): 187-236, 1995.

Gray95 Gray, J.: A Survey of Parallel Database Techniques and Systems, In: Tutorial Handout at the Int. Conf. on Very Large Databases, Zurich, 1995

GB+96 J. Gray, A. Bosworth et al: Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals, In: Proc. Intl. Conf. on Data Engineering, New Orleans, 1996

Has95 W. Hasan: Optimization of SQL Queries for Parallel Machines, PhD Thesis, Stanford Univ., 1995

HK+97 L. Haas, D. Kossmann et al: Optimizing Queries across Diverse Data Sources, In: Proc. of the 23rd VLDB Conf., Athens, Greece, 1997

HF+96 J. Han, Y. Fu et al: A Data Mining Query Language for Relational Databases, In: Proc. SIGMOD Conf, Montreal, 1996.

HGY98 J. Han, W. Gong, Y. Yin: Mining Segment-Wise Periodic Patterns in Time Related Databases, In: Proc. Intl. Conf. on Knowledge Discovery and Data Mining, New York City, NY, August 1998

HKK97 E.-H. Han, G. Karypis, V. Kumar: Scalable Parallel Data Mining for Association Rules, In: SIGMOD Conference, Tucson, Arizona, 1997

HFV96 W. Hasan, D. Florescu, P. Valduriez: Open Issues in Parallel Query Optimization, In: SIGMOD Record 25(3), 1996

Hi98 M. Hilbig: Development of a Cost-Based Query Optimizer for the Parallel Relational Database System MIDAS (in German), Master Thesis, Fakultät für Informatik, Technische Universität München, 1998

HS93 W. Hong, M. Stonebraker: Optimization of Parallel Query Execution Plans in XPRS, In: Distributed and Parallel Databases, pp. 9-32, 1993

HWF93 M. A. W. Houtsma, A. N. Wildschut, J. Flokstra: Implementation and performance evaluation of a parallel transitive closure algorithm on PRISMA/DB, in: Proceedings of the 19th Intl. Conf. on Very Large Data Bases, Dublin, 1993.

HP95 P. Hsu, D. Parker: Improving SQL with generalized quantifiers, In: Proc. Data Engineering Conference, Taipeh, Taiwan, 1995.

Ibm98 IBM DB2 Universal Database, Version 5, IBM Corp., 1998

Inf98 Extended Parallel Option for Informix Dynamic Server: Technical Brief, Informix Software, http://www.informix.com

Ja99 M. Jaedicke: New Concepts for Parallel Object-Relational Query Processing, PhD Thesis, Fakultät für Informatik, Universität Stuttgart, 1999

JM98 M. Jaedicke, B. Mitschang: A Framework for Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS, Proc. SIGMOD Conf., Seattle, 1998

JM99 M. Jaedicke, B. Mitschang: User-Defined Table Operators: Enhancing Extensibility for ORDBMS, Proc. VLDB Conference, Edinburgh, 1999.

JMP97 A. Jhingran, T. Malkemus, S. Padmanabhan: Query Optimization in DB2 Parallel Edition, In: Data Engineering Bulletin, 20(2), 1997

Ju99 K. Julisch: Extensibility and Efficiency of Top-Down Query Optimizers, Master Thesis, Universität Stuttgart, 1999

KD96 N. Kabra, D. DeWitt: OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization, Proc. ACM SIGMOD Conf., 1996

KD98 N. Kabra, D. DeWitt: Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans, Proc. ACM SIGMOD Conf., 1999

KB98 K. Krueger-Barverls: Development of a Rule-based Query Optimizer for the Parallel Object-Relational Database System MIDAS (in German), Master Thesis, Fakultät für Informatik, Technische Universität München, 1998

LK98 D. Lin, Z. M. Kedem: Pincer-Search: a New Algorithm for Discovering the Maximum Frequent Set, In: Proc EDBT Conf., Valencia, Spain, 1998

Li94 Listl, A.: Using Subpages for Cache Coherency Control in Parallel Database Systems, Proc. PARLE Conf., 1994

LB97 Listl, A., Bozas, G.: Performance Gains Using Subpages for Cache Coherency Control, Proceedings of the 8th International Workshop on Database and Expert Systems Applications, Toulouse, France, 1997

Lo88 G. Lohman: Grammar-like Functional Rules for Representing Query Optimization Alternatives, In: Proc. of the ACM SIGMOD Conf., Chicago, 1988

LC+93 M.-L. Lo, M.-S. Chen et al: On Optimal Processor Allocation to Support Pipelined Hash Joins, In: Proc. SIGMOD Conf., Washington D. C., 1993

LOT94 H. Lu, B. Ooi, K. Tan (eds.): Query Processing in Parallel Relational Database Systems, IEEE Computer Society Press, Los Alamitos, California u.a., 1994

LST91 H. Lu, M. Shan, K. L. Tan: Optimization of Multi-Way Join Queries for Parallel Execution, In: Proc. VLDB Conf., San Mateo, USA, 1991

LT94 H. Lu, K. Tan, : Load-Balanced Join Processing in Shared-Nothing Systems, in: Journal of Parallel and Distributed Computing 23, p. 382-398, 1994

LVZ93 R. Lanzelotte, P. Valduriez, M. Zait: On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces, In: Proc. VLDB Conf., Dublin, 1993

McB+96 W. McKenna, L. Burger et al: EROC: A Toolkit for Building NEATO Query Optimizers, In: Proc. of the 22nd VLDB Conf., Mumbai, India, 1996

MD95 M. Mehta, D. DeWitt: Managing Intra-operator Parallelism in Parallel Database Systems, in: Proceedings of the 21st Intl. Conference on Very Large Data Bases, p. 382-394, Zurich, 1995

MSD93 M. Mehta, V. Soloviev, D. DeWitt: Batch Scheduling in parallel database systems, in: Proceedings of the 9th Intl. Conf. on Data Engineering, p. 400-410, Vienna, 1993

MPC96 R. Meo, G. Psaila, S. Ceri: A New SQL-like Operator for Mining Association Rules, In: Proc. VLDB Conf, Mumbai, India, 1996

MPC98 R. Meo, G. Psaila, S. Ceri: A Tightly-Coupled Architecture for Data Mining, In: DE Conf., Orlando, 1998

MH95 H. W. Mewes, K. Heumann: Genome Analysis: Pattern Search in Biological Macromolecules. CPM 1995: 261-285

Mi95 B. Mitschang: Query Processing in Database Systems (in German), Vieweg Verlag, 1995.

MC98 L. Molesky, M. Caruso: Managing Financial Time Series Data: Object-Relational and Object Database Systems, Tutorial VLDB Conf., New York City, 1998

NJM97 C. Nippl, M. Jaedicke, B. Mitschang: Accelerating Profiling Services by Parallel Database Technology, In: Proc. PDPTA Conf., Las Vegas, 1997

NM98a C. Nippl, B. Mitschang: TOPAZ: a Cost-Based, Rule-Driven, Multi-Phase Parallelizer, Proc. VLDB Conf., New York City, 1998

NM98b C. Nippl, B. Mitschang: Towards Deadlock-Preventing Query Optimization and Parallelization, In: Proc. Intl. Conf. on Parallel and Distributed Computing Systems, Chicago, Illinois, 1998

NZM99 C. Nippl, S. Zimmermann, B. Mitschang: Design, Implementation and Evaluation of Data Rivers for Efficient Intra-Query Parallelism, Technical Report TUM-I0018, Technische Universität München, 1999.

NRM00a C. Nippl, A. Reiser, B. Mitschang: Towards Deep Integration of Data Mining Technology with Data Warehouses, Technical Report, Technische Universität München, 2000.

NRM00b C. Nippl, A. Reiser, B. Mitschang: Conquering the Search Space for the Calculation of the Maximal Frequent Set, Technical Report, Technische Universität München, 2000.

NZT96 M. Norman, T. Zurek, P. Thanisch: Much Ado about Shared-Nothing, In: ACM Sigmod Records, 23(3), 1996

Or99 Oracle8 Parallel Server Concepts, Oracle Corp, http://www.oracle.com.

OL90 K. Ono, G.M. Lohman: Measuring the Complexity of Join Enumeration in Query Optimization, In: Proc. Intl. Conf. on VLDB, Brisbane, 1990

ON+95 F. Ozcan, S. Nural et al: A Region Based Query Optimizer through Cascades Optimizer Framework, In: DE Bulletin 18(3), Sept 1995

PCY97 J. S. Park, M.S. Chane, P.S. Yu: Using a Hash-Based Method with Transaction Trimming for Mining Association Rules, In: IEEE Trans. on TKDE, 9(5), Sept. 1997

PGK97 A. Pellenkoft, C. Galindo-Legaria, M. Kersten: The Complexity of Transformation-Based Join Enumeration, In: Proc. VLDB Conf., Athens, 1997

Pe98 S. Perathoner: Development of a Component for Load Balancing for the Parallel Database System MIDAS (in German), Master Thesis, Fakultät für Informatik, Technische Universität München, 1998

Qi96 Qian, X.: Query Folding, in: IEEE Data Engineering Conference, pp. 48-55, 1996

RBG96 S. Rao, A. Badia, D. v. Gucht: Providing Better Support for a Class of Decision Support Queries, In: Proc. SIGMOD Conf., Montreal, 1996

Re98 Red Brick Systems Inc., http://www.redbrick.com/rbs-g/html/whpap.html

RP98 B. Reinwald, H. Pirahesh: SQL Open Heterogeneous Data Access, SIGMOD Conf., Seattle, 1998

RC88 A. Rosenthal, S. Chakravarthy: Anatomy of a modular multiple query optimizer, in: Proceedings of the 14th Intl. Conf. on Very Large Data Bases, p. 230-249, Los Angeles, 1988

SON95 A. Savasare, E. Omiecinski, S. Navathe: An Efficient Algorithm for Mining Association Rules in Large Databases, In: Proc. VLDB Conf., Zurich, 1995

STA98 S. Sarawagi, S. Thomas, R. Agrawal: Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications, In: Proc. ACM SIGMOD Conf, Seattle, 1998

Schn97 D. Schneider: The Ins and Outs of Data Warehousing, In: Tutorial on the VLDB Conference, Athens, 1997

SD90 D. Schneider, D. DeWitt: Tradeoffs in Processing Complex Join Queries via Hashing in Multi-processor Database Machines, In: Proc. of the Intl. VLDB Conference, Melbourne, Australia, 1990

Se88 T. Sellis: Multiple-query optimization, in: ACM Transactions on Database Systems, 13(1):23-52, March 1988

SM+98 L. Shapiro, D. Maier, K. Billings, Y. Fan, B. Vance, Q. Wang, H. Wu: Group Pruning in the Columbia Query Optimizer, Internal Report, Portland State University, 1998

SN95 A. Shatdal, J. Naughton: Adaptive Parallel Aggregation Algorithms, Proc. SIGMOD Conf., San Jose, 1995

STY93 E. Shekita, K. L. Tan, H. Young: Multi-Join Optimization for Symmetric Multiprocessors, In: Proc. VLDB Conf., Dublin, 1993

SK98 T. Shintani, M. Kitsuregawa: Parallel Mining Algorithms for Generalized Association Rules with Classification Hierarchy, In: Proc. SIGMOD Conference, Seattle, 1998

SK96 T. Shintani, M. Kitsuregawa: Hash Based Parallel Algorithms for Mining Association Rules. PDIS 1996

StMo96 M. Stonebraker, D. Moore: ORDBMS - The next Great Wave, Morgan Kaufman Publishers, 1996

SQL99 ISO/IEC 9075:1999, Information technology-Database languages-SQL-Part2: Foundation (SQL/Foundation), will be published in 1999.

Ta97 K.L. Tan: Decoupling Load-Balancing and Optimization Issues: A Two-Phase Query Processing Framework for Shared-Nothing Systems,In: Int. Journal of Computer Systems and Engineering 12(1), Jan. 1997

TL96 K. Tan, H. Lu: Scheduling Multiple Queries in Symmetric Multiprocessors, in: Information Sciences, Elsevier Science Publishing Inc, North-Holland, Vol95, Nos 1 & 2, p. 125-153, November 1996

Tand97 Nonstop SQL/MX Database, Tandem Computers Inc., 1997

TD93 J. Thomas, S. Dessloch: A Plan-Operator Concept for Client-Based Knowledge Processing, Proc. 19th VLDB Conference, Dublin, 1993

TPC95 Transaction Processing Performance Council. TPC Benchmark D, Stand. Spec,, Rev. 1.0, 1995

Trans95 TransBase System and Installation Guide, Version 4.2, TransAction Software GmbH, 1995

TFT99 T. Tsunoda, M. Fukagawa, and T. Takagi: Time and Memory Efficient Algorithm for Extracting Palindromic and Repetitive Subsequences in Nucleic Acid Sequences, In: Pacific Symposium on Biocomputing, 1999.

Va93 Valduriez, P.: Parallel Database Systems: Open Problems and New Issues, In: Distributed and Parallel Databases, Vol.1, No. 2, pp.137-166, April 1993.

WZ96 H. Williams, J. Zobel: Indexing nucleotide databases for fast query evaluation, In: Proc. EDBT Conf. Avignon, 1996.

WFA95 A. Wilschut, J. Flokstra, P. Apers: Parallel Evaluation of Multi-Join Queries, In: Proc. ACM SIGMOD Conf., 1995

Xu98 Y. Xu: Efficiency in the Columbia Query Optimizer, Master Thesis, Portland State University, 1998

Za98 M. J. Zaki: Efficient Enumeration of Frequent Sequences, In: Proc. CIKM Conference, Bethesda, 1998.

ZP+97 M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li: New Algorithms for Fast Discovery of Association Rules, In: Proc. Intl. Conf. on Knowledge Discovery and Data Mining, Newport Beach, California, 1997

ZZS93 M. Ziane, M. Zait, B. Salamet: Parallel Query Processing with ZigZag Trees, In: Very Large Databases Journal, 2(3), March 1993

Zi99 S. Zimmermann: PhD Thesis in preparation, Fakultät für Informatik, Technische Universität München, 1999

Zo97 C. Zou: XPS: A High Performance Parallel Database Server, In: Data Engineering Bulletin 20(2), 1997

Appendix B

Operators and Rules

B.1 Selected operators used in the MIDAS execution engine[1]

MIDAS Operator	Functionality
Rel	Relation scan
Restr	Relational restriction (predicate is given by the 2nd input)
Proj	Relational projection (corresponding tuple given by the 2nd input)
Build	Tuple construction (corresponding to the projection)
Times	Cartesian product; part of the index nested-loops construction
Mjoin	Merge join
Hjoin	Hash join
Group	Group by
Aggr	Aggregation
Uniq	Duplicate elimination
Sort	Sort
Corr	Materialization of intermediate results for repeated execution
Union	Union with duplicate elimination
Intersect	Intersection of two input sets
Attr	Attribute value
Const	Constant value
Cast	Type modification
Add	Addition
Sub	Substraction
Mul	Multiplication
Div	Division
Eq	Comparison (=)
Lt	Comparison (<)
Gt	Comparison (>)

MIDAS Operator	Functionality
Le	Comparison (<=)
Ge	Comparison (>=)
And	Logical “and”
Or	Logical “or”
Like	Comparison (pattern)

1. This represents only the list of some basic MIDAS operators, mostly inherited from the *TransBase* system. Further extensions, as e.g. user-defined functions and table operators, are not listed here.

B.2 Selected rules used by the TOPAZ parallelizer

Phase 1	
Name	**Description**
Insert_Pipe_1	Introduces inter-operator parallelism below a unary operator
Insert_Pipe_2	Introduces inter-operator parallelism below a unary operator
Corr_to_Send	Create a data river for the reuse of intermediate results in case of a correlation

Phase 2	
Name	**Description**
Mjoin_to_parMjoin	Parallelize a merge join operator
Rel_to_Pscan	Relation scan to parallel scan
Rel_to_Rscan	Relation scan to round-robin scan
Times_to_parTimes	Parallelize cartesian product and index nested-loops join
Hjoin_to_parHjoin	Parallelize a hash join operator
Group_to_parGroup	Parallelize group by
Aggr_to_parAggr	Parallelize aggregation
Uniq_to_parUniq	Parallelize duplicate elimination
Sort_to_parSort	Parallelize sorting
Union_to_parUnion	Parallelize union with duplicate elimination
Intersect_to_parIntersect	Parallelize intersection

Phase 3	
Name	**Description**
SendSend_to_Send	Replace two neighboring *send* operators by a single repartitioning node
SendCorr_to_Send	Replace neighboring *send* and *corr* operators by a single *send* node
CorrSend_to_Send	Replace neighboring *corr* and *send* operators by a single *send* node
SendRel_to_Pscan	Replace neighboring *send* and *rel* operators by a single *pscan* node
SendRel_to_Rrscan	Replace neighboring *send* and *rel* operators by a single *rrscan* node
ProjSend_to_SendProj	Push *send* through the *proj* operator (upwards)
SendProj_to_ProjSend	Push *send* through the *proj* operator (downwards)
RestrSend_to_SendRestr	Push *send* through the *restr* operator (upwards)

Phase 3	
Name	**Description**
SendRestr_to_RestrSend	Push *send* through the *restr* operator (downwards)
SortSend_to_SendSort	Push *send* through the *sort* operator (upwards)
SendSort_to_SortSend	Push *send* through the *sort* operator (downwards)
UniqSend_to_SendUniq	Push *send* through the *uniq* operator (upwards)
SendUniq_to_UniqSend	Push *send* through the *uniq* operator (downwards)
UnionSend_to_SendUnion	Push both input *send* operators through the *union* operator (upwards)
SendUnion_to_UnionSend	Push *send* operator through the *union* operator (downwards)
IntersectSend_to_SendIntersect	Push both input *send* operators through the *intersect* operator (upwards)
SendIntersect_to_IntersectSend	Push *send* operator through the *intersect* operator (downwards)
SendMjoin_to_MjoinSend	Push *send* through the *mjoin* operator (downwards)
MjoinSend_to_SendMjoin	Push *send* operator through the *mjoin* operator (upwards)
MjoinSendLeft_to_SendMjoin	Push left input *send* operator through the *mjoin* operator (upwards)
MjoinSendRight_to_SendMjoin	Push right input *send* operator through the *mjoin* operator (upwards)
SendHjoin_to_HjoinSend	Push *send* through the *hjoin* operator (downwards)
HjoinSend_to_SendHjoin	Push *send* operator through the *hjoin* operator (upwards)
HjoinSendLeft_to_SendHjoin	Push left input *send* operator through the *hjoin* operator (upwards)
HjoinSendRight_to_SendHjoin	Push right input *send* operator through the *hjoin* operator (upwards)
GroupSend_to_SendGroup	Push *send* through the *group* operator (upwards)
SendGroup_to_GroupSend	Push *send* through the *group* operator (downwards)
TimesSend_to_SendTimes	Push left input *send* operator through the *times* operator (upwards)
SendTimes_to_TimesSend	Push *send* through the *times* operator (downwards)

Phase 4	
Name	**Description**
Remove_SendNode	Remove *send* operator in order to perform block combination

B.3 Logical operators used in Model-M.

Name	Functionality
GET	Get a table from disk
SELECT	Relational restriction
PROJECT	Relational projection
GROUP_BY	Logical grouping operator
M_AGGR	Aggregation applied to the entire input relation
M_UNIQ	Duplicate elimination
EQJOIN	Models both joins as well as cartesian products
M_UNION	Logical operator modeling the union of the two input tables

B.4 Physical operators used in Model-M

Name	Functionality
REL_SCAN	Full table scan
REL_INDEX_SCAN	Table scan using an index
RESTRICTION	Physical restriction operator
P_PROJECT	Physical projection operator
P_GROUP_BY	Divides the input relation into groups and eventually aggregates them according to an aggregation function
P_AGGR	Similar to P_GROUP_BY, except that the aggregation function is applied to the entire input relation
P_UNIQ	Physical duplicate elimination, requires a sorted input
P_HASH_UNIQ	Physical duplicate elimination by using a hash table, no sorted input is necessary
P_SORT	Performs a sorting on the input table according to the specified attributes Can also perform duplicate elimination
HASH_JOIN	Implements the join of the two input tables by using a hash table If the hash table doesn't fit into the main memory, disk spoolings are necessary; No requirements with respect to the input tables
MERGE_JOIN	Joins by comparing the two input streams on the join attribute; Both inputs have to be sorted on the join attribute
INL_JOIN	For each tuple of the left input the join partners are retrieved by an index access on the right table Implies a correlation
P_TIMES	Returns the cartesian product of the two input tables
P_UNION	Returns the union of the two input tables

B.5 Rules used by the Model-M optimizer

Transformation Rules	
Name	**Description**
EQJOIN_COMMUTE	Join commutativity
EQJOIN_LTOR	Left to right join associativity
PROJECT_UP	PROJECT thru EQJOIN upwards
REMOVE_PROJECT	PROJECT Idempotence
DFRS_EQJOIN_COMMUTE	Duplicate_free join commutativity
DFRS_EQJOIN_LTOR	Duplicate_free left to right join associativity
DFRS_EQJOIN_RTOL	Duplicate_free right to left join associativity
DFRS_EQJOIN_EXCHANGE	Duplicate_free exchange of inputs on a subplan containing 3 EQJOINS

Implementation Rules	
Name	**Description**
GET_to_REL_SCAN	Implements GET as a full table scan
GET_to_REL_INDEX_SCAN	Implements GET as an index scan, if a corresponding index is available
SELECT_to_RESTRICTION	Transforms a logical relational selection into the corresponding physical operator
PROJECT_to_P_PROJECT	Transforms a logical relational projection into the corresponding physical operator
GROUP_BY_to_P_GROUP_BY	Implements grouping and aggregation
M_AGGR_to_P_AGGR	Implements aggregation on the entire input relation
M_UNIQ_to_P_UNIQ	Implements duplicate elimination is the input table is already sorted
M_UNIQ_to_P_SORT	Implements duplicate elimination by sorting the input relation, if the necessary ordering is not satisfied
M_UNIQ_to_P_HASH_UNIQ	Implements duplicate elimination via hashing
EQJOIN_to_HASH_JOIN	Implements a join by using a hash table
EQJOIN_to_MERGE_JOIN	Implements a join by merging
EQJOIN_to_INL_JOIN	Implements a join by using an index and a correlation
EQJOIN_to_P_TIMES	Implements a physical cartesian product
M_UNION_to_P_UNION	Implements a physical union of the input tables

Enforcement Rules	
Name	**Description**
Insert_Sort	Introduces a sort operator to realize the desired ordering

Appendix C
Proofs and Algorithms

C.1 Proofs related to the *MFSSearch* algorithm

Proof to Theorem 1: Obviously, for an itemset on level N all subsets of level $N-1$ are generated if its *ElimList* contains all N elements. It is not necessary to expand its prefix of length $N-1$ since it is implicitly evaluated through *ECS*. Hence, it is sufficient to include the first $N-1$ elements into the *ElimList*.Consider a superset X on level N with the *ElimList* $E_X = \{1,2,...,N-1\}$ and the sibling subsets (on level $N-1$) $X_1, X_2,..., X_{N-1}$. In this case, subset X_i is generated by eliminating item $N-i$ from X, s.t. $X_i = X - \{N-i\}$. In this case we will demonstrate that any item y in the *ElimList* of X_i, s.t. $y > N-i$, generates only a duplicate subset on level $N-2$.

This subset, named e.g. $X_{i,N-y}$, is obtained by eliminating y from X_i:
$X_{i,N-y} = X_i - \{y\} = X - \{N-i, y\}$.

On the other hand, there exists a sibling of X_i, called X_{N-y} such that $X_{N-y} = X - \{y\}$. Since $y > N-i$, results that X_{N-y} has been expanded before X_i and that the *ElimList* of X_{N-y} also contains $N-i$. Results that X_{N-y} has already expanded a subset $X_{N-y,i}$ on level $N-2$ such that $X_{N-y,i} = X_{N-y} - \{N-i\} = X - \{y, N-i\}$.

Thus, $X_{i,N-y} = X_{i,N-y}$ and $X_{i,N-y}$ is expanded after $X_{i,N-y}$. Results that $X_{i,N-y}$ is a duplicate.♦

Proof to Theorem 2: Assume that all itemsets are expanded. In the top-down search, this process is done from higher levels to lower ones. The total number of elements on level i is $\binom{i}{N}$. However, not all of these elements need to be expanded, as some of them have already been processed as prefixes in level $i+1$. Thus, the elements that need to be expanded on level i is $\binom{i}{N} - \binom{i+1}{N}$.

Hence, for N items, the number of expanded itemsets is given by:

$$\binom{N}{N}+\left(\binom{N-1}{N}-\binom{N}{N}\right)+\left(\binom{N-2}{N}-\binom{N-1}{N}+\binom{N}{N}\right)+...+\left(\binom{1}{N}-\binom{2}{N}+\binom{3}{N}+...+(-1)^{N-1}\binom{N}{N}\right)=$$

$$= \binom{N}{N} + \binom{N-2}{N}+...+\binom{1}{N}, \text{ for } N \text{ odd or } \binom{N-1}{N} + \binom{N-3}{N} +... + \binom{1}{N} \text{ for } N \text{ even} =$$

$$= 2^{N-1}.♦$$

Proof to Theorem 3: By successively reducing the size of the *ElimList* as described in the *Expand* procedure, the direct subsets of an itemset *X* are only those that have not been expanded before by another sibling. From the nature of top-down processing, the direct subsets of an itemset *X* will be processed after *X*. The cross subsets are related to siblings of *X* that are expanded before *X*. As in the backward processing siblings are processed in the opposite order than they are expanded, results that also these cross subsets of *X* will be processed after *X*. ♦

Proof to Theorem 4: For the domain *1,2,...,N-1,N*, the last itemset to be expanded is itemset *Z={N}*. However, *Z* is a subset of any itemset in the search space. ♦

Proof to Theorem 5: Assume $X_1, X_2 \in F$ and $X_1 \subset X_2$, From Theorem 3 follows that X_1 will be processed later than X_2. But X_2 is already frequent, from which results that it prunes X_1, so that X_1 cannot be also included in *F. Contradiction.* ♦

Proof to Theorem 6: As presented at the beginning of this section, the subsets of *Y* are obtained by eliminating elements of *E* from *Y*. If there is one element $y \in Y, y \notin X$, so that $y \notin E$, results that *y* will be present in all direct subsets of *Y*. Follows that all subsets of *Y* contain one element that is not included in *X*. Thus they cannot be subsets of *X*. ♦

Proof to Theorem 7: The direct subsets of *Y* are obtained by eliminating elements of $ElimList_Y$ from *Y*. From *X* is included in *Y*, and $ElimList_Y$ doesn't contain any elements from *X*, results that all direct subsets of *Y* also contain *X*. Since *X* is infrequent, these direct subsets can also be pruned. ♦

Proof to Theorem 8: Assume that there exists an itemset $Y=\{P_0, P_1,..., P_n, P_{n+1},...,N\}$ that is expanded after *X*. But $X \subset Y$, s.t. according to Theorem 3, *X* must be expanded after *Y. Contradiction.* ♦

C.2 The *MFSSearch* algorithm for the backward exploration scenario

MFSSearch *algorithm for a finite item domain 1,2,...,N*
1. X := {1,2,...,N}; E_X := {1,2,...,N-1}; $F_X := \emptyset$; $IF_X := \emptyset$,
2. get MIP, sup from ECS(X)
3. ***if*** *(sup < minsup)* // X infrequent, MIP = PMax
4. ***if*** *(|MIP| < |X| - 1)* // Condition (3)
5. $IF_X := F_X \cup MIP;$ // Propagate Relevant Infrequent Itemset

6. ***else if*** *(|MIP| = |X|)* // Condition (4)
7. ***return*** *X \ {N};* // X \ {N} Maximal Frequent Itemset
8. Expand(X, E_X, F_X, IF_X);

Expand*(Itemset X, ElimList E, FrequentSet F, InfrequentSet IF)*
1. ***for*** *each i = 1, n-1 (n = size of ElimList)*
2. *$X_i := X \setminus \{e_{n-i}\}$;*
3. ***if*** *($\exists F_X \in F, X_i \subset F_X$)*
4. *$X_i := \emptyset$;* // Cross Top-Down Pruning
5. ***else***
6. *$E_i := E \setminus \{e_{n-i}, ..., e_{n-1}\}$;*
7. *$F_i := \emptyset$; $IF_i := \emptyset$;*
8. ***for*** *each $F_X \in F$*
9. ***if*** *condition (1)* // Condition (1)
10. $F_i := F_i \cup F_X$; // Propagate Relevant Frequent Itemsets

11. ***for*** *each $IF_X \in I$, $IF_X \subset X_i$*
12. ***if*** *condition (2)* // Condition (2)
13. *$X_i := \emptyset$;* // Bottom-Up Pruning affecting also direct subsets
14. ***else***
15. *Mark X_i as infrequent;* // Bottom-Up Pruning affecting only the itemset
16. $IF_i := IF_i \cup IF_X$; // Propagate Relevant Infrequent Itemsets

17. ***for*** *each $X_i \neq \emptyset$, i := n-1, 1* // Backward Exploration
18. ***if*** $\neg Infrequent(X_i)$
19. *get MIP, sup from ECS(X_i)*
20. ***if*** *(sup < minsup)* // X_i infrequent, MIP
21. ***if*** *(|MIP| < |X_i| - 1)* // Condition (3)
22. *BottomUp (MIP);*
23. ***else if*** *(|MIP| = |X_i|)* // Condition (4)
24. ***return*** *X_i \ {N};* // X_i \ {N} Maximal Frequent Itemset
25. *Expand(X_i, E_i, F_i, IF_i);*
26. ***else***
27. *CrossTopDown(X_i);*
28. ***return*** *X_i;* // Maximal Frequent Itemset
29. ***else*** *Expand(X_i, E_i, F_i, IF_i);*

CrossTopDown *(Frequent_Itemset X)*
1. ***for*** *each candidate Y, Y expanded but unexplored*
2. ***if*** *condition (1)*
3. $F_Y := F_Y \cup X$;

BottomUp*(Infrequent_Itemset I)*
1. ***for*** *each candidate Y, Y expanded but unexplored*
2. ***if*** $I \subset Y$
3. ***if*** *condition (2)*
4. $Y := \emptyset$; *// prune Y together with its direct subsets*
5. ***else***
6. *Mark Y as Infrequent;* *// prune only Y*
7. $IF_Y := IF_Y \cup I$; *// propagate I to be taken into account for subsets of Y*

As can be seen from *MFSSearch*, the procedure *Expand* is used to address both pruning and search strategies within the given search space.

C.3 The Expand procedure adapted to the forward exploration scenario

Expand*(Itemset X, ElimList E, InfrequentSet IF)*

1. **for** each i = 1, n-1 (n = size of ElimList)
2. $X_i := X \setminus \{e_{n-i}\}$;
3. $E_i := E \setminus \{e_{n-i}, ..., e_{n-1}\}$;
4. $IF_i := \emptyset$;
5. **for** each $IF_X \in I$, $IF_X \subset X_i$
6. **if** condition (2)
7. $X_i := \emptyset$; // Bottom-Up Pruning affecting also direct subsets
8. **else**
9. Mark X_i as infrequent; // Bottom-Up Pruning affecting only the itemset
10. **if** $I \subset PMax_{X_i}$ // prune also PMax of X_i
11. Mark $PMax_{X_i}$ as Infrequent;
12. $IF_i := IF_i \cup IF_X$; // Propagate Relevant Infrequent Itemsets
13. **for** each $X_i \neq \emptyset$, i := 1, n-1 // Forward Exploration
14. **if** $\neg Infrequent(X_i)$ // probe X_i
15. get MIP, sup from ECS(X_i);
16. **if** (sup < minsup) // X_i infrequent, MIP = PMax
17. BottomUp (MIP);
18. **if** (|MIP| = |X_i|) // condition (4)
19. UpdateMFS($X_i \setminus \{N\}$); // $X_i \setminus \{N\}$ Frequent Itemset
20. Expand(X_i, E_i, IF_i);
21. **else**
22. UpdateMFS(**(**X_i); // X_i Frequent Itemset
23. **else**
24. **if** $\neg Infrequent(X_i - \{N\})$ // probe PMax of X_i
25. get MIP, sup from ECS($X_i \setminus \{N\}$);
26. **if** (sup > minsup)
27. UpdateMFS(**(**$X_i \setminus \{N\}$); // Xi \ {N} Frequent Itemset
28. Expand(X_i, E_i, IF_i);

UpdateMFS*(Frequent_Itemset X)*

1. **for each** $Y \in MFS$
2. **if** Y is a subset of X
3. eliminate Y;
4. $MFS := MFS \cup X$;

BottomUp*(Infrequent Itemset I)*

1. **for** each candidate Y, Y expanded but unexplored
2. **if** $I \subset Y$
3. **if** condition (2)
4. Y:= $\emptyset$; // prune Y together with its direct subsets
5. **else**
6. Mark Y as Infrequent; // prune only Y
7. **if** $I \subset PMax_Y$
8. Mark $PMax_Y$ as Infrequent; // prune also $PMax_Y$
9. $IF_Y := IF_Y \cup I$; // propagate I to be taken into account for subsets of Y

Appendix D
Auxiliary Lists

D.1 List of Figures

D.2 List of Tables