Dynamic Query Processing in a Parallel Object-Oriented Database System

Promotie commissie:
Prof. dr. P.M.G. Apers, promotor
Prof. dr. M.L. Kersten, co-promotor
Prof. dr. ir. N.J.I. Mars, secretaris
Dr. H.M. Blanken
Prof. dr. L.O. Hertzberger
Prof. dr. S.J. Mullender
Dr. P. Valduriez, INRIA-Rocqencourt

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

van den Berg, Carel Arie

Dynamic query processing in a parallel object-oriented database system / Carel Arie van den Berg. - Amsterdam : Stichting Mathematisch Centrum. -Ill.
Thesis Enschede. - With index, ref. - With summary in Dutch.
ISBN: 90-6196-434-2
NUGI 852
Subject headings: parallel database systems.

Cover: (Man carrying a load up the mountain to find new horizons)

Dynamic Query Processing in a Parallel Object-Oriented Database System

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. Th. J. A. Popma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 25 februari 1994 te 13.15

door

Carel Arie van den Berg

geboren op 14 augustus 1963
te Bergen

Dit proefschrift is goedgekeurd door:

Professor P.M.G. Apers, promotor.
Professor M.L. Kersten, assistent-promotor.

Voor mijn Vader

Dankwoord

Ik draag dit proefschrift op aan mijn vader die te vroeg gestorven is. Hij heeft bij mij al op jonge leeftijd mijn interesse gewekt voor de wetenschap en is door de jaren heen mijn voorbeeld geweest. Ik wil bij deze ook stilstaan bij de inspanningen die mijn moeder zich voor het gezin getroost heeft.

Drie jaar geleden na het aflopen van het PRISMA project begon ik met het in dit proefschrift beschreven onderzoek. Na ruim twee jaar had ik nog een lange weg te gaan om de resultaten op schrift te stellen. Hoewel dit voornamelijk een solistische onderneming is had ik het niet kunnen volbrengen zonder de directe of indirecte hulp van velen. Ik dank hun allen voor hun bijdrage. De bijdrage van een aantal van hen wil ik hier speciaal noemen.

Na mijn afstuderen nam Martin Kersten mij aan bij het CWI voor het PRISMA-project. Hij heeft vanaf het begin zijn vertrouwen in mij getoond en is al die jaren nauw betrokken geweest bij mijn onderzoek. Verder zijn onze discussies voor mij een bron van inspiratie geweest.

De brede opzet van het PRISMA project was een unieke gelegenheid om mijn kennis van vele deelgebieden van de informatica te verdiepen, vooral op het gebied van parallelle database systemen. Ik heb met veel genoegen samengewerkt met de leden van de PRISMA-groep. Ik wil met name Peter Apers, Jan Flokstra, Paul Grefen, Maurice Houtsma, Erik van Kuijk, Rob van de Weg en Annita Wilschut van de PRISMA/DB groep van de Universiteit Twente noemen.

In 1990 heb ik samen met Satish Shair-Ali op het Philips Natlab de eerste experimenten verricht met dynamische query verwerking op de PRISMA machine. De samenwerking met Satish was bijzonder prettig en de resultaten waren doorslaggevend voor de keuze van mijn promotie onderwerp.

I thank Patrick Valduriez for agreeing to be a member of my thesis committee and for taking the trouble to travel from Paris to participate in my thesis defense. Furthermore, I would like to thank Cesar Galindo for his comments on the manuscript.

Ik bedank de leden van de CWI database groep voor hun reacties en opbouwende kritiek op het onderzoek. Mijn vrienden hebben mij in deze drukke tijd geholpen door mij op gezette tijden van mijn werk los te rukken.

De bijdrage van jou, Sita, is niet in woorden uit te drukken. Je hebt mij telkens weer aangespoord om het proefschrift af te ronden wanneer ik in de verleiding kwam om het onderzoek verder uit te breiden. Ik dank je voor je liefde, geduld en voor de vele dingen die ik van je leer die de wetenschap ontstijgen.

Carel van den Berg, 11 januari 1994

Contents

Chapter 1

Introduction

Timely information has become increasingly important for today's competitive businesses. Furthermore, it has proven to be a valuable commodity. The number of information services or business relying on timely information is growing fast. It is reported that the data volume is increasing by 25-35% per year, while at the same time the amount of data stored per person is increasing[Gro90]. Moreover, the massive amount of data is not only used for simple data intensive applications, but it is also used to extract information by relating the data stored. This leads to an increase in query complexity and data complexity.

These demands foreseen cannot be easily met using traditional disk- based data-base technology, because the I/O bottleneck forms a physical limitation to improve responsiveness. Disk technology has shown an improvement of only a factor 2 over the last 10 years in response time and throughput. It is unlikely that this will change dramatically in the near future, because the rotational speed of a disk meets its physical limit.

In contrast the CPU speed has been doubling every year. Furthermore, the break-even point for storage cost for main-memory compared to disk is expected to be reached within the next two decades [Gib91]. The influence of new solid-state memory technology, e.g. flash memory, could lead to an earlier transition from disk storage.

These developments have led to research in parallel data-base systems consisting of a large number of off-the-shelf, and ,therefore, cheap processors. The processors are interconnected by a high-speed network. Typically each processor is equipped with a large amount of memory and a disk. By declustering the data over the available processors data can be accessed in parallel, leading to an improved response time. A further improvement can be obtained if the hot-set

of the data can be kept in main memory. These systems are commonly known as main-memory data-base systems.

Examples of commercial parallel data-base systems are Teradata DBC/1012 [Pag92] and Tandem's NonStopSQL [Gro87]. Recently, the Esprit EDS prototype [WT90] has been developed into a commercial product, called the Goldrush machine. Research prototype systems are Bubba [Bea90], Gamma [DGS+90], and PRISMA [AKW+92].

The relational data model upon which these systems are based is well understood and has proven to be both cost effective and efficient to support simple administrative and business applications. However, the data model is not rich enough to support scientific or non-standard applications. This is exemplified by the requirement for multi-media data bases [HRD93, vdBvD93] to support hyper-text structures, video, and audio data types and in geographic information systems (GIS) [KvdB91] by the demand for efficient support and access to multi-dimensional data structures. This mismatch between the data structures used in the application program and the data-base data model is commonly known as the *impedance mismatch*.

Object-Oriented Data Base Systems (OODBMS) reduce the impedance mismatch by offering a rich data model and the ability to specify operations on user-defined complex data types. The research on OODBMS has resulted in a large number of prototypes and commercial systems. Examples of these systems are O_2 [Dea90], Versant, ObjectStore, and ORION [KBGW90]. These systems have been developed independently and, consequently, do not share a common data model. Only recently the OMG consortium has been formed to specify a common interface and data model resulting in the ODMG data model [Cat93].

The baseline for data-base-system research is the development of efficient, and effective systems to support today's applications. Therefore, the combination of both research issues, parallel query processing and OODBMS, seems justified to attain this goal. Moreover, few research efforts have addressed the design of a parallel OODBMS or Complex Object Server [Tee93].

This thesis is a monograph on parallel query processing in a main-memory data base. Specifically, the application of parallel query processing in an OODBMS is examined. The combination of parallel query processing and OODBMS increases the problem of efficient query processing. The main topic of this thesis is to explore a novel query processing architecture which employs dynamic (adaptive) query processing techniques to improve the performance of a parallel data- base system. The discussion addresses many issues of data-base systems that have a strong impact on performance: the object storage model, query optimization, load balancing, and buffer management.

Before we can state our research goal more precisely we have to introduce the parallel query processing issues to explain the shortcomings of current parallel query processing architectures. We then present the outline of this thesis.

1.1 Issues in parallel query processing

In this section we give a short overview of the factors that determine the performance of a parallel data-base system. We assume the reader has a basic understanding of the relational model and relational algebra. We concentrate

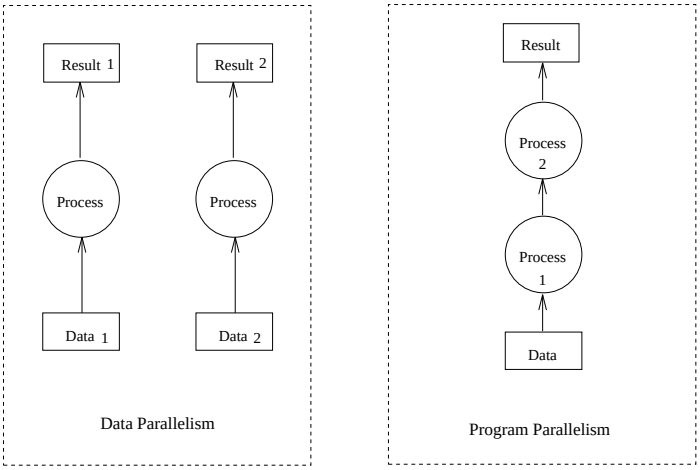


Figure 1.1: Two forms of parallel execution

therefore on query processing. First, we make a distinction between two types of parallelism, and introduce the techniques used to exploit them. Next, we give an overview of query processing in a parallel data-base system. Finally, we sketch the problems inherent in the current architectures.

1.1.1 Parallel query execution

In parallel data-base systems two orthogonal types of parallelism can be found: *data parallelism* and *program parallelism* [PMC+90]. The difference is illustrated in Figure 1.1.

Data parallelism

Data parallelism is obtained by *data partitioning* or *declustering*. In this technique the tuples in a relation are divided in sub-sets or fragments. The number of fragments is termed the *partitioning degree*. In distributed data-bases this technique is also known as *horizontal fragmentation*. By allocating the fragments on different processors or disks, they can be accessed in parallel. The basic idea is that the response time of an operation (query) is reduced by executing it on the (smaller) fragments in parallel. Consider for instance a select operation. If the operand relation is partitioned and allocated on different processors, a range select operation can be executed on all the relation fragments in parallel leading to a reduced response time. In *dynamic partitioning* the relations or intermediate relations are partitioned as part of the query process to introduce parallelism. This technique is effective for calculating expensive operations like the join operation. The invested partitioning overhead must be justified by the reduced response time for calculating the join operation in parallel.

An important issue in parallel data-base systems related to data partitioning is *data placement*. This has a great impact on the system load distribution. It

is better to avoid data transport by bringing the operations to the data than by bringing the data to the operations. If two fragments are frequently used together it is wise to place them on the same memory. The placement depends therefore on the workload. A corollary of this is that if the workload shifts, the data placement must be adjusted, preferably at run-time without interrupting running applications.

To increase the data availability in view of processor failures or concurrent running queries, data can be replicated. The *data replication* technique maintains copies of a fragment on different processors. This brings up the additional problem for keeping the replicas in a consistent state. For this purpose *replica control* algorithms have been designed. Maintaining replicas further complicates the data placement task.

The influence of these three techniques *data partitioning*, *data placement* and *data replication* on the system performance is difficult to predict and depends on several factors: the partitioning degree, workload, and replication degree.

Program parallelism

Program parallelism is obtained through *query decomposition*. A query is split in sub-queries which are executed in parallel in a producer-consumer relationship. This technique is also known as pipeline parallelism.

The advantage of this scheme is that intermediate results produced in a query pipeline do not have to be stored. Instead they are temporarily maintained in a buffer between two sub-query processes. To keep this buffer small it is important that the rate at which one process produces an intermediate result equals the rate at which the other processes the data. The effect of data flow execution on query response time and processor load under ideal circumstances has been studied in detail by Wilschut [WA91, WFA92, Wil93]. A general cost model, which can accurately predict the response time for a pipelined query is still a research issue.

The allocation of sub-queries on processors has, similar to data placement, a major impact on the query performance. This allocation is not only determined by the placement of the fragments or intermediate results required by the sub-query, but also by the expected CPU cost for computing a sub-query and the actual processor load. The load distribution is not fixed during query evaluation. Consequently is *load balancing*, in a static pipelined query structure difficult to achieve.

1.1.2 Query processing overview

The query process is divided into three different stages: *query translation*, *query optimization*, and *query execution*. This is illustrated in Figure 1.2, which shows a typical parallel query processing architecture.

In the *query translation* phase the query is translated from its textual representation into a canonical internal representation. In the process the query is syntactically and semantically analyzed using the data-base schema. Many internal representations are possible, but mostly a query is translated into a

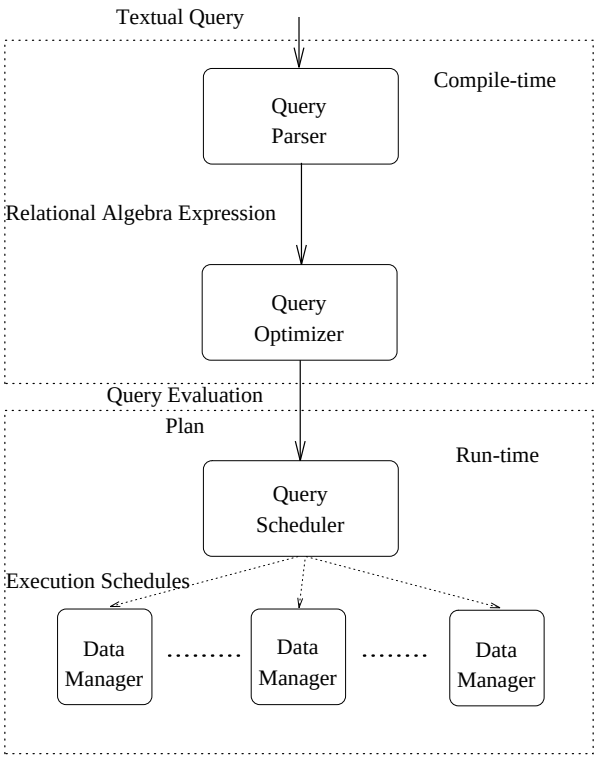


Figure 1.2: A typical parallel query processing architecture

relational algebra expression, for example eXtended Relational Algebra [WG89] was used in PRISMA and LERA in the EDS project [CBSB92].

A possible approach to *parallel query optimization* is to separate it into a logical optimization part and parallel optimization part. This approach is based on the assumption that decisions in the logical optimization phase and parallel optimization are independent. This assumption is not valid, but at least it reduces the complexity of parallel query optimization[HS93].

In the *logical optimization phase* the internal representation is transformed into a *query evaluation plan* (QEP). In this transformation the query optimizer uses rewrite rules based on the algebraic properties of the relational operations to generate alternative, but semantically equivalent, QEPs. The QEP is commonly represented by an operator tree, where the nodes in the tree represent the relational operations and the edges a data dependency. Examples of these logical optimizations are changing the join-order and pushing selections and projection down the operator tree. A *cost model* is used to select between the alternative query plans and to limit the search space.

For a given query many possible QEPs exist. This number grows, roughly speaking, exponentially with the number of primitive operations in the query. The query optimization cost can therefore be significant and must be taken into account.

In the *parallelization phase* the optimizer uses the data partitioning and query decomposition technique to produce a parallel QEP. The operators in the operator tree are assigned to processors, on the basis of the data allocation and sub-query cost. In XPRS [SKPO88] the parallelization phase is performed at query start-up time to obtain the best possible load balance.

In the *query execution phase* the QEP is put into execution by a query scheduler. The *query scheduler* assigns the sub-queries to the processors, sets-up the communication structure for the pipelined sub-query processes, acquires read/write locks for the accessed fragments, and starts up the processing.

1.1.3 Research problem and objectives

It is the task of the Query Optimizer to select a QEP, that ideally results in a minimal response time and that uses the system resources efficiently. The sheer number of possible query plans prohibits an exhaustive search so that the selected query plan is not likely to be the optimal plan.

At query compilation time the Query Optimizer uses cost formulas and database statistics to generate and evaluate alternative QEPs. Because the final QEP is produced before the query runs on the data base, we refer to this technique as *Static Query Processing* (SQP). Static Query Processing has two inherent problems, which are related to the timing of query optimization. These problems are related to data skew and sub-query allocation.

- The QEPs are based on size estimates of intermediate results. These estimates are error prone due to the statistics being maintained and skewed data. The error culminates with query complexity. A suboptimal query plan results.

- The allocation of sub-queries to processors is timed at query startup time assuming it runs in isolation. As the processor load continuously changes due to eg. concurrently running queries, this leads to processing bottlenecks in query pipelines and, therefore, often to an underutilized system.

When some of the optimization decisions are delayed until at run time more information is available to improve the optimality of the QEP at the expense of run-time optimization overhead. This approach is called Dynamic Query Processing (DQP).

In this thesis we propose a dynamic query processing architecture for a parallel main-memory OODBMS running on a shared-nothing multi-processor system. Our claim is that this query processing technique obtains a better efficiency and lower query response times than the traditional techniques, because it is designed to adapt the query execution to the current status of both the system resources (CPU and memory), as well as the query behavior itself. Especially in view of the increase in query complexity, data complexity and information volume we think that this scheme facilitates and improves parallel query optimization.

The research is mainly focussed on the following questions:

- How do you reduce the effect of data skew and load distribution in a parallel DQP architecture? Delaying optimization decisions to effectively reduce these effects introduces run-time overhead. Therefore, a common approach is to limit the number of run-time optimizations. An alternative approach is taken in this thesis. We want to design a query processing architecture where run-time optimizations can be performed cheaply such that a QEP can be adjusted frequently taking the current load distribution and the actual intermediate result sizes into account.

- How do you parallelize an OODBMS? To answer this question we first need to fix an object model and its query language. Once this is done we must decide on an object storage model that allows the exploitation of data parallelism and program parallelism.

We approach these research questions by identifying the key processes in the query processing architecture and evaluating their performance through experiments using mathematical models, simulation models or prototype implementations. We have selected a multiprocessor system running the Amoeba operating system [MvRT+90] and a network of SGI workstations as target platforms.

1.2 Thesis outline

This thesis is organized in the following chapters.

Chapter 2 discusses the issues and criteria that influenced the design of the Goblin OODBMS. The main observation made in this chapter is that a dynamic query processing architecture should be used. Other boundary conditions are that the design should be based on a main-memory shared-nothing architecture.

The Goblin data model and query language is presented in Chapter 3. The data model is mainly introduced to make the thesis self-contained.

In Chapter 4 we discuss the Goblin (distributed) storage model and compare three alternative object representation models. On the basis of the Goblin data model and the boundary conditions we have selected the Decomposed Storage Model.

After the storage model the dynamic query processing scheme is presented in Chapter 5. It discusses two alternative approaches to dynamic query processing. One based on data partitioning and another based on query decomposition. From these two the data partitioning approach is used because it facilitates load balancing and at the same time introduces parallelism.

The Goblin query processing architecture is presented in Chapter 6. Followed by Chapters 7 to 10, which discuss task generation, task elimination, task allocation, and task evaluation, respectively. The task generator drives the query evaluation process. Given a query and a partitioned data base it generates tasks that evaluate the query on a part of the data base. The task elimination technique is a dynamic query optimization technique, which reduces the amount of work using feedback information on the query process. The task allocation algorithm and the task evaluation algorithm, finally, decide *where* a task is executed and *how* a task is executed, respectively.

On the basis of a partial implementation of Goblin we have measured the performance of its key algorithms. The results are combined to predict the performance of the completed prototype. The results of these experiments can be found in Chapter 11.

Finally, in Chapter 12 we give a summary of the main results and indicate areas for future research on this architecture.

Chapter 2

The Goblin OODBMS

2.1 Introduction

The Goblin OODBMS presented in this thesis illustrates the issues involved in parallel data-base systems in general and query processing in particular. The main design issue for Goblin is its dynamic query processing architecture. Its (partial) implementation is used to test the efficiency of the processing scheme. This chapter addresses issues stemming from the application domain, technological trends and software engineering. These result in a list of design criteria to guide the design and implementation of the system.

Goblin is an experimental system focussed on applications with large volumes of data and that require a rich type structure. Astronomy, robotics, CIM, or geography [Ker91, KvdB91] are examples of such application areas. Moreover, the Goblin design takes into account the characteristics of its envisioned application domain, the hardware trends, and the deficiency of the current DBMS designs to effectively exploit parallelism.

The design of a parallel OODBMS involves many issues. Apart from the underlying parallel platform, it ranges from the programming language, data model, storage model to the query processing architecture. In this Chapter our observations are summarized by design criteria for the Goblin programming language, its query processing architecture, and the data representation.

The boundary conditions for the Goblin project are set by its application domain, current trends in technology, and the standard software engineering criteria, such as flexibility and modular design. These considerations have helped to formulate from the onset a set of design criteria for the Goblin system. The influence from application, technology and software engineering are discussed to

a limited extent in the next sections.

2.2 The application domain

2.2.1 The impedance mismatch

Integration of data bases into larger software systems has the effect that data bases are accessed more by application programs than by interactive users directly. New, complex applications do not primarily access data-base systems through their 4GL[1] interface. The interface requirements for application programs are different from those offered by 4GLs of relational data-base systems. This results in an interface problem, commonly termed as *impedance mismatch*, signifying the loss of power in the interface. In this context the impedance mismatch refers to overhead incurred by the conversion of data in the application data-base interface. This is caused by the semantic incompatibility between the data-base model and programming language model.

The impedance mismatch can be reduced if DBMS and application language are based on the same type system and storage model. This goal can be achieved by extending an existing programming language with data-base capabilities through new language constructs or through libraries. Examples of these systems are GemStone (SmallTalk), Exodus (C) and Ontos (C++), which have added constructs for persistency, transaction management and concurrency control to the language.

An alternative and more radical approach is the design of a completely new data model and data manipulation language (Galileo). The main disadvantage of this approach is that it complicates the integration with existing software packages and requires the application programmer to learn yet another programming language.

In the hybrid approach the data-base programming language is embedded in the application language. This approach is taken for instance by O_2. It provides a SQL-like language for specifying data-base queries. These queries are included in the application program written in C or C++. A preprocessor replaces these embedded queries by calls to the data-base system.

In Goblin we take a significant subset of C[2] and provide a clear interface with DBPL primitives. This is a hybrid approach. The benefit is that, because exploitation of parallelism is one of the key issues in Goblin, a data-base programming language is developed that contains easily parallelizable constructs for data manipulation, which can be embedded in an application program. Therefore to reduce the impedance mismatch we arrive at the following requirement for the application language interface.

Criterion 2.1 The language should have a type system compatible with its predominant application programming language.

[1] fourth generation language e.g. SQL

[2] C++ was considered, but initially rejected due to its baroque language constructs. We may come up with a compatible syntax subset for C++.

2.2.2 Extensibility

In some cases a rich type system is not sufficient to support an application efficiently. Multi-media applications, for instance, store and manipulate video and audio data and often require special hardware for compressing, uncompressing and displaying the data. Furthermore, the resource requirements for audio and video are enormous and could impede the data-base performance if not handled well. These applications are better supported if the set of base types can be extended and the basic operations on these types can be defined.

Essential to extensible data-base systems is the ability to support new user-defined types effectively. This implies support at the language, optimization and access level of the DBMS. Examples of such systems are, for instance, Postgress [SRH90] and EXODUS [CDRS86] and Gral [BG89]. These systems can, by their modular design, easily be adapted to a changing hardware environment and to changing requirements of an application domain. Adding a new type mainly requires the definition of its optimization rules and the definition of a few basic storage- and access functions.

Consider, for instance, an application that manipulates images. By adding the image type with its operations, a set of rewrite rules and cost functions, the general query processing and optimization mechanism of the data-base system should be able to process queries against image types [BG89].

Therefore, we arrive at the following criterion:

Criterion 2.2 The language should support a facility for defining abstract data types including cost functions and optimization rules for the optimizer.

2.3 Software engineering

2.3.1 Language design

The same design criteria as for general programming languages are applicable to a modern DBPL, viz. simplicity, expressiveness, orthogonality and definiteness. Simplicity enables a user to master the language in a short period of time. The language should offer only a limited set of constructs that enable the programmer to express a concept in a single way.

Criterion 2.3 Goblin should have a limited set of orthogonal programming language features.

Definiteness means that language semantics and syntax are clearly defined. A concise and small formal language definition assists a novice or language implementor. Currently a lot of research effort is put in providing the field of OODBMS with a sound theoretical basis. In the design of the data model we were influenced by work of Cardelli [Car84] on subtype hierarchies. The general language flavor is borrowed from O_2.

It would be outside the scope of this thesis to attempt this exercise for Goblin. We will therefore only stress its importance in the following criterion and leave it at that.

Criterion 2.4 The Goblin language syntax and it semantics should be correctly and clearly defined.

2.4 Technological trends

2.4.1 Multiprocessor systems

In high-performance systems there is a trend towards MIMD architectures consisting of large numbers of off-the-shelf (cheap) processors. To exploit these architectures, Goblin focuses purposely on handling (large) collections of objects, because it is the operation on bulk data with order-independent semantics that has proven to be a decisive factor in the exploitation of parallelism. Fast navigational access to a single object is a focus of (more) efficient OOPL implementations. The primitive operations, in which user queries are transformed, operate therefore on sets of objects.

Criterion 2.5 The Goblin primitive operations should be set oriented to avoid message handling becoming a dominant processing factor.

Efficient resource management in a multiprocessor system includes the allocation of work to processors. Evidently, this requires information on load distribution that is only available at run-time. This is formulated in the following criterion:

Criterion 2.6 Goblin should have a dynamic query processing scheme to exploit available CPU resources effectively.

2.4.2 Main memory

The storage cost expressed in \$/byte for main-memory is declining faster that for magnetic disks. If the current trend continues, the break-even point will occur in the year 2015 [Gib91].

Therefore, the Goblin architecture focuses on loosely coupled multiprocessors with sufficient (combined) main memory to keep the data-base hot set and the intermediates for query processing memory resident. This assumption continues the thread started in 1986 with PRISMA[KAM+87, AKO88] and aligns with market expectations for cheap computer systems with an abundance of main memory.

The validity of this assumption is illustrated by the EDS project. The data-base system developed in the EDS project focuses on On-line Transaction Processing (OLTP) and Decision Support Systems (DSS). One machine is a shared-nothing architecture, consisting of processor clusters interconnected through a high performance network. Each cluster contains 8 - 64 processors, where each processor has 64 Mbytes of stable main memory. The other machine, DBS3 [BCV91], is a shared-memory machine, which is also equipped with a large main memory.

Criterion 2.7 Goblin should be designed as a main-memory parallel data-base management system.

2.4.3 Physical design

Parallel data bases add to the complexity of physical design. Traditionally, physical data-base design includes identifying attribute indices, the storage structure

to be used, and how the relations should be clustered. Input to data-base design are the predominant queries, estimated data-base size and the data-base schema. More information on physical data-base design can be found in [TF82].

Furthermore, in a distributed system query processing implies finding a balance between data transport and query allocation. At the same time, exploiting parallelism requires data replication. As access patterns change over time, an optimal physical data organization, is likely to be optimal for only a short period of time. In the ideal situation, the data storage is self-organizing with respect to indexing, clustering and replication. However, a sufficient body of experience from both theory and systems is lacking to support this goal. Therefore, any investment in building a new system should be prepared to investigate new adaptive techniques.

Criterion 2.8 The Goblin object storage and indexing scheme should facilitate the use of adaptive techniques.

2.4.4 Operating systems

In Goblin we expect parallelism to be exploited effectively in query dominant environments with local updates. This means that portions of the object space can be replicated on demand, while their contents is kept synchronized using a read-one-write-all protocol using the facilities of the underlying operating system.

A large part of current DBMS systems provide solutions to problems that should be ideally handled by the operating system. In Goblin we do not aim for a (re-)implementation of basic distributed operating system facilities, such as memory-mapped files, persistent storage, network management, concurrency control primitives, and caching. Instead, we focus on mechanisms to advise a cooperative distributed operating system on how to optimize its resource allocation. Such facilities are readily available in the latest operating system kernels [JR86]. For instance, the ability to pin or unpin pages in memory or to give page buffering advice.

Criterion 2.9 Goblin relies on the (future) availability of advanced distributed operating systems, to offer globally accessible persistent data, concurrency-control primitives and cache-control primitives.

2.5 Conclusion

The Goblin design is influenced by issues and developments from three different domains: the technology, the application domain and software engineering. We have selected from each of these domains a few issues and formulated design criteria based on the observations.

Of these domains the application domain and technology have the greatest impact on the design. Basically, the application domain made us decide to develop an object oriented data base. The technology has lead us to implement it as a main-memory shared-nothing data-base system.

In the following chapter we will give a flavor of the Goblin object oriented language, which provides the baseline of this thesis. The rest of this thesis is primarily concerned with the design and implementation of the Goblin OODBMS.

Chapter 3

The Goblin Query Language

3.1 Introduction

Effective use of parallelism for query processing does not only pose constraints on the underlying hardware and software architecture, but also puts requirements on the data model and query language.

For instance, the concept of message forwarding in object oriented data models possibly impedes efficient execution. It forces the execution model to handle objects on an individual basis. Our data model is chosen such that it can be easily translated into a relational model. The associated relational operations are set oriented and can be efficiently implemented for handling bulk data.

Furthermore, the language should provide constructs that can be parallelized. It is difficult to parallelize a query that uses control flow statements together with globally shared variables to access data, because it fixes the order in which the objects are accessed. A non-procedural, declarative query specification offers more opportunities for parallelization, but also complicates the programming of some inherently sequential processes. The goal is finding a reasonable balance between imperative constructs and declarative programming.

In the following sections the main issues in the design of the data model and query language are presented to the level required to understand the subsequent chapters.

3.2 Object-oriented data-base concepts

The research community has not yet reached an agreement on the features and characteristics that should be supported by an object-oriented data-base system. This is illustrated by the wide variety of object-oriented database systems.

Consider, for instance, the Postgress and GemStone systems. The Postgress system is an example of a relational system extended with object-oriented features, while the GemStone system can be seen as an object-oriented language system extended with database functionality, like persistency, transaction management and query facilities.

In an attempt to solve these differences in 1989 a group of researchers compiled a statement on the features an object-oriented data-base system should at least support [ABD+92]. Only, recently a consortium of industry, the OMG group, has started in defining a common standard [Cat93]. To be self-contained, and because our data model is slightly different, we introduce the basic concepts in the following sections.

3.2.1 Complex objects

Being focussed on the object-oriented paradigm, the notion of *object identity* is an essential part of the model. Objects represent entities in the Universe of Discourse (UoD). Once an object becomes a part of the UoD, it has a unique identity, which allows it to be distinguished from other objects in the UoD.

Complex objects are modeled using *object attributes* and *object methods*. The *object attributes* describe the *state* of an object in terms of its relationships with other objects (or values), while the *object methods* specify the *behavior* or *state transitions* of the object through procedural abstractions. The object methods also provide the interface with the application program and the data base.

During its (UoD) lifetime the number of relationships and the properties of relationships can change. Consider, for instance, an object that represents a person. Once a person marries, the object establishes a 1-1 relationship with another person object. Their children can be naturally represented by 1-N ary relationship to both parent objects.

The behavior of an object, i.e., the methods an object can execute, can also change during its life-time. For instance, if a person gets a job he becomes an employee. Consequently, the associated object will be able to answer all the methods that are applicable to employee objects.

The key observation, elucidated by this example is that both the relationships an object participates in and its behavior are highly dynamic properties determined mostly outside the context of any computerized system. Conversely, a computerized system can not preclude the relationships of the object in the UoD. They are not fixed during schema design and the database should be able to handle exceptions to the rule invented at design time.

3.2.2 Classes

To simplify object handling in a computerized system and real-world information system, they are grouped into *classes* based on commonality of their relationships and their behavior. A class has an unique name, a *class specification*, and a *class extent*.

The class specification specifies constraints on the structure, behavior, and state, that hold for objects in the class, or it serves as a template, when an object is created, specifying the minimal set of attributes and applicable methods.

In addition, the class extent is the set of objects that minimally satisfy the constraints from the class specification. The objects in the class extent can be manipulated by applying methods on all objects in the class extent, or selecting a subset by querying the class extent.

There are two approaches to populate the class extent: explicitly or implicitly. In the *object factory*[ABD+92] approach objects are created by applying a *new* operation to a class, which results in adding the object to its class extent. Once created, the object remains in the class extent until it is explicitly destroyed. In the *object taxonomy* or implicit approach, the class specification is used as a shorthand to create an object with a certain set of properties, because the same object can be created by adding attributes and methods individually; the class extent is dynamically populated. Objects in the database that have the minimal properties required by a class are added automatically to its extent. If properties are dropped, the object is possibly removed from the class extent.

The most significant difference between the two approaches is that in the object factory approach it is possible to define two classes having the same class specification, but with non-intersecting class extents. In other words, the class extent is used as a set mechanism, which is often available in the same data model as a separate constructor as well.

In our opinion this is an improper combination of concepts. If there is a need to distinguish between objects by structure or state, the distinguishing property should be named and added to the class specification. This aligns with the approach in natural sciences where objects are classified implicitly on the basis of the objects they are related to, and their behavior. In Goblin we follow a similar approach.

3.2.3 Inheritance and class hierarchy

Inheritance helps a data-base designer in factoring out (shared) specifications. Furthermore, it leads to a concise description of the UoD.

A subclass can be defined by adding methods, constraints or attributes to a class already defined. The specialized class inherits the attributes and methods of the old class. Constraints over objects in the old class will also hold for the objects of its subclass. This kind of inheritance is also called inclusion inheritance [Car84].

For instance, if the class *Person* with attributes *name* and *spouse* and methods *marry* and *die* is defined, then the new class *Employee* can be defined to be a specialization of the *Person* class with the *ISA* construct: *CLASS Employee ISA Person*. This construct specifies that the *Employee* class has at least the attributes and methods defined for the *Person* class. The attributes defined in the remainder of the specification are added to this set of inherited properties.

Together with their inheritance relations the classes form a *class hierarchy*. This hierarchy is essentially an object taxonomy, which distinguishes and groups objects on the basis of their properties. The top of the class hierarchy is formed by the most general class: OBJECT. If an object looses all its properties or does not satisfy any of the class specifications, it is still a member of this class.

3.2.4 Objects versus values

An essential feature of an object-oriented data-base system is to simplify modeling complex objects. A common approach is to provide a set of base types (eg. *int, string, float*) and a collection of type constructors, like *tuple*, *set*, *list* and *array*. A type defines, set-theoretically speaking, a subset of all possible values. For instance, the type INT contains all integer values. An integer variable refers to any element in the integer domain. Objects are commonly defined as a pair consisting of an object identifier and a (structured) value.

An alternate viewpoint is to consider the base types and type constructors to be base classes and constructor or parameterized classes, respectively. The main distinction between the two approaches is that the first approach supports both values and objects, while the second approach only supports objects.

In the early working documents of Goblin [Ker91] a distinction was made between *objects* and *values*, similar to the approach taken in O_2. In [Bea88] this choice is mainly driven by engineering considerations.

A point raised to advocate the support of both values and objects is that it is cheaper to update privately owned objects. Moreover, in the implementation of a programming language, the detection of privately owned objects plays an important role in the optimization of updates to private objects. In a database context, however, the problem lies not in the optimization of updates of, or access to *single* objects, but in the optimization of queries on *large sets of similar* objects. In Goblin, to support efficient content-based data access, the physical representation does not necessarily reflect the logical representation. Therefore the need to recognize private objects for query-optimization purposes largely disappears.

Furthermore, duplicate elimination is cheaper if both values and objects are supported. To recognize duplicate objects a distinction must be made between identity and equality. Two objects are identical if their object identifier is the same. Two objects are equal if their values are recursively equal. In a pure object-oriented data model they have to (recursively) refer to the same base objects. Therefore, the whole object structure must be traversed before it can be decided that two objects are equal.

Another argument is that values are more appropriate for representing temporary results. This is true for queries that retrieve a single attribute value. However, query results are generally complex structures, which capture the relationship between objects existing in the database. The query can thus be considered an ad-hoc classification of the objects residing in the database. In this light each solution can be seen as an object itself with a unique identity.

The final argument forms the application-program data-base interface. As most applications are written in a value based language, the data base should offer both objects and complex values for the interface. However, this can be avoided if the data base can not be manipulated directly, but is only accessible through methods. The values of the base types occurring in the methods parameter list are 'mapped' on their counterparts of the base classes. Complex objects are created with the methods defined for the class and they are referred to by properly typed variables.

3.3 Language overview

This section gives a brief overview of the Goblin language and concentrates on a small subset of the language, relevant to the design of the Goblin DBMS. A more complete preliminary description can be found in [KvdBS+93].

We give a top down description of the language starting with typing and class specification. The remaining concepts like functions, methods, statements, expressions are defined on the way.

3.3.1 Types and subtypes

In a programming language the structure of a value is captured by its type definition. Typing is a powerful concept to reduce semantic errors and to produce efficient programs. In a database context, however, data outlives the program and even worse, the real world entities represented, can change their type dynamically. Therefore, types are used in Goblin in a class specification to define the minimally required structure for objects in the class extent.

In the data model the type of an object was defined recursively in terms of the base types and type constructors. In general two kind of object can be distinguished: atomic objects and complex objects.

Atomic objects are defined using the set of atomic types OBJECT (generic object), INT (integer values), STR (character strings), FLOAT (floating-point numbers) and BOOL (boolean values). Furthermore, the collection of base types can be extended using an abstract-data-type facility. For brevity, this facility will not be further discussed. Atomic objects can not change their value or be decomposed into subobjects. A fixed set of functions is defined on them as prescribed by their compile-time ADT. For instance, on integer objects the function to multiply two integer objects defines a mapping from two integer objects to an integer object.

To model complex objects the Goblin data model provides the standard set of constructors TUPLE, UNION, SET, ARRAY, BAG, and LIST. These constructors group multiple objects into a single object. For instance, a tuple type object has its own identity and it associates the tuple object with a collection of subobjects. Contrary to atomic objects, the composite objects can be decomposed into subobjects and the association with these subobjects can be changed.

Example 3.1 In the following example the types `Person` *and* `Address` *are defined. Note that the definition of* `Person` *is recursive. The recursion on the* `spouse` *and* `kids` *attributes imply that the subobjects are restricted to objects of the type* `Person`*. For readability type names are written in capital letters in our examples.*

```
TYPE ADDRESS = TUPLE(
   STR street;
   INT number;
   STR city;)
```

```
TYPE PERSON = TUPLE(
   STR name;
   ADDRESS address;
   INT dob;
   PERSON spouse;
   SET(PERSON) kids)
```

To obtain a terse specification of tuple types, it is possible to define a tuple type by specialization or generalization of a tuple type. These operations form a tuple type by adding or subtracting attributes from a predefined tuple type, respectively.

Example 3.2 The first example illustrates how the `Student` *type is defined as a specialization of the* `Person` *type by extending it with the* `major` *attribute. The second example shows type generalization, considering the definition of the* `Person` *type to be a generalization of a* `Student`.

```
TYPE Student= Person  + TUPLE(STR major);

TYPE Person = Student - TUPLE(STR major);
```

3.3.2 Class and subclass

A class specification serves two purposes. First it is used as a mechanism to group and provide access to similar objects. Using the class name, functions and methods can be applied to all the objects in the class extent. Second, it is a template for creating objects with a minimal set of prerequisite properties. The latter usage will be discussed in Section 3.3.6.

Classification of individuals on the basis of their properties is generally used technique to control the complexity of the real world. As the properties of individuals evolve over time, they will move from one class to another without manual intervention. Therefore, in Goblin class membership is considered to be a dynamic property, which is determined by two factors: structure and state. If the data base is updated, the class extents will be changed accordingly. Thus, in Goblin classes play a role similar to data-base views in relational systems. Furthermore, the class specification determines the set of methods that can be applied to the objects in the class extent.

The minimal class specification consists of a class name and a tuple-type specification. Consequently, the objects in the corresponding class extent can not be manipulated through methods or functions other than those which are predefined for objects of the specified type. Thus, because the type is always a tuple type, the attribute values of the objects in the class extent can be retrieved.

Example 3.3 The following class definition defines the Person *class to consist of objects having the structure specified by* PERSON.

```
CLASS Person TYPE PERSON
```

The behavior is specified by the methods and functions that can be applied to the objects in the class extent. The user can extend the standard set of functions and methods implied by the type by defining its own set of operations. Each method and function definition consists of a specification and implementation part.

Its specification defines the minimally required types for the parameters and the *target type*. The latter type refers to the type of the object to which the operation is applied. For a correct class specification the class type must be a subtype of the target types of its methods and functions, because an operation can not refer to an attribute that is not defined for the objects in the class extent.

Methods or functions can either be implemented in the Goblin language itself or be linked to externally defined functions. The latter possibility is included to provide access to large existing applications or software.

Example 3.4 Continuing our example, we consider extending the Person *class with a function* age *to determine the age of a person and a method* move, *which changes person's address. The implementation of these operations can be found further on.*

```
CLASS Person TYPE PERSON
Person.age();
Person.move(ADDRESS NewAddress);
```

Classification by state is controlled with a *class constraint*. This constraint is a predicate satisfied by all the objects in the class extent. It consists of the keyword WHERE followed by a boolean expression over the attributes defined by the class type. The symbol SELF in the class constraint refers to an individual object from the class extent.

Example 3.5 The following example completes the Person *class definition with a constraint, which specifies that the age of a person should be a positive number and that its address must be defined.*

```
CLASS Person TYPE PERSON
WHERE SELF.age >= 0 AND NOT(SELF.address == NIL)
Person.age();
Person.move(ADDRESS NewAddress);
```

The class inheritance construct ISA allows reuse of class specifications. The methods applicable to objects from the original class are also applicable to the

objects of the new class. Furthermore, the class constraint is the conjunction of the class constraint of the original and the newly defined class. Finally, the type of the new class is simply the aggregation of the type of the original class and the defined class.

Example 3.6 To illustrate the effect of inheritance we give two semantically equivalent specifications of the `Employee` *class. The first specification uses the inheritance relation and adds an attribute, method, and constraint to the* `Person` *class.*

```
TYPE WORK = TUPLE (STR company; INT salary);

CLASS Employee ISA Person TYPE WORK
WHERE SELF.salary > 10,000;
Employee.raise(INT increase);
```

This specification is equivalent to the following explicitly defined class:

```
TYPE EMPLOYEE=PERSON + WORK
CLASS Employee TYPE EMPLOYEE
WHERE SELF.age >= 0 AND NOT(SELF.address == NIL)
                        AND SELF.salary > 10,000;
Employee.age();
Employee.move(ADDRESS NewAddress);
Employee.raise(INT increase);
```

3.3.3 Derived classes

The classes defined until now are implicitly populated by objects that already exist in the data base. However, an important feature of data bases is the possibility to combine the stored information and make the relations between the objects explicit. For this purpose the Goblin provides the *derived class* construct. It is the only construct for querying the data base.

With this construct a new class can be defined in terms of existing classes. The derived class specification enumerates in a *binding list* the class extents from which it is derived. The binding list is a list of class name and attribute pairs. These pairs implicitly define the type of the derived class to be a tuple type of the named attributes and their corresponding type.

Similar to an ordinary class, methods and functions can be defined which operate on the underlying class extents. The derived class extent follows the changes in the underlying class extent.

Without the class constraint the extent of the derived class consists of the Cartesian product of the class extents occurring in the binding list. Similar to the normal class specification, is the class constraint a condition which must be satisfied by all the objects in the class extent.

Example 3.7 The following example illustrates how the class extent for couples is constructed from the `Person` *class. Note that the extent of the Couples class is implicitly populated and reflects changes in the database. The attributes* `p1` *and* `p2` *of the* `Couples` *class extent range over the class extent of* `Person`*. The type* `COUPLE` *defines which attributes in the binding list occur in the result. Furthermore, it defines the properties maintained in the Couples class to be able to define the function* `lat`*, which checks whether a couple lives separately.*

```
TYPE COUPLE= TUPLE( PERSON p1; PERSON p2 );

CLASS Couples
FROM Person p1, Person p2
WHERE p1.spouse == p2 AND p2.spouse == p1
Couples.lat()
```

3.3.4 Functions and methods

The major advantage of OODBMS over relational databases is the possibility to specify the behavior of objects. This feature reduces the impedance mismatch between the procedural application language and the declarative data-base query language.

In Goblin the behavior of an object of a specific type is captured by functions and methods. Functions are used to model derived attributes and do not change the data-base state. Methods are intended to model object updates, and guarantee the common transaction properties.

For the base types the standard set of functions for manipulating boolean, numeric and string objects is supported. These functions define mappings between objects of the base types. There are no methods defined for the base types, because these objects are static entities.

For the complex types, functions are defined to access subobjects. The *dot* operation provides access to subobjects of a tuple object. If a tuple o has an attribute a, the operation $o.a$ returns the object associated with attribute a. The result type is the same as the type of the subobject.

On sets and bags the standard operators union ($+$), difference ($-$), intersect ($*$), membership test (IN) and set comparison operations are provided. The index operations ([]) provides access to the individual elements of array and list objects. List objects can be concatenated with the operator ($+$).

For the aggregate constructors, set, array and list, standard methods are defined to insert ($+$) and delete ($-$) elements. The equivalent operation on tuple objects adds ($: +$) and deletes ($: -$) tuple attributes, respectively. The latter two operations change the type of an object and therefore have an effect on the classification of the object.

Example 3.8 The following example illustrates how a `Person` *object becomes a member of the* `Employee` *class by adding the required attributes* `salary` *and*

`company`. *The modification must be performed in a method.*

```
Person.workfor(STR company; INT salary)
{
   SELF:+ WORK;
   SELF.company= company;
   SELF.salary = salary;
}
```

Naturally the user can extend the set of methods and standard functions. The next section summarizes the basic building blocks. For a complete overview we must refer to [KvdBS+93].

3.3.5 Statements and expressions

Most of the imperative programming constructs are offered to let a user define his own functions and methods. These features include expressions, assignment-, conditional-, repetition-, return- statement and function calls. Within a function or method definition the symbol *SELF* refers to the object to which the function or method is applied. Function and method application is annotated with the *dot* operator. If o is an object and f an operation then $o.f()$ executes the operation code on the object o. If the operation does not have an argument, the brackets can be omitted.

The *dot* operator for function application and attribute selection is left-associative, which allows the programmer to intermix attribute selections and function applications in a single expression. This expression is called a *path-expression*, because it specifies a traversal through several objects.

If a path expression occurs as the left hand of an assignment, the statement associates the object identifier of the object specified by the path expression, to the object occurring in the right hand side. The object type of the right hand side must be a subtype of the object on the left hand side.

Example 3.9 The following example defines the `age` *function and* `move` *method used in the class specification of* `Person`. *Function and method specification use a C-like syntax.*

The variable `now` *is a global integer variable, which maintains the time. The assignment in the definition of method* `move` *changes the association of the* `address` *attribute.*

```
INT Person.age()
{
   RETURN now - SELF.dob;
}
```

```
Person.move(ADDRESS NewPlace)
{
   SELF.address = NewPlace
}
```

3.3.6 Objects and class extents

The class specifications also serve as templates for creating objects. Initially the created object will therefore be added to the class extent, if it satisfies the class constraint.

Previously we have described how the type of tuple objects can evolve using the operations to add or delete attributes. Similar operations are available to modify the behavior of objects by adding or deleting methods and functions to objects.

Example 3.10 To facilitate manipulation of objects, variables can be used to hold an object reference. The following illustrates the creation of a `Person` *object. Uninitialized attributes refer to the* NIL *object.*

```
John=Person(name='John Doe',
            address=Address(street='42st',
                            number=1239,
                            city='New York'),
            dob=1892008)
```

3.3.7 Application interface

The first step in the design of an application on the basis of an existing data base is to identify the information to be extracted. This information may already be available in the form of the previously `Person` class, or may be extracted by querying several classes, like the `Couple` derived class. This steps typically results in the definition of a set of derived classes.

In the next step the operations on the data is defined. These operations can be display operations, linking the Goblin system through functions to a graphical user interface, or update operations, which add new or modify objects through method calls.

The Goblin programming model is based on classification and method application. First classification attaches a name to a collection of objects of interest. Then on these objects operations can be performed by applying the method on the objects in the class extent.

Example 3.11 The following application prints out the names of couples that live separately. First a class is specified to identify these couples, then the display operation is defined and, finally, the display operation is applied to the objects of the class defined.

```
CLASS Lat-couple ISA Couple
WHERE lat(SELF);

Couple.display();
{
   printf("%s, %s\n",p1.name,p2.name)
}

Lat-couple.display();
```

3.4 Conclusion

In this chapter we introduced the data model and a subset of the Goblin DBPL. The Goblin data model is designed to support a dynamic environment, where objects evolve during their lifetime. This is, contrary to other object-oriented data models, not restricted to the objects state, but also involves the structure.

In this light a class is considered to be a mechanism to dynamically group objects which satisfy a minimal set of properties. The defined classes form a taxonomy, which allows exceptions; "Platipus" objects from the UoD, which are neither mammal nor bird can still be modeled and exist in the data base.

The language provides the derived class concept to access the data base. The declarative nature provides ample opportunities for parallelization and set-oriented operations. Methods and functions form the basis for the application interface and support imperative programming construct to achieve the requirement of a fulfledged DBPL.

This thesis is focussed on parallel query processing. This chapter gives only a flavor of the language. Consequently, many issues in the language design and its data model have not been addressed.

Chapter 4

The Goblin storage model

4.1 Introduction

The storage model is an important performance factor for a data-base system. The Goblin data-base system design takes a new approach which justifies considering the possible storage model in depth. The issues of interest are its main-memory design and dynamic query processing.

Goblin is a main-memory object-oriented data base, while most work in this field has concentrated on disk-based systems. The influence of main-memory on the storage model requires attention to find a new balance between storage and processing.

Goblin is a parallel system and uses a dynamic query-processing scheme. Flexible replication and data declustering is essential for efficient parallel query processing. Furthermore, the storage model should be adaptive and support run-time query optimization, such as on-the-fly (partial) indexing.

In this chapter we first give an overview of the approaches to object representation researched in the past. Second, we present the Goblin storage model as a layered architecture, and finally, we describe each of the storage layers in detail.

4.2 Object representation issues

The base line for an OODBMS is to choose a mapping of the objects from the conceptual model to a *physical representation*, such that a good update and retrieval performance is obtained.

From Chapter 3 we known that at the conceptual level a distinction is made between three kind of objects: *atom*, *tuple* and *set*. The atom type objects are used to represent values from the common base types (int, string, float).

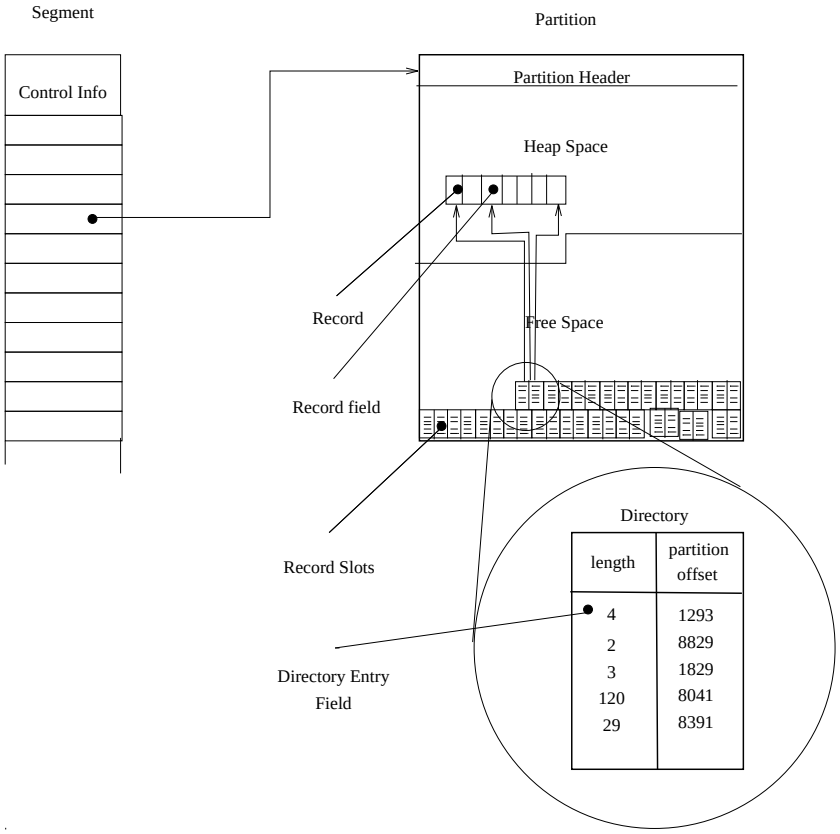


Figure 4.1: Physical level storage concepts

Tuple objects are of the form $(oid, a_1 : v_1, \ldots, a_k : v_k)$. The a_i are the attribute names and v_i are the corresponding attribute values. The attribute values can either be objects identified by their object identifier, or values from one of the basic types (integer, real, string) supported by the system. Set objects are of the form $(oid, \{v_i\})$, where oid is the object identifier of the set and v_i is a collection of values of the same type. List and array objects can be considered to be a refinement of a set or tuple object using an implicit naming scheme of its elements.

At the physical level the basic concepts are *segments*, *partitions*, *directory*, *records*, and *surrogates*. These concepts are introduced in the following paragraphs. The relationship between these concepts is depicted in Figure 4.1.

Each relation or class extent is stored in a variable sized *segment*. A segment further consists of a variable number of fixed size partitions, which form the basic unit for allocation, locking and recovery. For main memory data bases the partition sizes are generally larger than disk pages and range from 64 kbyte - 256 kbyte [LSL92].

The partitions store the actual records. For this, each partition consists of a header containing control information, a list of record slots, and a heap space. The record slots contain the fixed size directory structure of the records. The actual data is allocated in the heap space of the partition.

Surrogates are system generated unique identifiers, which are independent of any physical address. A record is a contiguous amount of memory of a (possibly variable) number of fields. Each field is used to store a data item, which can either be a surrogate referring to another record, an atomic value, or a set.

The fields correspond to the attributes at the conceptual level. The mapping from attribute to field is determined by the storage model. It is either fixed at record creation time or it is dynamic to accommodate variable length attribute values. In the latter case the record contains a directory to associate each attribute with a reference to the stored data.

In the following paragraphs the design issues involved in choosing an effective and flexible storage representation are discussed.

4.2.1 Clustering and declustering

Traditionally the two main techniques used for physical data-base design are *clustering* and *declustering*. Originally they were developed to improve the performance of disk resident relational bases. In fact, most research on object representation remains focussed on disk resident data [VKC86, CDRS86, VBD89, HO88].

In designing a storage model for disk resident database systems the *I/O bottleneck* forms the main impediment for a better performance. The bottleneck is caused by the low disk bandwidth and slow disk access time, due to rotational delays and seek time. The performance can be improved by reducing the number of I/O requests and by increasing the I/O bandwidth through parallelism.

The number of I/O requests is reduced by storing attributes frequently referenced together in the same physical record. This physical adjacency of data improves access and update performance, because the disk access time depends on the location of the previously accessed record and the rotational delay of the disks. Grouping object attributes into a single record is also known as *clustering*.

The I/O bandwidth is increased using parallel disks, such as in RAID (Redundant Array of Inexpensive Disks) technology [Gib91], which also improves the reliability of the I/O subsystem. The parallel disks are not visible through the RAID interface. From the data-base system the RAID disk simply appears to be a fast and reliable disk.

Another approach, where the parallelism is visible to the DBMS is *declustering*, which distributes the records over partitions and stores them on different disks. The main objective of declustering is performance improvement through exploitation of parallel disk I/O. In relational data-base terminology declustering is also known as fragmentation. Declustering is an issue orthogonal to the storage model. Once the record structures have been determined for a database scheme, the instances can be declustered over the available disks.

For memory resident databases physical adjacency is less of an issue, because

the memory access time is largely independent from the physical address [1]. In this case the CPU cost for data access and update data is more important. Therefore, evaluation of a storage model for main-memory must take the CPU cost into account.

In loosely and tightly coupled parallel memory resident DBMS the data is distributed over the memories of the available processors. The result of this scheme is that queries can be executed in parallel on the composing data fragments. Furthermore, to achieve effective parallel execution, the distribution of the data fragments must be taken into account for query scheduling.

4.2.2 Object sharing

The ability to handle shared objects is an issue rarely addressed explicitly in papers on storage models for object oriented data bases. In fact many papers use the term *complex object* to denote *nested relations* or *non-first-normal-form* relations, which do not support directly the concept of sharing.

An object is *shared* if it is referenced from multiple other objects, called its parent objects. The clustering techniques aggregate object attributes into a single physical record. If one of the attributes is a shared object, the storage model must define how this is represented. There are basically two approaches to reconcile clustering and object sharing: *replication* [HZ87] and *election* .

The *replication* strategy effectively stores a copy of the shared object with each of its parent objects, achieving a perfect data locality. However, this approach suffers from the overhead required to keep the replicas coherent. An update of a shared object must be effectuated on all its replicas to guarantee data-base consistency. Furthermore, if the data is declustered, the parent objects could be allocated on different sites, requiring a replica control mechanism to keep track of shared objects to ensure data-base consistency. A possible implementation would be to store a reference with a shared object to each of its replicas. Updates on a shared object can then be effectuated by following the chain of replicas and update each one in turn.

The *election* strategy stores the shared object with one of its parents and merely stores a reference to the shared object with its foster parents. The choice of candidate parent is either done automatically, based on statistics about the frequency of reference [TN91], or is under user control.

Object clustering in O_2 [AK92] and ORION [KBG89] are examples of the user-driven approach. The user can specify with IS_PART_OF relationships (called placement trees in O_2) the object location. However, the clustering of shared objects is still determined non-deterministically by the system.

In a parallel main-memory system object sharing becomes an issue when the objects are allocated on separate memory segments.

4.2.3 Object dynamicity

To support the Goblin data model there is yet another issue that plays an important role: the object properties can change during its life time. Thus an object

[1] If you do not consider the very fast but small cache memory.

can acquire or drop attributes. A common example of this object dynamicity is an evolving set-valued attribute.

In both cases the storage model should be able to handle these changes efficiently. Both updates may involve a reorganization of the physical record. For instance, extending a set-valued attribute may result in a reorganization of the record if the memory set aside is insufficient. In the main-memory implementation of Starburst [LSL92], this problem is solved by introducing *tombstones*, which are references left behind at the old location to denote the new physical location. A side effect is that in each access of an object attribute, it must be tested whether the data or a tombstone can be found at a certain record offset and an additional dereference operation may be introduced.

4.2.4 Data-base workload

The performance of a clustering technique strongly depends on the predominant query access patterns. In analogy of the definitions given in [HO88], we discriminate query classes on the basis of their access patterns.

class A queries, which manipulate a large number of attributes of a few objects.

class B queries, which manipulate a few attributes of a large number of objects.

class C queries, which represent the average case, where the ratio accessed attributes per object is more balanced.

In the following examples we show representatives for each workload class using the `Person` class introduced in Example 3.1.

Example 4.1 A representative of a class A query is the selection of a single object. Class B queries extract only a few attributes from all objects of a certain class. Class C queries represents the mixed case.

```
CLASS A ISA Person
WHERE SELF.name==``John Doe'' ;
```

Class B queries have in common that they select only a few attributes of all objects. In Goblin this is expressed through the type specification.

```
TYPE NAME=TUPLE(STR name)

CLASS B TYPE NAME
FROM Person p, p.name name;
```

Class C queries represent the mixed case, where only a part of the object is retrieved from a selection of objects.

```
TYPE CTYPE= NAME + TUPLE(ADDRESS address)
```

```
CLASS C TYPE CTYPE
FROM Person p, p.name name, p.address address
WHERE SELF.address.city =='Paris'
```

One of the design guidelines mentioned in Chapter 2 is our focus on applications with large collections of similar data, rather than navigational queries. In other words, the B and C workload are the focal point for the Goblin architecture.

4.3 Object storage models

Having shortly introduced the main design issues of an OODBMS storage model, we will now explore some in more detail before we present the choice made for the Goblin storage model.

Example 4.2 As a running example of the different storage schemes we consider the representation of the following Person object (See example 3.1). It is used to calculate the minimal storage cost excluding search access paths.

```
a1 = Address(street='42 st.',number=1239,
             city='New York')
tom = Person(name='Tom Doe',)
alice= Person(name='Alice Doe',)
jane = Person(name='Jane Doe',address=a1,
              dob=829128,address=a1,
              spouse=john,kids={tom,alice})
john = Person(name='John Doe',address=a1,
              dob=829109,address=a1,
              spouse=jane,kids={tom,alice})
```

We use a graphical presentation to show the different storage schemes. In Figure 4.2 a legend can be found.

4.3.1 Flattened Storage Model

In the flattened storage model (FSM), also known as the direct storage model, each object with all its attributes is stored into a consecutive byte sequence. Tuple- and set valued attributes are contained within a physical record. For instance, the ADABAS hierarchical database system stores the segments of a hierarchical record in a single file [Oll71]. In OASIS [Wie83] a tuple with all its descendants are stored in a single variable length record.

To accommodate variable sized fields, the record contains a directory for each object, subobject, or set-valued attribute, which encodes for each attribute value its offset within the physical record. Access to a particular attribute requires at least one additional dereference operation to retrieve the field offset from the

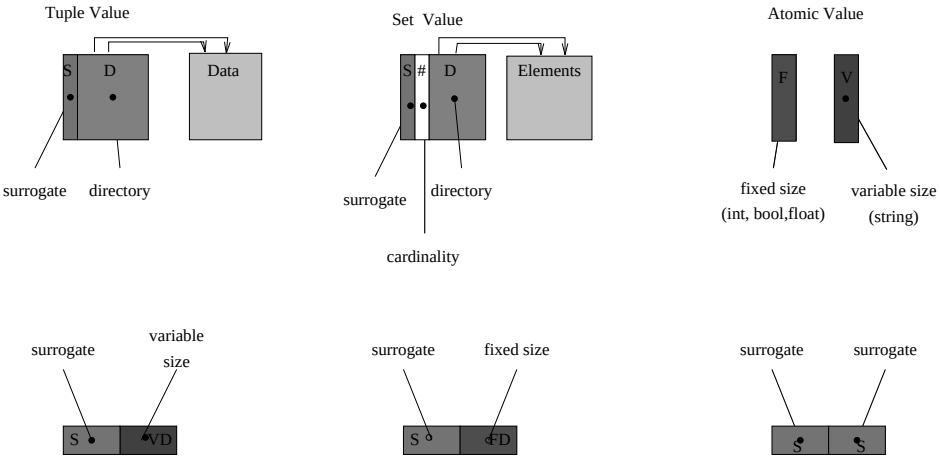


Figure 4.2: The symbols used for representing tuple, set and atomic values

record directory for single valued attributes and two dereferences for set-valued attributes. One to retrieve the offset for the directory containing the offsets of its elements.

Example 4.3 In the FSM the object "John Doe" is represented by a single record, containing its address, and the complete records of its children (See Figure 4.3).

Let s, i and p denote respectively the size of a string, integer and pointer respectively. Then we count the total number of bytes used this data base as follows. Using a replication strategy the kids are stored with both their parent objects `john` *and* `jane`.

The `address` *has a fixed size of $2s+i$ Assuming fixed length strings, the field offset of the attributes is determined at compile time, so that this object does not require a directory.*

A `person` *object references the subobjects* `address`, `spouse` *and* `kids`. *The record directory contains therefore 3 pointers. If the set* `kids` *contains n elements, an additional $i+n.p$ bytes must be reserved to store the references to the subobjects.*

Both children require $3p$ bytes for the directory, $2s+i$ to represent the address and $s+i$ to represent their date of birth and name; in total: $3p+3s+2i$.

The parent objects require an additional $i+2p$ bytes to store the directory for the set attribute and $3(3p+3s+i)$ bytes storage for the `spouse` *and both children subobjects.*

Thus storing the whole data base consisting of the four person objects requires a total of $4(3p+3s+i)+2(2p+i+3(3p+3s+i))) = 34p+30s+12i$ bytes.

For disk resident data bases FSM is efficient if complete objects are the unit of manipulation. For memory resident data bases the main advantage stems from the fact that no joins are required to reconstruct an object, because all object

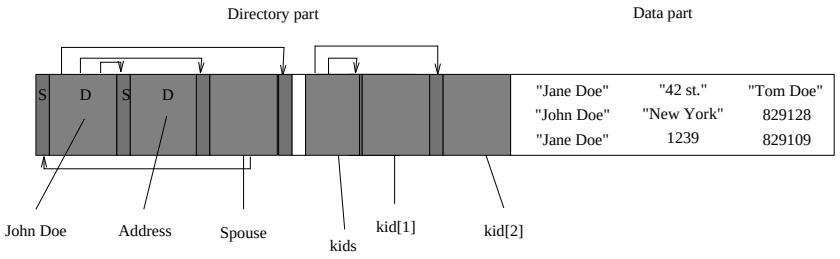


Figure 4.3: The FSM storage model

attributes are stored within a single physical record. The join is essentially precomputed.

Furthermore, the 1-1 correspondence between the conceptual and physical record, facilitates the compilation of queries using conventional compilation techniques.

The drawbacks of this storage model become apparent when object sharing and object structure updates are considered. Subobjects, like the children `tom` and `alice` from the example, are contained in the physical record of their parent objects. If the subobjects are shared, they have more than one parent and consequently it is not obvious where the subobjects must be stored (See Section 4.2.2).

Handling object sharing by a replication technique introduces storage and update overhead to keep the replicas consistent. If, on the other hand, the election technique is used, additional dereference operations are introduced, thereby eliminating the initial advantage of FSM for efficient manipulation of complete objects.

The properties of an object can change. For instance new attributes can be defined or a set valued attribute can be extended beyond its initial size. Because each physical record contains all the subobjects reachable from the stored object root, modification of the object properties often requires a reorganization of the physical record layout. Furthermore, programs which are based on the old record layout must be recompiled for the new record structure.

4.3.2 Normalized Storage Model

In the normalized storage model (NSM) a complex object is decomposed into a set of records containing only atomic values or surrogates. In particular each set of objects corresponds to a normalized relation. Similar to FSM, each record contains a directory to store the field offset for each attribute.

Under this approach, updates on shared objects or modification of set-valued attributes can be performed without the shortcomings of FSM. However, access to subobjects or retrieval of the complete object requires join processing. It should be noted that this access cost can be reduced by maintaining join indices [Val87].

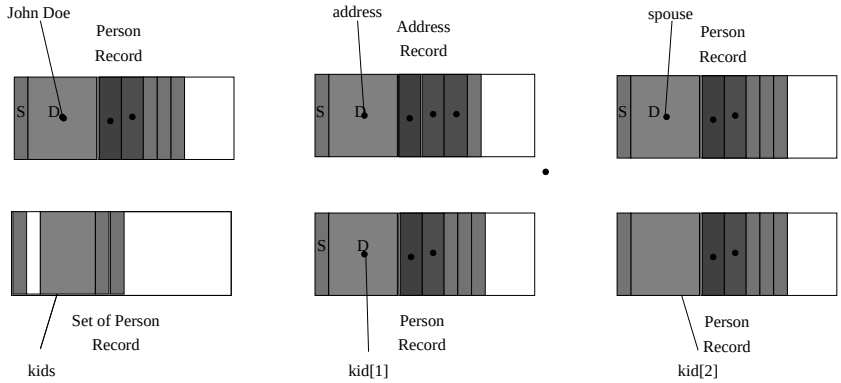


Figure 4.4: The NSM storage model

A prime disadvantage is again that modification of the object properties requires reconstruction of the record, but, contrary to FSM part of the object needs reconstruction. As subobjects can be shared, the old record can not be simply deallocated. To avoid dangling object references from the parent objects, a tombstone should be left at the memory location, which refers to the new record [LSL92].

Example 4.4 Continuing our example, under NSM the person object is represented by the records shown in Figure 4.4. Basically all the subobjects are represented by separate records. The person objects `tom`, `alice`, `jane` *and* `john` *are represented by separate person record structures. Furthermore, they reference each other either directly, or through a record structure presenting the* `set-of-person` *object* `kids`.

The NSM data model uses the same directory structure as FSM. However, as subobjects are stored separately, the total storage cost is much less than in FSM. A person object now only contains references to its subobjects `spouse`, `address` *and* `kids`. *The total storage requirement for the example data base is therefore: one* `address` *record of* $(p+2s+i)$ *bytes, four* `person` *records of* $(5p+s+i)$ *bytes, and one* `set-of-persons` *record of* $(3p+i)$ *bytes, giving a total of:* $(24p+6s+6i)$ *bytes.*

4.3.3 Decomposed Storage Model

In the decomposed storage model (DSM) each attribute is mapped onto a binary relation and each value is associated with the surrogate of its conceptual tuple or set. Similar to NSM, DSM has the advantage that shared objects are stored once only. Furthermore, the storage requirements for DSM are not necessarily larger than for NSM. In DSM each attribute value requires the storage of a surrogate, while in NSM storage is required in the directory to record the field offset of the attribute value or to represent a NULL value.

Furthermore, whereas in NSM it is possible to access an atomic attribute

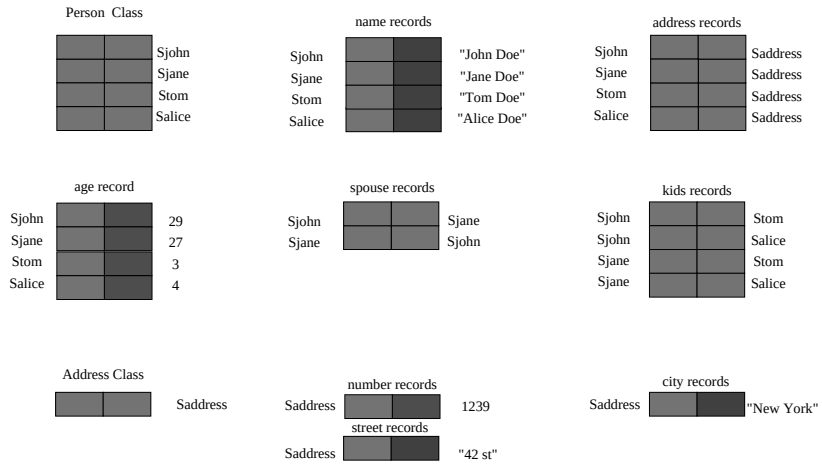


Figure 4.5: The DSM storage model

value simply by using its offset in a record, in DSM a lookup operation must be performed to retrieve the attribute requested. Hash-based join indices can be maintained to speed up this processing.

Example 4.5 Using DSM, the Person and Address objects will be completely decomposed in a set of binary relations. The surrogates are used to maintain the structural relationships between the object attributes. Each attribute is represented by a relation. The special surrogate S_{anchor} *identifies the anchor point of the data and is merely used to encode that* S_{john} *is a valid object stored in the data base. Null values do not have to be recorded because class membership implicitly determines the type structure. The DSM storage model is illustrated in Figure 4.5.*

In the DSM model the records store simply `( oid, value)` *associations. The* `address` *record is thus stored in three binary associations (*`oid, city`*), (*`oid, street`*), and (*`oid, number`*), requiring a total of* $3p + 2s + i$ *bytes. Similarly, the person object is represented by 5 associations. The* `kids` *set-valued attribute is also represented by a binary relation associating parents with their children. Note that contrary to DSM and FSM non-existing associations do not require storage. Thus the non-existing* `spouse` *and* `kids` *associations of the children do not contribute to the total storage requirement.*

The storage requirement for a parent and a child is $8p + s + i$ *and* $4p + s + i$*, respectively. The kids association requires an additional* $4p$*. Thus the total storage requirement is* $27p + 6s + 5i$*.*

Notice that the storage overhead is limited to $3p$ *compared to NSM and that the number of kids is implied by the representation. If the number of optional attributes increases, the storage requirement for the DSM scheme is even less than for NSM.*

4.3.4 Other storage models

For completeness we will mention two hybrid models. On the basis of FSM, DSM and NSM two hybrid models have been defined: Partial Decomposed Storage Model (P-DSM) [VKC86] and Partial Normalized Storage Model (P-NSM) [HO88].

P-DSM is a combination of NSM and DSM. It vertically partitions an object such that attributes used together frequently are stored in the same file. Often the attributes are associated with the surrogate of the tuple or set of which it is a part. P-NSM can be seen as a combination of FSM and NSM. In this approach an object is vertically partitioned, such that complete subobjects are stored in the same file.

These storage models offer the possibility to combine the advantages of both schemes and to tune the database partitioning to a certain query mix (See for instance [TF82][pages 201–224]). Obviously, for highly dynamic workloads this approach is not suited. As these models are derived from the basic storage models we will not discuss them in further detail.

4.3.5 Storage model comparison

Both the NSM and DSM approach can express object sharing in their storage model. In NSM the unit for clustering is the collection of atomic attributes with their surrogate, which is simply its object identity. In FSM object sharing is cumbersome and it can only be implemented by data replication or using an election strategy.

DSM offers the best support for dynamically changing objects. In the NSM and FSM approach adding new attributes to an object requires a complete storage reorganization, because the mapping of attributes to record fields is fixed at object creation time. Consequently, when an attribute is added to the object a new record layout must be determined for both approaches. Adding elements to a set valued attribute is cheap in both NSM and DSM, but possibly requires a record reorganization in FSM.

Furthermore, DSM has the advantage that selection of storage techniques can be done independently for each attribute, eg. fixed or variable sized record field. In NSM and FSM, where many attributes are stored together, these decisions strongly interact. Similar arguments hold for indexing and data compression.

Judging from the storage requirements for the example data base we conclude that DSM and NSM require far less storage than FSM. DSM has the added advantage over NSM that undefined associations (eg. an undefined `kids` or `spouse` attribute) do not require storage at all. Thus even though intuitively the DSM approach introduces more storage overhead than NSM by duplicating the `oid` of an object for each of its attributes, we learn from this example that this is not a rule and basically depends on the number of optional attribute values.

For class B and C queries DSM shows a better performance in disk-resident data bases [VKC86]. As I/O cost is the main cost factor in these systems it is not possible to carry the same conclusions over to memory-resident systems.

The high maintenance and reconstruction costs for DSM do not apply to a

	FSM	NSM	DSM	OOPL
shared objects	– –	+	+	++
object evolution	– –	–	++	–
declustering	–	+	++	– –
storage overhead	– –	+	+	+
class A workload	++	+	–	++
class B workload	–	+	++	±
class C workload	+	+	+	+

Table 4.1: Summary of the qualitative storage model comparison

main-memory system, because these extra costs are far less prohibitive than for disk based systems. We feel that a well-designed main-memory based DSM system should at least provide the performance of an NSM-based system. The reason is that at the lowest implementation level in NSM each attribute of a flat object can be represented by a single pointer and a (mostly) fixed offset. DSM merely requires a register file to denote the object components. However, it is clear that DSM favors class B queries and NSM and FSM favor class A queries.

DSM provides better opportunities for load balancing than using NSM or FSM. If the attributes of a single object are accessed frequently, DSM can spread the attributes of the object over several processors. This is not possible with NSM, where the unit of allocation is a single (sub)object.

Furthermore, in a distributed setting where semi-join operations are extensively used to reduce data transport, DSM does not require an expensive projection operation as with NSM.

The qualitative comparison of the three storage models introduced so far: FSM, NSM and DSM, is summarized in Table 4.1. We have also included an object oriented programming language (OOPL) in this comparison.

4.4 Goblin storage model

The approach taken in Goblin is to pursue the DSM track further with a focus towards a main-memory implementation and a loosely coupled multiprocessor. The rationale behind this choice stems from the disadvantage of fixing a single storage structure, —to deal with aggregate types only—, from the outset of a DBMS implementation. It means that extensibility at the lower system level is sacrificed a priori for initially good performance.

Rejection of the NSM approach is a direct consequence of its disadvantages (=advantages of DSM) and the feature of Goblin to permit users to add attributes to objects on an individual basis. Using the NSM approach would require continual data-base re-organization or an implementation that claims storage space for all possible attributes of an object from the outset.

Furthermore, a DSM approach is more efficient in a distributed environment, where semi-join operations do not require expensive unpacking of an NSM storage structure.

Therefore, the Goblin storage management scheme is a DSM-like implementation which allows extending the set of basic types by defining a minimal set of storage management functions. The details of our storage management scheme are elaborated upon in the subsequent sections.

4.4.1 Storage model overview

The Goblin storage model is divided into four layers: the *schema layer*, the *summary layer*, the *data layer* and the *storage layer*.

The *schema* layer manages the intentional data, which describes for each class its type structure, the applicable methods, and the class constraints.

The *summary* layer administrates the data fragmentation and distribution. The binary relations are declustered into fragments and allocated on the processors available. This information is maintained for each binary relation in a *Redistribution Association Table* or RAT. The information is organized as an ordinary relation[2]. This approach has the advantage that the relational operations can be used to determine the fragments that participate in a query. Actually, as will be shown in Chapter 7, it is possible to *simulate* the query first against the summary data stored in the RATs, before distributing the query subtasks.

The *data* layer manages the binary relation fragments. The fragments of the binary relations are stored in *binary associations tables* or BATs. They form the unit of allocation and processing. A set of relational operations is defined for BATs, which is powerful enough to support the Goblin query language.

The *storage* layer is formed by the *Global Persistent Object (store)* (GPO). This layer provides buffer management and persistent storage in a distributed environment. Furthermore, it offers primitives for transaction management and concurrency control.

4.4.2 The schema layer

The data-base schema describes the type, methods and constraints for the classes maintained by the DBMS. This information is primarily used for type checking. Either a *static* of *dynamic* approach can be followed to maintain the schema.

In the static approach the data-base definition is compiled into the programs, i.e., the knowledge of the structure, methods and constraints is embedded in the program code. This results in fast programs at the cost of redundant administration storage. It is also highly inflexible, because the description of the data is kept separate from the data itself (e.g. in program header files). This means that schema updates requires program location and recompilation and it becomes difficult to write a generic program to process an arbitrary object.

Alternatively the schema information is kept with the objects (e.g. Smalltalk). This allows for modeling flexibility, because each object can have a different type. The prime disadvantage is the storage overhead of type information with each data element. Moreover, it incurs processing overhead to repeatedly type check operations against individual objects.

In a data-base environment a more reasonable approach is to factor out the type information for bulk data. This corresponds to explicitly storing both ob-

[2]represented by BATs

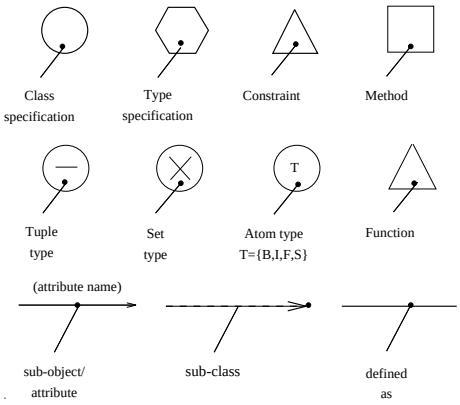


Figure 4.6: Legend for the schema graph components

ject relationships and access information. The former is used to type check operations, while the latter is used for access to bulk data. This means losing some of the flexibility of dynamic typing, but it also greatly reduces the storage and interpretation overhead. By "bulk" processing a high performance is attainable.

In Goblin we focus on the third approach, i.e. factoring out type information for bulk data only, because operations on bulk data provides a good handle on parallel query processing.

The schema layer is administered by the Class Administration Tables (CAT). The CAT is actually a directed graph, where the nodes correspond to the type constructors, atomic types, methods, functions and constraints, described by the data model. The edges define the relations between the nodes. This can be an inheritance or subclass relation, or an IS-PART-OF relation. The inheritance edges describe the relations between class specifications and the subobject edges are used to define the type structure. For instance, a tuple type node is connected through labeled edges with its subobjects. Figure 4.6 gives a graphical presentation of the graph components.

Example 4.6 To illustrate the schema layer we give a graphical presentation of the `Person` *and* `Employee` *class specification as described in the previous chapter (See Figure 4.7).*

Note that the implicit inheritance relation between the most general class `Object` *and* `Employee` *and* `Person` *is included in the schema.*

Associated with each edge is a reference to its extent. In Goblin, it is a reference to summary information about the extent to be described next.

4.4.3 The summary layer

The Goblin summary layer serves as an access path to the partitioned data-base extent. To support distributed query processing each binary relation is a priori

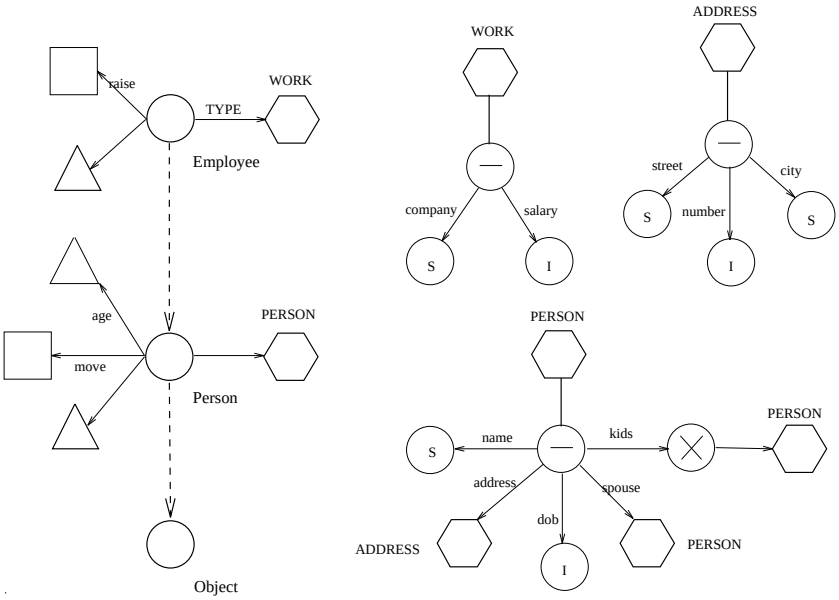


Figure 4.7: The Employee and Person schema definition

declustered into fragments and allocated on distinct processors. Each fragment is mapped on a physical partition. In the query processing phase the fragmentation information is used to direct query requests on the conceptual binary relations to the actual stored partitions. Basically there are three approaches to store the fragmentation information: *reconstruction rules*, *fragmentation rules*, or *fragment data*.

A *reconstruction rule* expresses how the relation can be reconstructed from its fragments. For horizontal fragmentation this rule is simply the union of all the fragments. During the query translation process, view substitution can be used to transform the query on the relation into a query on its fragments. The main disadvantage of this approach is that not enough fragmentation information is available to use for query optimization. For each query all the partitions of a relation must be accessed.

Fragmentation rules represent an opposite approach. These rules store the fragmentation information. For hash-based fragmentation it suffices to store the fragmentation attribute and the number of fragments. Using a default naming scheme for the fragment relations, the query optimizer can translate the query into a query over the fragments and exploit the fragmentation information. For range-based fragmentation the fragment constraint must be maintained for each fragment. To exploit this information, however, the query optimizer must have a semantic query-optimization capability [vK93].

The final approach records *fragment data* in a summary data base, which contains for each fragment attribute its value range. This approach enables

to perform semantic query optimization by simply executing the query first on the summary data base, thereby eliminating those fragments, or fragment combinations, that do not contribute to the query result.

This method is not limited to a particular fragmentation scheme. For range- and count-based fragmentation, the attribute ranges are stored in the summary data base. For hash-partitioned data the hash value for each fragment is maintained.

The summary data base can be considered as a general and flexible indexing mechanism. Running the query first against the summary data base reduces the search space considerably, especially if the binary relations are partitioned on both attributes.

In Goblin the partition information is maintained in a summary data base. This decision is based on the following observations:

- A common technique in distributed query processing is to perform selections first. Performing the selection operation first on the summary data base, reduces the amount of processing required in the following stages.
- Join processing on fragments requires communication. Performing a join operation first on the summary data eliminates fragment combinations that do not contribute to the query result, and, thereby, reduces the amount of communication.
- Efficient equi-join processing is achieved if the join attribute is range partitioned. However, efficient theta-join processing is not possible for hash-partitioned relations.
- The overhead for running a summary query depends on the ratio summary data and data-base size. This ratio and therefore the overhead, can be controlled by adjusting the fragmentation degree.

The summary data is maintained in Redistribution Administration Tables (RAT). Each binary relation has a RAT to store the fragment identities (*hbid* and *tbid*), and the minimum and maximum values for its first and second attribute *hmin*,*hmax* and *tmin* and *tmax*, respectively. The fragment identity is included twice in the summary relation so that after joining two summary relations, the fragment identity of both source relations is kept in the resulting relation (See further on). The fragment allocation information is stored separately as they can be replicated and stored on multiple sites.

Example 4.7 The `name` *attribute of the* `Person` *class is maintained in a binary relation. It records the association between tuple objects and strings. Table 4.2 shows the summary relation* `Name[hbid,hmin, hmax, tbid, tmin, tmax]` *if this relation is partitioned into 5 fragments; each one stored in a BAT partition.*

Summary query processing resembles traditional query processing, but requires a redefinition of the relational operations to use the fragmentation information. To support the summary query process the find, select, equi-join and theta-join operation are modified to use the fragmentation information. It depends on the fragmentation method, whether the summary information can

hbid	hmin	hmax	tbid	tmin	tmax
1	00000	12763	1	"Abiteboul"	"Bergsten"
2	20000	32515	2	"Bodorik"	"DeWitt"
3	40000	52109	3	"Eich"	"Hafez"
4	60000	78198	4	"Hornick"	"Khoshafian"
5	80000	99999	5	"Kim"	"Yu"

Table 4.2: The RAT for the `name` relation

Relational operator	Range partitioning	Hash partitioning	Description
$\sigma_v R$	$\tilde{\sigma}_v \tilde{R}$ (4.1)	$\overline{\sigma}_v \bar{R}$ (4.6)	find tuples matching value
$\sigma_{[l,h]} R$	$\tilde{\sigma}_{[l,h]} \tilde{R}$ (4.2)	n.a.	range selection
$R \bowtie S$	$\tilde{R} \tilde{\bowtie} \tilde{S}$ (4.7)	$\bar{R} \overline{\bowtie} \bar{S}$ (4.3)	equi-join
$R \bowtie_\theta S$	$\tilde{R} \tilde{\bowtie}_\theta \tilde{S}$ (4.4)	n.a.	theta-join, $\theta \in \{<, \leq, \geq, >\}$

Table 4.3: The main RAT operations (n.a. = not available)

be used. To distinguish these operations from the common set of operations we use the symbols $\tilde{\sigma}_v$, $\tilde{\sigma}_{[l,h]}$, $\tilde{\bowtie}$, $\tilde{\bowtie}_\theta$ and $\overline{\sigma}$, $\overline{\bowtie}$ for range partitioned data and hash partitioned data, respectively. This is summarized in Table 4.3.

4.4.4 Range partitioning

For range partitioned relations these operations can be expressed in relational expressions on the summary relations. Essential in this translation is the definition of equality. For range-based partitioning two ranges are "equal" if they overlap. Given two summary relations for R and S, $\tilde{R}[hbid, hmin, hmax, tbid, tmin, tmax]$ and $\tilde{S}[hbid, hmin, hmax, tbid, tmin, tmax]$, respectively, and equality as defined above, we arrive at the following definitions for the relational operations on the summary relations.

$$\tilde{\sigma}_v \tilde{S} = \{t \in \tilde{S} | \tilde{S}.tmin \leq v \leq \tilde{S}.tmax)\} \quad (4.1)$$

$$\tilde{\sigma}_{[l,h]} \tilde{S} = \{t \in \tilde{S} | \tilde{S}.tmin \leq h \wedge l \leq \tilde{S}.tmax)\} \quad (4.2)$$

$$\tilde{R} \tilde{\bowtie} \tilde{S} = \pi_A(\tilde{R} \bowtie_C \tilde{S}) \quad (4.3)$$

where:

$$A = [\tilde{R}.bid, \tilde{R}.hmin, \tilde{R}.hmax, \tilde{S}.bid, \tilde{S}.tmin, \tilde{S}.tmax]$$
$$C = (\tilde{R}.tmin < \tilde{S}.hmax \wedge \tilde{R}.tmax > \tilde{S}.hmin)$$

$$\tilde{R} \tilde{\bowtie}_\theta \tilde{S} \equiv \pi_A(\tilde{R} \bowtie_C \tilde{S}) \quad (4.4)$$

where:

$$A = [\tilde{R}.bid, \tilde{R}.hmin, \tilde{R}.hmax, \tilde{S}.bid, \tilde{S}.tmin, \tilde{S}.tmax]$$
$$C = \begin{cases} (\tilde{R}.tmax < \tilde{S}.hmin) & \text{if } \theta = " < " \\ (\tilde{R}.tmax \leq \tilde{S}.hmin) & \text{if } \theta = " \leq " \\ (\tilde{R}.tmin \geq \tilde{S}.hmax) & \text{if } \theta = " \geq " \\ (\tilde{R}.tmin > \tilde{S}.hmax) & \text{if } \theta = " > " \end{cases} \tag{4.5}$$

4.4.5 Hash partitioning

For hash partitioned data equality is defined on the hash value. Consequently, no equivalent expressions exist for the range selection and theta join operation other than the identity operation and the Cartesian product, respectively. Given two summary relations $\bar{R}[hbid, hhash, tbid, thash]$, $\bar{S}[hbid, hhash, tbid, thash]$ and the hash function $hash$, we arrive at the following operations on the summary data for the equi-join and find operation.

$$\begin{aligned} \overline{\sigma}_v \bar{S} &= \{t \in \bar{S} | \bar{S}.thash = hash(v)\} \\ \bar{R} \overline{\bowtie} \bar{S} &= \pi_A(\bar{R} \bowtie_C \bar{S}) \\ where & \begin{cases} A = [\bar{R}.bid, \bar{R}.hhash, \bar{S}.bid, \bar{S}.thash] \\ C = (\bar{R}.thash = \bar{S}.hhash) \end{cases} \end{aligned} \tag{4.6}$$

The detailed discussion on construction and processing of summary queries can be found in Chapter 6. In the current implementation the RAT relations are mapped on binary relations, enabling an implementation using BATs. The following section describes the issues involved in the design and implementation of the data layer.

4.4.6 The data layer

The main task of the data layer is to manipulate and store the fragments of binary relations, which are stored in Binary Association Tables or BATs. The BAT corresponds to the notion of partition as presented in Section 4.2. The two attributes stored in a BAT are referred to as *head* and *tail*.

The head and tail attribute type can be any of the base types ({BOOL,INT, STR,OID,FLOAT}), but is fixed for the BAT at creation time. This has the advantage that type checking, offset calculation, and selection of the routines to compare, access and store the attributes needs to be done only once for bulk operations, thereby avoiding run-time overhead. The set of base types can be extended by defining six routines to manipulate elements of that type. This interface forms part of the ADT facility and is not further discussed.

The BAT interface is divided into five groups of operations: BAT creation, BUN manipulation, iterators, relational operations and transaction management.

4.4.6.1 Data definition

At creation time the programmer specifies the basic BAT properties. This consists minimally of the attribute types. Initially, the BAT is given a system

generated unique name, which is used by the summary layer to identify the BAT. Furthermore, the behavior of the operations can be influenced by specifying that an attribute has a key property, so that each value must be (locally) unique. Finally, the user can specify for individual attributes that a hash- or comparison based index must be maintained. In most cases, however, the BAT will construct indices dynamically before the execution of a relational- or sort-operation that will benefit from index support.

4.4.6.2 Data access

The BAT contains a number of fixed-size slots to store the binary associations. These slots are called BUNs. A BUN variable is a pointer to the storage area of the BAT. The BAT implementation allocates the BUN slots contiguously, so that iteration over the available BUNs can be performed cheaply. Consequently, after a BAT update, previously retrieved BUNs may not longer refer to the same record slot.

After creation, the BAT can be filled either by loading the BAT from disk or through insertion of individual head-tail attribute pairs. The BAT interface provides operations to manipulate these BUNs. Next to BUN update operations, a search operation and operations to access the head and tail attributes of the BUN are available.

In some cases, for instance after sorting the BAT contents, all BUNs or a subrange of them need to be accessed sequentially. For this purpose, the BAT interface provides an iterator mechanism, which accesses the stored BUNs in succession. The iterator construct also serves as the building block for the relational operations, which often require iteration over an attribute range.

4.4.6.3 Relational operations

The BAT interface offers the ordinary set of relational operations, like the set operations, the select operation, equi-join and theta-join operation. Furthermore, two special operations are provided to support the query processing scheme: *semi-join*, *mark* and *remark*. All operations produce a binary relation. For the join operations the join attribute is omitted from the result. This approach has the advantage that the relational operations have to consider binary relations only and can be implemented efficiently. An obvious disadvantage is that the join attribute is lost in the process and potentially it has to be recovered through a semi-join operation later on in the QEP.

The DSM model also leads to frequent *semi-join* operations. Namely, each tuple attribute is represented by a binary relation, where the head attribute corresponds to the tuple identity (oid) and the second attribute represents the attribute value. After a selection on a tuple attribute, the semi-join operation on the OID of the tuple is used to reduce the binary relations for the remaining attributes.

The *mark* operation is used to invent unique object identifiers to represent query results. Similar to the permanent objects, query results are also represented using DSM. Each object represents a combination of objects from the

data base that satisfies the structure, behavior and state constraint specified by the query. The mark operation gives a name to the individual combinations.

The *remark* operation is a variant of the *mark* operation. It assigns unique object identifiers to the tuples of its operand, but contrary to the mark operation it returns two binary relations; one for each attribute.

4.4.6.4 Transaction management

The BATs form the unit for allocation, locking and recovery. Operations are provided for transaction management and concurrency control. For transaction management, the BAT interface implements the local part of the two- phase commit protocol, and consists of operations to begin, precommit, abort or commit a transaction. For concurrency control the primitive operations consists of requests for read-only and exclusive locks. Although these operations are provided by the BAT interface, they are implemented by the storage layer, which provides the data layer an interface for creating and manipulating global persistent objects.

4.4.6.5 Implementation considerations

Since BATs model binary associations, there are only a limited number of implementation strategies. Namely, a component of the association can be stored explicitly or implicitly. This leads to the following implementation schemes:

- (implicit, explicit) or (explicit, implicit), which is closest to an array-like implementation, where the location of a value is calculated from the one of the attributes;
- (implicit, implicit), which describe a pure functional association;
- (explicit, explicit), which is used to support a non-predominant access pattern.

Within each scheme there are ample opportunities to further optimize towards CPU processing or storage cost. For example, for sparse domain and range of an association an (explicit,explicit) scheme can be augmented with two search structures to obtain a fast retrieval and minimal storage.

Goblin does not insist on a single storage method for a particular BAT. As long as two implementations provide the same interface, they can be interchanged freely. One of our main goals is to hide these alternative implementations behind the BAT interface description and to exploit the differences as best as possible without interference of the user. That is, Goblin adapts the implementation using statistics about BAT usage.

The adaptive algorithm lets the BAT automatically select among the internal representation that is best under the given circumstances. Therefore, each BAT implementation includes cost functions to help making decisions. In particular, the BAT programmer should supply storage-, search-, and transport- costs functions. Balancing the requirements is captured by a single adaptation routine, which is time-, query-, update-, or user- triggered. Once called, it compares

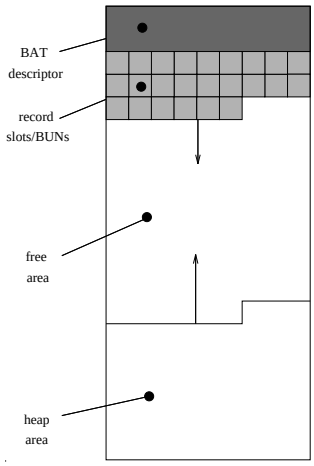


Figure 4.8: The BAT memory layout

the weighted cost of the current implementation against the benefits of an alternative representation. If needed, it will convert the BAT to its new storage structure. It may also decide to keep multiple incarnations around to satisfy conflicting usage patterns at the cost of additional update overhead.

Currently, only the (explicit,explicit) scheme is implemented, which creates indices at run time to speed up the relational operations. The algorithms of most binary operations are hash-based and they first construct a hash index on one operand. This hash-index is then retained until it is invalidated by an update operation. Consequently, most operations have significantly different *hot* and *cold* execution times. For instance, the cold and hot execution times of joining two 10k binary relations are respectively 120 ms and 90 ms[3].

The memory layout of the BAT is depicted in Figure 4.8 and consists of three areas: a descriptor, a fixed size area of record slots and finally a heap space.

The descriptor records the current state of the BAT, which consists of its name, the attribute types, the available indices, information on the average attribute value size and the cardinality. Furthermore, the descriptor maintains pointers to identify the first free record slot and the top of the heap area.

The record slots store the fixed size BUNs. For variable sized attribute values, for instance strings, the BUN contains a reference to the string, which is allocated in the heap area. To facilitate BAT relocation or storage on disk, references are represented by offsets relative to the BAT address.

4.4.7 The storage layer

The previous sections have been focussed on object and schema representation, partitioning and access structures. BAT persistency, stability and consistency are not addressed by the schema, summary, and data layer. For these issues we

[3]Measured for the Goblin kernel V2.0 on a SGI R3000/Irix 4.05 running at 33Mhz

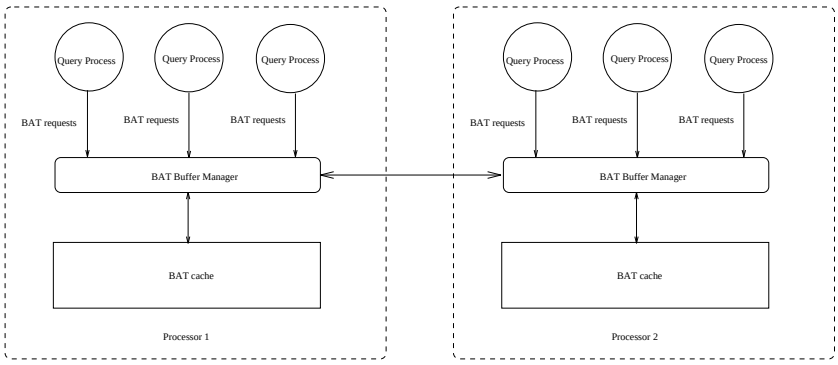


Figure 4.9: The storage layer

partly rely on the evolution of file systems for distributed operating systems. We believe that novel operating systems offer transaction management primitives, use replication techniques to increase availability and stability, and perform load balancing for coarse grain objects (i.e. files). In the current implementation this functionality is implemented by a separate storage management process, as it is not yet offered by the Amoeba or IRIX operating systems.

On each processor there is a BAT Buffer Manager or BBM, which manages part of the collection of BATs. The BBM layer offers a mechanism for creating, updating, and destroying globally accessible and persistent objects in general and BATs in particular. Furthermore, basic transaction support functions are provided.

The BBM stores the global persistent BATs in a local memory buffer and controls its contents. Its task can be compared to that of the buffer manager in disk based systems. The main difference is that if a BAT is not locally available, this results in retrieval of a copy of the BAT from another BBM process instead of from disk.

The BBM uses a buffer replacement policy that favors frequently used objects in order to reduce the number of buffer misses. As the Goblin Query Scheduler has an overview of what data is required on each processor, the BBM offers primitives to control the buffer replacement policy. The global setup is depicted in Figure 4.9.

An efficient method for obtaining persistency in main-memory data-base systems is achieved by using stable memory for maintaining the log records and using disk to store the latest checkpoint [LC87]. Alternatively, persistency and data consistency of BATs can be achieved efficiently through data replication if the following conditions are satisfied:

- Network partition failures do not occur. As the Goblin project goal is realization of a parallel DBMS, instead of a globally distributed DBMS, the processors are interconnected through a dedicated, reliable network (In the prototype a single ethernet connection). In the event of a network

failure all processing is delayed at the cost of decreased availability until the network is up again.

- The number of simultaneous site failures does not exceed the replication degree of BAT. The probability of information loss can be reduced to an acceptable level given the probability of a site failure at the cost of memory consumption and speed of data updates.

The direct consequence of these assumptions is that the replication control algorithm, which maintains the *one-copy-serializability* property, can be based on the simple *read-one-write-all-available* (ROWA) protocol. Quorum based algorithms do not have to be considered, because network partitions are not assumed.

The prime target of a persistent storage layer is to maintain global database consistency despite system failures. The storage layer design requires an analysis of the different causes resulting in a transaction failure. Given the Goblin architecture the possible causes are an *application abort* or a site failures.

An application abort is generated by a user interrupt or generated by the application code. The detection of such a failure is therefore straightforward. The abort will be reported to the process, which coordinates the global transaction, which will then initiate a global abort procedure. The BBM storage layer then undoes all the updates made by the application and rolls back to the previous consistent state. This roll-back functionality is provided by the BAT interface.

A site failure is the result of a bug in the application software or system software, or the result of a hardware problem. Such a failure is detected by the BBM layer, when it fails to update or access a replica on a remote site. This kind of error should be detected before a data request is made because multiple site failures could remain undetected. If a site failure occurs while a transaction is in progress, the transaction is aborted. Furthermore, to maintain the minimum replication degree the data which were stored on the the crashed site are distributed over the remaining sites. Thus one of the replicas becomes a primary copy. When the crashed site recovers, it will gradually absorb data through data migration.

BATs, which partake in a global transaction are updated atomically using a 2PC[4] protocol. Each transaction is assigned a unique identifier, which is then used to store the recovery information of the objects. The global commit or abort decision is recorded and propagated to all sites. (As the global log maintaining the transaction status is also a globally accessible persistent object, the same mechanism as for ordinary objects is used to store the transaction administration.)

During its lifetime, a globally unique identifier is associated with a BAT. BATs are referenced by this identifier. If a BAT is not present in the local buffer, it is retrieved from another site using its identifier.

BATs can be moved or copied by the BBM layer from one processor to another. To transport the BAT it has to be converted to a representation, which does not use local memory addresses. This functionality is offered by the *marshal* routine

[4]two phase commit

provided by the BAT interface, which converts the BAT to a byte sequence. The routine *unmarshal* converts the byte sequence to its BAT representation.

The basic mechanism for obtaining BAT persistency in Goblin is replication. To keep the replicas consistent with each other, the replicas need to be updated if one of them is changed. Replicas are updated by only transmitting the recent updates, which are maintained by the BAT. Replicas can then be updated by sending the log to the replica and replaying it on the BAT replica with a log redo routine. In case of a transaction abort the updates can be undone with the log undo routine. The log must therefore also contain the old values.

4.5 Conclusion

This chapter discussed the alternative storage models for object representation in a main memory context. Many of the issues discussed are similar to those for NF^2 relations, like clustering and declustering. We have, however, also discussed a few issues that are typical for object oriented systems, like object sharing and object dynamicity.

Given the assumptions on Goblin applications, the decomposed storage model becomes the prime choice, because it allows an efficient support of object sharing, and object updates, and still has low storage overhead.

Finally, the design of the Goblin storage model was presented. The binary relations resulting from the DSM approach are a priori partitioned into fragments, which are declustered over the available processors. A novelty in this approach is the use of a summary data base. This data base allows query processing to be performed in two phases. The first phase runs a query on the summary data base, and serves as a dynamic optimization step by selecting fragment combinations, which potentially contribute to the query result. These fragment combinations are executed in the second phase. In a distributed system, however, this evaluation can be performed in parallel.

Chapter 5

Dynamic Query Processing

5.1 Introduction

Static query processing schemes (SQP) as described in Chapter 1 generate a single query evaluation plan (QEP), taking optimization decisions on the basis of the statistics available at compile time. Thus, the query schedule and allocation topology of subqueries to processors is fixed for the duration of the query evaluation. This often leads to a suboptimal execution due to unreliable or outdated cost estimates or to an impractical exploration of the space of feasible QEPs. In this Chapter we introduce an alternative processing technique called dynamic query processing (DQP) as a possible solution to this limitation.

The prime objective of the DQP scheme is similar to those of the SQP schemes. Namely, minimization of the query response time, not only for queries run in isolation, but also for a workload of concurrent running queries. The DQP scheme is an alternative to achieve these goals in view of two important problems in parallel query processing: coming up with a reliable estimate of intermediate result sizes and predicting the load distribution accurately for the run-time of the query. These problems are tackled in a DQP architecture with two mechanisms: a feedback mechanism to reduce the amount of work and a load balancing technique to avoid congestions in query pipelines.

In a DQP scheme some of the optimization decisions are taken at query processing time on the basis of the feedback information. An abstract DQP architecture consists of two components: a *Query Scheduler* and a *Query Evaluator* (See Figure 6.1). The *Query Scheduler* controls and drives the query execution by constructing query evaluation plans.

Each QEP is subsequently executed by the *Query Evaluator*. This component

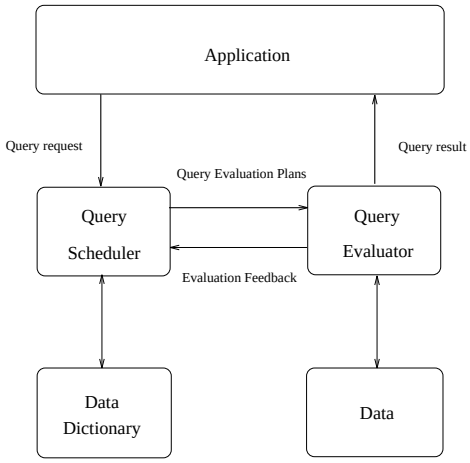


Figure 5.1: The general DQP architecture

can actually consist of several processes which can either evaluate several QEPs in parallel or use pipelining and data parallelism to execute individual QEPs in parallel.

At specific points in the execution of a QEP the Query Evaluator sends feedback information to the Query Scheduler. Such as information on the load distribution and intermediate result sizes observed. The Query Scheduler then decides to keep the current QEP, or change it for the remainder of the query. The overall query evaluation strategy determines when information on the query evaluation is feed-back.

In the remainder of this chapter we discuss three alternative evaluation strategies for dynamic query processing. Thereafter, we give a short overview of related work and, finally, we describe the application of DQP to the Goblin parallel OODBMS. The detailed discussion of the Goblin query processing architecture can be found in Chapter 6.

5.2 Query evaluation strategies

An essential characteristic of a DQP architecture is that a query is not processed in its entirety, but in subqueries or steps. After each subquery the query evaluation plan for the next subquery is re-considered. Therefore, the overall query evaluation plan can be adjusted at run time to adapt to variations in the data- and load distribution. For this purpose important performance parameters, like the sizes of intermediate results and the processor load are monitored and feedback to a query scheduler. The scheduler can then optimize the query schedule and allocation plan.

An important issue in the design of a DQP is the processing *granularity* of these subqueries. If the granularity is small the level of control provided to the Query Scheduler is large. The disadvantage of a small granularity is the

devastating effect on the scheduling overhead, which eliminates the performance improvement from dynamic query optimization. The granularity can range from the individual operations, which introduce a lot of control overhead, to the complete query, which basically corresponds to the SQP scheme with a query-abort and re-run facility.

Another issue is the query *decomposition* technique. There are two orthogonal approaches for this. Basically, they correspond to the methods used for query parallelization, namely pipelining and task spreading. In this context we adopt the terms *query step* and *data step* to stress that the primary objective is not to parallelize the query, but to introduce control points in the query process, where feedback information on the execution is returned to the Query Scheduler. These methods are briefly discussed in the subsequent sections using a (simple) query to illustrate the differences.

Example 5.1 Given relations R, S, T and U we consider the following 4-way join query:

$$Q \quad = \quad R \bowtie S \bowtie T \bowtie U$$

5.2.1 The query step approach

The query step approach divides the query tree produced by the parser into subqueries such that they can be executed in a pipelined fashion. For dynamic query processing, however, control operations are inserted at some points in the query tree to direct the run-time optimization and load balancing. This operator tests whether the intermediate result size differs too much from threshold values set at query optimization time. In that case the remaining part of the query is re-optimized using the new information [BR88, Ngu81]. With this technique the smallest granularity obtained is a decomposition into subqueries around a single relational operator.

The advantage of this approach is that it can be applied to the standard SQP schemes. For instance, Graefe defines a *choose-plan operator* to insert control points in a QEP [GW89]. At run-time this operator evaluates cost formulas to choose between alternative query evaluation plans. The disadvantage of this approach is that the alternative QEP are determined at query compilation time. Furthermore, it is difficult to decide where to insert these choose-plan operators in the query tree. Adding too many choose-plan operators leads to large QEPs and reduce the amount of pipeline parallelism.

Example 5.2 The query example can be evaluated using different join orders. Which order to choose can best be determined at run-time after each join. In the following we assume that the optimizer produced two feasible join orders $(R \bowtie S) \bowtie (T \bowtie U)$ *and* $((R \bowtie S) \bowtie T) \bowtie U$*. The scheduler takes the run-time decision on the basis of the cardinality of the intermediate result* $T_1 = R \bowtie S$*. On four processors this could lead to the following execution:*

site	*subquery*
P_1	$T_1 = R \bowtie S$

If $|T_1| > threshold$*:*

site	*subquery*
P_1	$T_2 = T_1 \bowtie T$
P_2	$Q = T_2 \bowtie U$

If $|T_1| \leq threshold$*:*

site	*subquery*
P_2	$T_2 = T \bowtie U$
P_1	$Q = T_1 \bowtie T_2$

Note that in the evaluation only two of the four processors are used. Without the control point this query could have been evaluated on three processors and exploit pipeline parallelism.

5.2.2 The data step approach

In the data step approach the relations are partitioned such that the query is replaced by the union of independent subqueries. These subqueries or tasks are subsequently executed by the Query Evaluator. After each subquery execution the Query Evaluator sends feedback information to the Query Scheduler. The smallest granularity is achieved when the partitioning degree equals the cardinality of the relations. In this case each fragment consists of a single tuple and the subquery just test whether a certain combination of tuples satisfies the query constraint.

The advantage of this approach is that the tasks can be executed in parallel. Furthermore, because the allocation of tasks to processors is not fixed at query compile time and there are a large number of tasks, load balancing is easy to achieve.

After each task evaluation, the Query Scheduler uses the feedback information to reduce the number of tasks remaining, and in a parallel system to adjust the task allocation.

Example 5.3 The sample query is replaced by a large number of similar query tasks. If each relation is partitioned into two fragments and the query evaluator uses four processors, then the query can be evaluated as follows:

site	*task*
P_1	$T_1 = R_1 \bowtie S_1 \bowtie T_1 \bowtie U_1$
P_2	$T_2 = R_1 \bowtie S_2 \bowtie T_1 \bowtie U_1$
P_3	$T_3 = R_2 \bowtie S_1 \bowtie T_1 \bowtie U_1$
P_4	$T_4 = R_2 \bowtie S_2 \bowtie T_1 \bowtie U_1$
P_1	$T_5 = R_1 \bowtie S_1 \bowtie T_1 \bowtie U_2$
P_2	$T_6 = R_1 \bowtie S_2 \bowtie T_1 \bowtie U_2$
P_3	$T_7 = R_2 \bowtie S_1 \bowtie T_1 \bowtie U_2$
P_4	$T_8 = R_2 \bowtie S_2 \bowtie T_1 \bowtie U_2$

site	*task*
P_1	$T_9 = R_1 \bowtie S_1 \bowtie T_2 \bowtie U_2$
P_2	$T_{10} = R_1 \bowtie S_2 \bowtie T_2 \bowtie U_2$
P_3	$T_{11} = R_2 \bowtie S_1 \bowtie T_2 \bowtie U_2$
P_4	$T_{12} = R_2 \bowtie S_2 \bowtie T_2 \bowtie U_2$
P_1	$T_{13} = R_1 \bowtie S_1 \bowtie T_2 \bowtie U_1$
P_2	$T_{14} = R_1 \bowtie S_2 \bowtie T_2 \bowtie U_1$
P_3	$T_{15} = R_2 \bowtie S_1 \bowtie T_2 \bowtie U_1$
P_4	$T_{16} = R_2 \bowtie S_2 \bowtie T_2 \bowtie U_1$

Note that in the task assignment the fragment distribution is taken into account. The tasks that are consecutively allocated to a processor change only in a single fragment, which minimizes the I/O.

5.2.3 Query restart

In this coarse method the query is evaluated as a unit and interrupted if a threshold specified for a monitored resource is exceeded at run-time. The decision to re-consider the query plan at run-time results in a query abort, followed by the generation of a new query plan, and a restart of the query evaluation.

The advantage of this scheme is that a large collection of query plans is considered, because at each restart the original query is re-optimized using up-to-date statistics obtained from the aborted query.

The disadvantage is that when an interrupt occurs in the middle of an execution, only partial information on intermediate result sizes can be feed-back to the scheduler. Furthermore, intermediate results, if they exist, are not re-used. And finally, it is not clear whether the performance gain from the improved QEP justifies the work invested in the first try.

Example 5.4 For our query there exist many alternative query evaluation plans. Consider the first query evaluation plan to execute the query according to the join order $(R \bowtie S) \bowtie (T\ JoinU)$*. In this evaluation both pipelining and task spreading is used to improve the response time. The join operations are executed in parallel on different processors as follows:*

site	*subquery*
P_1	$T_1 = R \bowtie S$
P_2	$T_2 = T \bowtie U$
P_3	$T_3 = T_1 \bowtie T_2$

The resource consumption is monitored during query execution. If it exceeds a compile time determined threshold, the query is aborted and the execution information is feedback to the scheduler.

If it turns out that the cardinality of the intermediate result produced by $T \bowtie U$ *exceeded a limit and caused the query abort, the query scheduler generates a new query evaluation plan based on this information and restarts the query execution so that this join operation is performed last:*

site	*subquery*
P_1	$T_1 = R \bowtie S$
P_2	$T_2 = T_1 \bowtie T$
P_3	$T_3 = T_2 \bowtie U$

Note that the intermediate results are not re-used. Furthermore, only three of the available four processors are used.

5.3 The Goblin approach

The DQP scheme presented in this thesis is based on the data-step approach. The relations are partitioned into fragments a priori. The query result is then the union over the subquery results for all fragment combinations. This choice is motivated by considering that:

- In the query step approach, misjudgements in the initial subquery evaluation can not be undone, because the intermediate results are already generated. The effects are carried over to the remainder of the query.

- The data step approach results in a large number of independent tasks. In a parallel system this increases the level of parallelism to be exploited and it can easily be adjusted to match the requirements of the data base application, such as the rate at which data can be consumed by the user program.

- We use a main-memory system where the data is partitioned such that the task can be executed in the memory of a single processor.

An important concern in the design of this query processing scheme is the number of tasks. If the relations are partitioned in too many fragments, it leads to a large number of task evaluations and schedule overhead.

For instance if a relation R_i is partitioned into n_i fragments then $\prod_i n_i$ similar subqueries have to be evaluated [1]. Given a large multiprocessor platform these subqueries can, in principle, be evaluated in parallel. However, the speedup will be limited due to the speed by which the fragment data can be prepared or distributed over the processor pool.

Furthermore, sequential evaluation (or limited parallel evaluation) creates an opportunity for dynamic query optimization; it is possible to reduce the amount of work using statistics of previous query task evaluations and semantic knowledge of the query operations. In a system based on the query step, the individual subqueries are generally not of the same form, in the data step approach they are similar. Before presenting the Goblin query processing scheme in detail we briefly address the related work on dynamic query processing.

5.4 Related work

The two main aspects that influence the efficiency of the query evaluation plan are data distribution and load distribution. In a dynamic query processing architecture the *query optimization* scheme reduces the effects of data distribution variations on the execution time and the *load balancing* scheme tries to adapt the query process allocation to variations in the load distribution.

The following two sections summarize the results from related work.

[1] Depending on the query this number can be reduced by choosing a suitable partitioning function. For instance, using hash-based partitioning of the operands of an equi-join operation reduces the number of subquery evaluations to $\lceil \frac{n_1}{h} \rceil \lceil \frac{n_2}{h} \rceil$, where h is the number of hash buckets.

5.4.1 Query optimization

Bodorik and Riordon [BR88] and Nguyen [Ngu81] propose a scheme based on a *threshold* mechanism. This scheme basically follows the query step approach where the query plan is corrected when the actual size of a partial result exceeds the estimated size by a certain threshold value.

Graefe and Ward [GW89] introduce the notion of Dynamic Query Evaluation Plans to solve the problem of producing query plans for parameterized queries. Query execution involves evaluation of a decision procedure for the actual query constants and the data distribution. Thereafter the components of an access module are dynamically linked to obtain an appropriate execution plan. They primarily focus on access methods, but their approach is also applicable to parallel query processing. Actually, in this approach the query evaluation plan is not adjusted at run-time, but the decision to choose an alternative query evaluation plan is delayed until query startup.

Another approach is used in the XPRS shared memory DBMS. The query optimizer produces an optimized sequential QEP, which is parallelized at query startup time. However, after query startup the QEP can not be changed.

5.4.2 Load balancing

Lu and Carey [LC86] present a task allocation algorithm to balance the system load and to minimize the communication cost. It shows that load balancing leads to significant reductions in the average time a query task waits for I/O and CPU resources.

Murphy [Mur89] focussed on performance improvement for query execution on shared memory multiprocessors using a minimal number of processors and a limited amount of database buffers. The method is based on scheduling page reads and page join operations efficiently.

Similar to the approach of Murphy, we consider query evaluation as a scheduling problem. First, the query is transformed into a query program, which solves the query for a portion of the database at a single processor. Second, the relations involved are partitioned into fragments. Finally, combinations of these fragments form query tasks, which are executed on the available processors by a centralized scheduler. The query scheduler controls the load balancing and it performs logical query optimization using up to date information on query task execution and the availability of fragments. Our dynamic query processing scheme aims at improved processing of pre-compiled parameterized queries, which exhibit large potential parallelism or none at all depending on the parameter settings upon query execution.

In a pilot study of our approach [vdBKSA91] we focussed on load distribution in this system. Specifically, we tried to identify the bottlenecks in the system architecture through a simulation and a subsequent validation on the PRISMA 100-node shared nothing multi-processor [AKO88]. We observed that in this first design the query evaluator formed the bottleneck. The overhead incurred by using a centralized scheduler to manage the load distribution was negligible in our distributed store environment, due to the subquery cost. These encouraging results lead to further research which is presented in this thesis.

5.5 Conclusion

In this chapter we have outlined the basic objectives and techniques to achieve Dynamic Query Processing. The basic idea is to postpone optimization decisions and adjust query evaluation plans at run-time. We argued that the main characteristics for a DQP architecture are the query step method and the granularity of the resulting subqueries.

We presented two orthogonal approaches to query decomposition for dynamic query processing: the query step and data step approach. In the first approach a QEP for part of the query is produced and executed. On the basis of its result a QEP for the next part of the query is produced. The second approach partitions the involved relations so that the query is replaced by the union of independent similar subqueries.

The granularity specifies the amount of query processing performed before a query evaluation plan is re-considered. It determines the level of control the scheduler has on the query evaluation.

In this thesis we investigate a DQP architecture based on data step and a small granularity because it facilitates exploitation of parallelism and run-time optimization. In the data step approach a small granularity leads to a large number of independent subqueries. The reduction of the number of subqueries is therefore a major research issue addressed in this thesis .

In the following chapters the Goblin OODBMS is presented. Attention is paid to its language aspects and storage model, but the main focus is its dynamic query processing architecture.

Chapter 6

The Goblin Query Processing Scheme

6.1 Introduction

The Goblin query-processing architecture is based on the assumption that the Goblin applications handle large amounts of similar data. A query is translated into a set of relational operations that process data set-at-a-time. Efficient support of navigational access, where an application retrieves and updates data by visiting individual objects through their attributes may require a totally different object representation scheme and query processing architecture. In such systems the performance for data access is increased by clustering objects frequently used together and by maintaining index structures for frequently evaluated path expressions [BK89].

In the traditional approaches a query schedule is generated at query compile time (static query processing). The query optimizer uses cost functions to chooses an optimal schedule from a large set of possible query schedules.

The cost formulas are mostly based on the number of distinct values and cardinality of attributes. With these statistics and under the assumption that the attribute value approaches a uniform distribution, the ordinality and cardinality after applying a relational operation can be estimated. However, if the data distribution is skewed, the error introduced can be significant [Loh89].

Furthermore, if the query consists of a large number of operations, the error component is increased at each operator leading to a totally unreliable estimate, and, therefore, a questionable optimal schedule. The decomposed storage model of Goblin (See Chapter 4) has the effect that objects have to be reconstructed from binary relations. The result is that queries tend to contain a large number of joins. For instance, if a query requests all n attributes of objects of a subset

of a class, n semi-join operations should be performed to retrieve the associated attribute values.

Another problem which is expected to be more pronounced in an object oriented system is the unpredictability of the system load, which could lead to a situation where one processor sits idle, while another forms the bottleneck in the query pipeline. The main reason is that basic types with their operations can be added to the system. For simple types like integer numbers and strings, the CPU cost of an operation is easy to determine using profiling, and to predict. However, when for instance an image type is added to the system, there can be a large variation in the processing time required for its complex operations. It can also be that the operation is performed remote by a special server.

To overcome the effect of data skew and the non-uniform load distribution Goblin uses a DQP scheme as presented in Chapter 5. We expect that the load balancing and dynamic query optimization scheme leads to a better system utilization and query response time. We will first describe the Goblin DQP architecture and then give an overview of query processing in Goblin.

6.2 The Goblin architecture

The Goblin architecture is modeled after the general DQP architecture presented in Chapter 5. The global architecture consists of three types of processes: a single Query Scheduler, a pool of Query Processors, which correspond to the Query Evaluator in the general DQP architecture, and an equal number of Buffer Managers (See Figure 6.1), which provide global data access.

The Query Scheduler receives query requests from an application program and drives and controls the query execution by generating subqueries or tasks and distributing these tasks dynamically over a pool of Query Processors. It uses a load balancing scheme to minimize the average task execution time. Furthermore, it implements a task elimination algorithm to optimize the query execution process using feedback information on task results.

The Query Processor executes the tasks in main-memory and it assists the Query Scheduler by sending feedback information on the average task execution time and occurrence of empty intermediate task results. The Query Processor obtains the fragments for the task from its local Buffer Manager process. The Buffer Managers together store the database and offer the Query Processors a globally accessible and persistent fragment store.

In the next subsections these components are discussed in more detail.

6.2.1 Buffer Manager

On each processor there is a Buffer Manager, which maintains part of the data base ensuring data persistency through data replication. If one of the processors crashes, the system can continue by using one of the replicas stored on another site. When the processor starts up again after a system crash, it can recover its data using the replicas managed by the other buffer managers or stored on disk.

During query execution data is copied on request and transmitted between the buffer managers on the network. Each buffer manager uses a significant amount

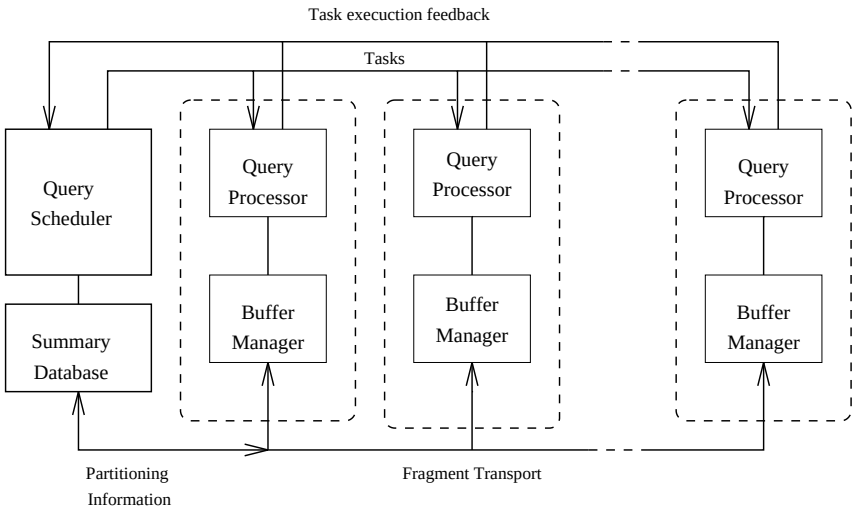


Figure 6.1: The Dynamic Query Processing architecture

of its processors main-memory to store the fragment copies and to maintain replicas.

The buffer contents is determined by the tasks that are being executed and by the buffer replacement policy. The fragments that are required by a task are fixed and can not be removed from the buffer. Several buffer replacements strategies can be considered, ranging from random strategies to traditional LRU algorithms. This choice is closely related to the task allocation algorithm used. In Chapter 9 this issue is examined more closely by comparing several combinations of task allocation and buffer replacement algorithms.

The buffer managers migrate or replicate fragments to distribute the fragment references evenly and to spread the storage for persistent fragments over the available processors. Fragment migration and replication is not controlled by the Query Scheduler. The goal of fragment allocation and replication in the Goblin architecture is to improve the fragment availability and access time over multiple queries, while the QS is only concerned with the efficient execution of a single query. The fragment allocation information is available to the QS to achieve this. The QS can then allocate subqueries over the available processors, so that it results in a minimal amount of data transport.

The buffer manager offers a standard set of transaction management primitives, such as shared/exclusive read/write locks on fragments and a two-phase commit protocol.

The data allocation and replication problem are not addressed in this thesis. Furthermore, the transaction management issues, logging and recovery are also considered to be outside the scope of this thesis. However, based on our experience on transaction management and recovery issues in the PRISMA project [vdBK90], we think that they can be solved satisfactorily.

6.2.2 Query Processor

The Query Processors (QP) form the engine of the query evaluation process. For each task the QP first tries to obtain the referred fragments from its local buffer manager. If the data is not available, the buffer manager retrieves the data from the remote sites.

In contrast to the SQP execution model, the execution order of the individual operations in the query is not fixed at compile time. Instead, all feasible evaluation plans are considered by the QP. The task evaluator of the QP selects a plan depending on the availability of the fragments and a cost estimate for each plan.

For instance, if the QP is requested by the QS to calculate $Q = R_1 \bowtie S_2 \bowtie T_1$, and fragments S_2 and T_1 are already present, it will first calculate $S_2 \bowtie T_1$, and store the intermediate result for further use. When fragment R_1 arrives, it completes the join operation, and informs the QS that it has evaluated the task $Q(R_1, S_2, T_1)$. The task-evaluation algorithm is discussed in detail in Chapter 10.

A task monitor keeps a record of the average task execution time, and of events, which are interesting for dynamic query optimization, like the occurrence of empty intermediate results. This information is feedback to the QS.

6.2.3 Query Scheduler

The functionality of the Query Scheduler is implemented by three subprocesses: the *Generator*, the *Allocator*, and the *Optimizer*. These processes communicate and coordinate their actions through the data structure called the *Task Table*. The *Generator*, *Allocator* and *Optimizer* use this data structure to store new task descriptions, select tasks for execution, and change or remove task descriptions, respectively. Figure 6.2 presents the global structure of the QS. In the following paragraphs, the functionality of the main processes is described.

The Generator initiates and drives the query-execution process by producing new tasks using the partitioning information stored in the summary database. This partitioning information combined with the query specification determines which fragment combinations might contribute to the query result. In essence it simulates the actual query on the summary database. The generated tasks are queued for execution in the *Task Table*.

The Optimizer performs logical optimizations of the query at run time. It uses its knowledge about the dependencies between the operators and operands in a query and the statistical information from task executions to remove or eliminate tasks from the *Task Table* data structure. An example of a logical optimization is task elimination. In this technique for each query a set of elimination rules is defined. Consider a four-way join query $Q = R \bowtie S \bowtie T \bowtie U$, where the relations R, S, T and U are partitioned into fragments. For this example the following rules can be derived:

$$
\begin{aligned}
|R_i \bowtie S_j| = 0 \quad &\longrightarrow \quad \forall_{x,y} |R_i \bowtie S_j \bowtie T_x \bowtie U_y| = 0 \\
|S_i \bowtie T_j| = 0 \quad &\longrightarrow \quad \forall_{x,y} |R_x \bowtie S_i \bowtie T_j \bowtie U_y)| = 0 \\
|T_i \bowtie U_j| = 0 \quad &\longrightarrow \quad \forall_{x,y} |R_x \bowtie S_y \bowtie T_i \bowtie U_j)| = 0
\end{aligned}
$$

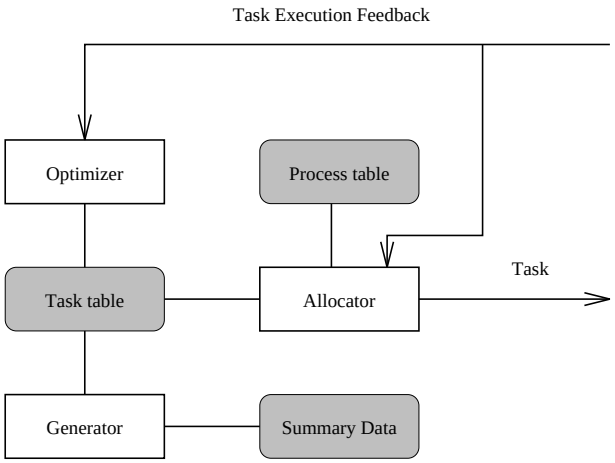


Figure 6.2: The main components of the Query Scheduler

Thus, if the result of a task execution is empty, because of an empty partial join (eg. $R_1 \bowtie S_7$), then all the other tasks with this fragment combination (viz. $Q(R_1, S_7, T_x, U_y)$ will not contribute to the final query result and, therefore, do not need to be executed. This technique is discussed in detail in Chapter 8.

The Allocator is responsible for the load control and load balancing of the query evaluation. It selects tasks from the *Task Table* and assigns these tasks to the available Query Processors. For task selection the Allocator can use the fragment allocation information. For the selection of the processor site, the load distribution of the QPs, maintained in the *process table*, is also taken into account. This information is updated by task feedback information from the Query Processors.

These functionalities of the QS, task generation, task elimination (optimization) and task allocation are essential for the whole system performance. Therefore, they are addressed separately and in detail in Chapters 7, 8 and 9 for task generation, task elimination and task allocation, respectively.

Summarizing, the Goblin query processing architecture is designed to support the following features:

- The query scheduler provides a solution to the unpredictability of the load distribution, and at the same time uses the summary data base to exploit skewed data distributions.

- The task generation and task elimination processes reduce the large number of tasks resulting from the DQP scheme based on the data decomposition approach and a small task granularity.

- The task allocation process reduces the total number of fragment I/O requests by taking into account the fragment distribution.

- The query processor operates data driven to effectively handle strong fluctuations in the fragment arrival rate and dynamically optimize the query task on the basis of measurements and the available resources.

Having presented the Goblin architecture, we will now look at the query evaluation process in more detail, and specifically, discuss the first step in query execution namely, the translation of Goblin queries into an internal representation.

6.3 Query processing overview

In Chapter 3 we introduced the derived-class concept as the basic mechanism to query the Goblin data base. Each derived class defines a view on the data base through which the objects can be accessed and updated using the methods on the composing objects. Thus, the objects in the derived class reflect the current state of the data base. Once the application program applies a method to the derived class, the class definition is interpreted as a query on the classes from which it is derived.

The general Goblin query process is decomposed into three pipelined subqueries: an optional *split* subquery, an obligatory *process* subquery, and an optional *merge* subquery. The pipelined subqueries act as a filter and assembly line for the objects that enter the pipeline at one end and leave the pipeline at the other. These query pipelines can be further combined into networks of pipelined queries.

In the split subquery the binary relations involved in the query process are partitioned to ensure independent subqueries on the partitions in the query phase, and to reduce the amount of work in the *process* subquery. The split subquery is rarely necessary as the binary relations are generally already partitioned on both attributes in the binary relation.

In the next phase, the *process* subquery, the actual query is evaluated for all the effective fragment combinations. Whether a fragment combination is effective, or contributes to the query result can be checked by executing the query on the summary data of the fragments.

The final phase, or *merge* subquery is used for global operations. In a sort operation, for example, the merge phase merges the sorted fragments. For aggregate operations, the merge phase combines the intermediate results produced in the query phase.

The decomposition into query phases is performed at compile time, and corresponds to the heuristics used in query optimization in (parallel) relational database machines. Performing selections before the remainder of the query, and data partitioning to reduce the amount of work are examples of these heuristics. The main difference with query decomposition in relational database systems is that the bulk of the work is performed in the query phase by executing a combination of relational operations for each fragment combination. Thus, within each phase, only data parallelism is exploited. Furthermore, the task throughput of each phase can be dynamically adjusted, so that the input and output rates of the pipelined phases are balanced.

In this thesis we consider only the *process* subquery. We assume that the data is already partitioned, so that the split phase can be omitted. Furthermore, as we do not consider the translation of aggregate queries, i.e. the merge phase is also absent. In the following, the term query refers therefore to the process subquery.

The decomposed storage model is also used to represent the query result. The final assembly of objects from their composing binary relations is therefore left to the application process. The rationale for this approach is that the application will only infrequently need access to the whole object. The overhead for reconstructing part of the object at the application's site is low compared to transporting a fully reconstructed object from the database to the application process. In general the result consists of objects of a single class.

The result of the query evaluation is a set of binary relations that contain enough information to construct objects with the type of the derived class that satisfy the selection condition. Only when an external method (i.e. a C or C++ function) is applied to the objects of the derived class, the objects are reconstructed to the level required by the method.

The next sections describe the generic derived class, the translation process, the scheduling and task execution in greater detail.

6.3.1 The derived class

From the definition given in Section 3.3 we know that the general derived class definition consists of a *type-specification*, *binding-list*, and a *constraint*. In the following we describe the syntax of a derived class specification, using a BNF notation. Non-terminals are enclosed in brackets (⟨⟩). Terminals are specified in small caps for keywords (KEYWORD) and in italic script for other terminals.

⟨*derived-class*⟩ : CLASS *class-name*
TYPE ⟨*type-specification*⟩
FROM ⟨*binding-list*⟩
WHERE ⟨*selection-condition*⟩

The *type specification* determines the structure used to store the query result. It specifies which attributes from the binding list appear in the query result. In the following we will deal only with a special case, namely when all the attributes from the binding list are kept.

The *binding list* defines the name and object domain for each attribute. The attribute ranges over the associated class extent, and its type is implied by the class specification. Without the type specification, the binding list implicitly defines the extent of the derived class to consist of tuple objects, where each attribute ranges over its associated class extent.

The attribute values range over the class extents associated with the class names in the binding list. Without a selection condition the binding list class extent of the derived class consists of all the possible attribute value combinations.

The class names which appear in the binding list, can also be derived classes. The first time a derived class is used it is materialized. In the following we only consider the case that the class names in the binding list are materialized classes.

$$\begin{aligned}\langle \textit{binding-list} \rangle \; : \; & \textit{class-name attribute-name} \\ & \mid \textit{class-name attribute-name ','} \langle \textit{binding-list} \rangle\end{aligned}$$

The *selection condition* specifies the state constraint that holds for all the objects in the derived class extent. In materializing the class extent the selection condition is used to construct a query on the stored binary relations.

The basic building blocks for a condition are *path expressions*, *restriction terms*, *join terms* and *boolean terms*. Currently, we do not consider function and method calls. The complete condition is then an boolean expression over one or more terms.

$$\langle \textit{selection-condition} \rangle \; : \; \langle \textit{boolean-term} \rangle$$

$$\begin{aligned}\langle \textit{boolean-term} \rangle \; : \; & \langle \textit{join-term} \rangle \\ & \mid \langle \textit{restriction-term} \rangle \\ & \mid \langle \textit{boolean-term} \rangle \text{ AND } \langle \textit{boolean-term} \rangle \\ & \mid \langle \textit{boolean-term} \rangle \text{ OR } \langle \textit{boolean-term} \rangle\end{aligned}$$

$$\langle \textit{join-term} \rangle \; : \; \langle \textit{path-expression} \rangle \; \langle \textit{comp-op} \rangle \; \langle \textit{path-expression} \rangle$$

$$\langle \textit{restriction-term} \rangle \; : \; \langle \textit{path-expression} \rangle \; \langle \textit{comp-op} \rangle \; \langle \textit{constant} \rangle$$

$$\begin{aligned}\langle \textit{path-expression} \rangle \; : \; & \textit{attribute} \\ & \mid \textit{attribute} \cdot \langle \textit{path-expression} \rangle\end{aligned}$$

$$\langle \textit{comp-op} \rangle \; : \; \text{'}<\text{'} \mid \text{'}\leq\text{'} \mid \text{'}==\text{'} \mid \text{'}=\text{'} \mid \text{'}\geq\text{'} \mid \text{'}>\text{'}$$

The *path expression* specifies the objects that can be reached starting from the object referenced by an derived class or target attribute and traversing the structure of the object. The path specifies the traversal by concatenating the attribute names encountered. Note that as there exist single-valued and set-valued attributes, the result of the path expression represents a single object or a set of objects.

The basic predicates consists of the equality operations $=$ and $==$, which test for deep equality and object identity respectively, and the comparison operations $<, \leq, \geq, >$[1]. The default comparison operations are defined for the base objects.

[1]Currently we do not consider the equality operation, because it requires an exact match which implies that two objects have exactly the same properties. Furthermore, testing deep equality is an expensive operation, which requires determining the sets of base objects reachable from two objects, and testing whether the two are equal.

Composite objects are compared on their object identifiers, which have a system defined ordering.

The condition is built out of two kinds of basic terms: join terms and restrictions terms. The join term is an expression which relates two class attributes. In general the expression consists of two path expressions and a basic predicate. A restriction term specifies the minimal range of values for a path expression, using a basic predicate and a constant value.

The steps in the query translation process will be illustrated using the following example query taken from the Goblin language report [KvdBS+93].

Example 6.1 The class `Mail` *collects all letters sent by children to their parents living in Paris. As the class is defined in terms of existing classes, this is an example of a derived class.*

```
TYPE Letter=TUPLE(Person sender, receiver; STR text);

TYPE Mail= TUPLE(Letter l; Person p, c);

CLASS mail
TYPE Mail
FROM person p, person c, letter l
WHERE c IN p.kids AND l.sender == c AND l.receiver == p
       AND p.address.city == 'Paris'
```

6.3.2 Query translation

Although Goblin provides a sophisticated set of language constructs for data access, the translation of queries to query programs is rather straightforward. This stems from the object representation model, which enables us to represent query specifications on complex objects using simple binary predicates.

Furthermore, the Goblin query processing architecture is based on the assumption that query optimization can be done effectively at run-time. The execution order of relational operations is therefore delayed until run-time, and based on the cardinality or availability of the operands.

The translation and evaluation of the generic derived class is performed in the following steps:

- The class definition is transformed into a query graph, which represents the constraints that exist between the attributes of the derived class as specified by the class constraint.
- This query graph is used by the task generation program of the query scheduler. The query graph is interpreted on the summary data. The QS generates for this query graph fragment combinations which (can) contribute to the query result.
- The tasks generated by the scheduler are specified by the query graph and a set of fragment identifiers. The fragment identifiers are associated with

edges in the graph. The query processor interprets the query graph using this fragment assignment.

- Finally, for each method call in the application program, code is generated to access the attributes referred to by the method. The query result is delivered by Goblin to the application program as a set of binary relations. For externally defined functions the query result is transformed to the application language specific data structures. This aspect is not further addressed in this thesis.

The next section defines the components of a query graph. Furthermore, it discusses the transformation of the generic derived class into a query graph representation. This is specified by the operator T which transforms the basic syntactical components: path expressions, restriction terms, join terms and selection conditions, and boolean terms into graph components.

6.3.3 The query graph

The query graph representation is not a new concept or restricted to the specification of relational queries. Our query graph resembles the one used by Gardarin in [GGdM89] to direct the translation of logic programs to relational expressions. Both approaches consider only binary predicates (relations).

Gardarin associates with each rule in the logic program a query graph. Each variable is represented by a node and each predicate by an edge in the graph. In his article he discusses the transformation process of this query graph to a set of fixpoint equations and, finally, to a relational algebra program. His main concern, however, is to translate recursive logic programs.

Goblin queries are not recursive, but are simply select-project-join (SPJ) queries. The first step in the translation is to put the selection condition in a conjunctive normal form. Each conjunct defines a part of the query result, so that the total query result is formed by the union of the query result of each conjunct. In the following we consider the translation of a conjunct.

Example 6.2 The Mail example is already in a conjunctive normal form. The selection condition is therefore given by the following rule:

$$\begin{aligned} Mail(p,c,l) \quad :- \quad & kids(p,c) \wedge sender(l,c) \wedge receiver(l,p) \\ & \wedge address(p,a) \wedge city(a,x) \wedge equal(x,'Paris') \end{aligned}$$

The predicates kids, sender, address, city and receiver are defined by their corresponding binary relations. The variables p,l and c are the projection attributes specified by the type definition. The x variable is a free variable which ranges over the string attribute domain of the city relation. Note that the rule is never recursive. In other words the goal predicate (Mail) does not occur in the rule body.

In the next step this rule is translated to a query graph. A query graph is identified by a four tuple $G = \langle N, E, A \rangle$. The set of nodes $N = \{1, \ldots, n\}$

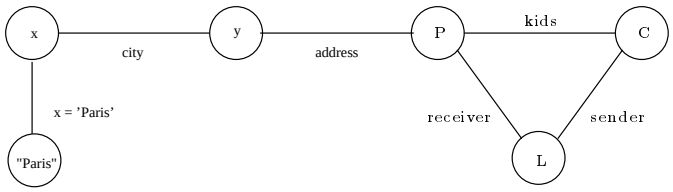


Figure 6.3: The query graph for the Mail query

corresponds to sets of data base objects. Some of these nodes are projection attributes others are anonymous attributes like the x attribute in the previous example. The set of projection attributes is given by $A \subset N$.

The constraints between objects is represented in the graph by the set of edges $E \subset N \times N$. An edge corresponds to an existing binary relation used by the storage model, or to a condition. This can be a selection predicate between a set of objects and an atomic object or a join predicate between two objects.

The number of edges connecting a node is called the *degree* and denoted by $d(x)$. It is defined by: $d(x) = |\{(e_1, e_2) \in E || e_1 = x \vee e_2 = x\}|$.

6.3.4 Query graph construction

The translation of a generic derived class specification to a query graph is straightforward. For each of the basic language constructs we define the translation to a query graph. The operation which maps the language constructs to a graph is denoted by T.

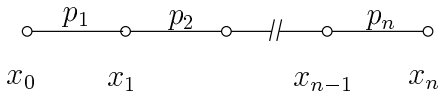
Example 6.3 The query graph for the Mail example query is presented in Figure 6.3. It illustrates the translation of path-, restrict-, and join expressions. Furthermore, note that the resulting query graph is cyclic.

6.3.4.1 Path expressions

A path expression denotes a traversal through the object graph. Applying the path expression to a specific object or set of objects, it defines the objects that can be reached.

The attributes in the path expression correspond to binary relations and are mapped onto edges. The intermediate nodes visited when traversing the object graph along the specified path are anonymous and correspond to the intermediate objects visited. Let $x_1, \ldots, x_n$ denote these intermediate anonymous nodes. Then the translation of a path expression is given by:

$$T(p_1 \cdot p_2 \cdots p_n) \longrightarrow$$



Path expressions do not occur in isolation, but form part of restriction- and join-terms. In these situations the path expression is bound at one side to a

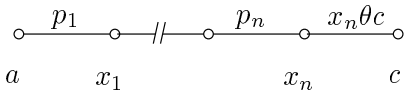
class attribute, and specifies traversals through the object graph starting from objects in the attribute domain.

6.3.4.2 Restriction terms

A restriction term selects objects from the attribute domain that satisfy a selection condition. The attribute range is commonly determined by a path expression and consists therefore of those objects that can be reached by traversing the path through the object graph.

A restriction term can be considered to be a special kind of binary operation, where one of the operands is a single object. To facilitate the translation process selection expressions are treated similarly to join expressions. If the selection predicate implies a range restriction, a condition edge is created labeled by the selection condition and connected to a node representing the constant.

$$T(a \cdot p_1 \cdot p_2 \cdots p_n \theta c) \longrightarrow$$



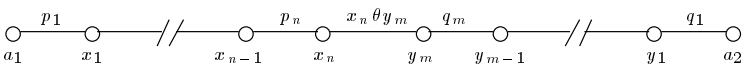
The attribute a is added to the set of projection attributes. This information is used in the task generation and task execution phase to determine the query result.

6.3.4.3 Join terms

A join term selects those object combinations from two target attribute domains that satisfy the join condition. The join condition is generally expressed between objects that are associated to the attribute domain through path expressions.

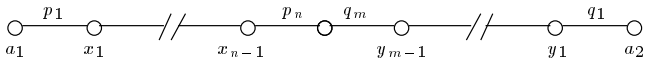
The join condition can be considered to be a virtual binary relation, which is not stored, but can be derived by evaluating the join condition on two sets of objects. In the object graph perspective, a join term selects those pairs of objects that are connected by two paths reaching from both objects and linked by an edge representing the join condition. The construction of the query graph is straightforward. Starting from each attribute, a sequence of edges is created. These edges are linked by unique intermediate nodes denoted by $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, to represent the intermediate objects in the object graph. The edges can be labeled either by the attribute names specifying the path, or by two node pairs (x_i, x_{i+1}). In any case, each real edge is associated with a binary relation. The join term is labeled by the condition θ that holds between the objects associated with nodes x_n and y_m.

$$T(a_1 \cdot p_1 \cdots p_n \theta a_2 \cdot q_1 \cdots q_m) \longrightarrow$$



If the join condition expresses equality of the object sets reachable from the attributes, the condition edge can be omitted as edges that meet in the query graph already imply equality of the domains associated with these edges.

$$T(a_1 \cdot p_1 \cdots p_n == a_2 \cdot q_1 \cdots q_m) \longrightarrow$$



Both attributes a_1 and a_2 are added to the set of projection attributes.

6.3.4.4 Boolean terms

Boolean terms are built from restriction- and join-terms using the boolean operations. Because the boolean terms are specified in conjunctive normal form, the translation consists simply of transforming the join-terms and restriction terms in subgraphs and combining them into a single query graph, consisting of one or more connected components.

If the query graph consists of two or more components, they are not connected by an edge, and therefore are the possible combinations not restricted by a constraint. The query result is then simply the Cartesian product of the result of each connected component. For the remainder we assume that the query graph consists of a single component, which is not a severe restriction. Because in general queries are used to produce meaningful information by combining data.

6.4 Conclusion

In this Chapter we have presented the Goblin dynamic query processing architecture. This architecture is modeled after the general DQP architecture presented in Chapter 5. It consists of a query scheduler, query processor and buffer manager.

This architecture is characterized by the novel two-level query processing scheme. The query is evaluated in two phases. One level consists of the query scheduler and the other of the query processor pool. The query scheduler first evaluates the query on a summary data base containing fragmentation information. Then the tasks produced by the QS are executed on the second level by the query processors. This scheme facilitates the implementation of a DQP architecture and enables the separation of optimization issues. This leads to a flexible and adaptive processing structure.

The query scheduler influences the number of I/O requests by its task allocation algorithm. It tries to reduce the total number of tasks with its task generation algorithm and task elimination algorithm. The query processor is only concerned with CPU optimization and memory utilization. These issues are discussed in detail in the subsequent chapters.

Furthermore, we introduced the query graph which is the internal query representation. The query graph forms the basis for summary query evaluation in the task generation algorithm and task execution in the query processor. Finally, we showed the translation of a query specification to a query graph.

Chapter 7

Task generation

7.1 Introduction

The Query Scheduler drives the query execution by generating tasks, assigns these tasks to the available query processors, and coordinates the transport of results to the application.

Task generation is based on running the query against the *summary database* as defined in Chapter 4. The summary database maintains for each fragment of a binary relation its identification and abstract information on the attribute values contained. In case of range-partitioned relations this consists of the minimum and maximum values for each attribute. For hash-partitioned relations it is simply the hash value.

The task generation algorithm queries the summary information for all the relations involved and selects those fragment combinations that can contribute to the query result. For this purpose the relational operations join, semi-join, and select have been defined for summary relations (See Section 4.4.3).

The summary data base should reflect the actual data base partitioning. Any change in an underlying binary relation must be propagated to the corresponding summary relation before being used. In this thesis we assume that the workload consist of read-only queries. Therefore, the overhead for maintaining the consistency of the summary data is not considered. Under a read-only workload the summary data base can be replicated to all available processors and maintained at low cost at run-time to reflect changes in the data base partitioning. The main cost factor taken into account for task generation is the CPU cost.

In a parallel system task generation and task execution are run in parallel. In a single processor environment the summary query is run as a batch job before the

tasks are executed. These approaches are referred to by the terms *navigational* and *batch* query, respectively.

The summary query is evaluated using the query graph defined in Chapter 3. The edges in the query graph are associated with their corresponding summary relations. First the selections specified in the query graph are applied. This is a good heuristic to reduce the cardinality of the operands involved in the remaining operations. Secondly, an execution order for computing the summary query is determined. The key to the solution of the summary query is based on a graph algorithm for the Chinese Postman Problem. The derived execution order is finally used in the third phase by the batch- or navigational- task generation algorithms. The next sections discuss these phases in more detail.

7.2 Notation and terminology

In this chapter we consider summary queries defined by a *query graph* $G = \langle N, E, A, card \rangle$ consisting of a single connected component (See Chapter 6). The query graph defines a constraint on the data base consisting of summary relations. The edges in the graph correspond either to summary relations or to selection or join conditions. Given an edge $e = (x, y)$, the associated relation is denoted as $R(x, y)$ and the associated condition is denoted as $C(x, y)$. The nodes in the graph correspond to sets of partition ranges or in the case of hash partitioning, to hash values. The set A identifies the nodes associated with the projection attributes.

The summary relations have been introduced in Chapter 4. They maintain for each fragment of a binary relation its unique identification *bid* and for each attribute, the attribute value range or the hash value. To simplify the notation, the summary information on both attribute values is named s_1 and s_2 for both range- and hash-partitioned relations. A summary relation R denoted by: $R[bid, s_1, s_2]$.

Example 7.1 To illustrate these concepts we use the Mail query defined in Example 6.1. In Figure 7.1 we show the corresponding query graph and illustrate its relation to the summary database. We assume that all the relations are hash-partitioned on both attributes. The summary relations maintain therefore for each fragment the hash values for both attributes. The domain of node x is defined by the hash-values stored in the `city` *summary relation. For node L it is determined by the intersection of the summary relations* `receiver` *and* `sender`.

The join, theta-join, and select operation have been redefined to process summary relations. The main distinction with the common operations is that they are based on a comparison operation on the partition information. For instance, when joining two summary relations of range partitioned relations, two fragments are considered equal if their attribute ranges overlap. To distinguish these operations from the ordinary relational operations, they are denoted as $\overline{\sigma}, \overline{\bowtie}, \overline{\ltimes}$ and $\overline{\bowtie}_\theta$ for the selection, join, semi-join, and theta-join operation, respectively.

The objective of summary query processing is to identify fragment combinations that (potentially) contribute to the query result. Therefore the summary query result is formed by combinations of fragment identifiers associated with

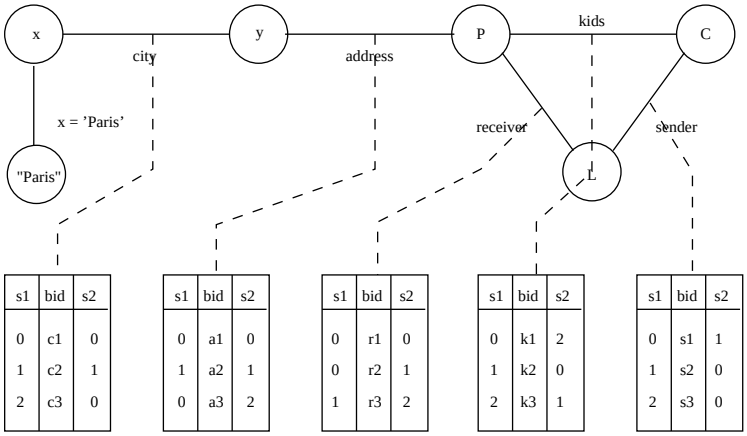


Figure 7.1: The query graph and its associated summary relations

relation edges. For summary query processing the query graph is therefore transformed into a more appropriate representation, the *join-index* graph. This will be explained further on. First the graph initialization is discussed.

7.3 Query graph initialization

During the initialization phase the edges in the query graph are bound to a set of summary relations. It involves a *binding* phase and *selection* phase.

In the *binding* phase the target attributes nodes, that are not yet connected to relation edges are *bound*. This means that each of these target attribute nodes is connected through a relation edge to a node representing its domain. This relation edge is associated to the summary relation representing the attribute's domain.

In the *selection* phase condition edges are successively removed from the graph. Performing selections first is generally a good heuristic, because these operations reduce the operands of remaining operations and are relatively cheap. Finally, the edges in the query graph are labeled by the cardinality of their associated summary relations.

Let the edge $(c, x) \in E$ correspond to a condition edge, where c is the node corresponding to the constant, and x the adjacent node and let $(y_1, x), \ldots, (y_k, x)$ denote the *relation* edges incident on x. Furthermore, the condition associated with the edge is referred to by $C(c, x)$. Then the following program evaluates the selection condition:

FOR i FROM 1 TO k DO
    $R(y_i, x)$:= $\overline{\sigma}_{C(c,x)} R(y_i, x)$
DONE

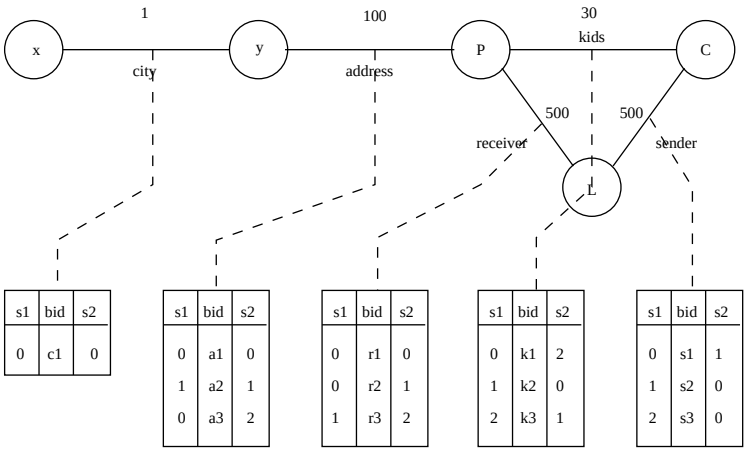


Figure 7.2: The initialized Mail query graph

The removal of a condition edge implies the evaluation of the selection condition against all the binary relations $R(y_i, x)$. The cardinality statistics of the updated relation edges is then adjusted.

Example 7.2 The binary relations in the Mail example are hash-partitioned. In the selection phase the condition edge connected to node x is removed, leading to the evaluation of the condition x = 'Paris' against the summary relation `city`. *Consequently, the summary relation is reduced to a single entry (See Figure 7.2).*

Fortunately, all the projection attribute domains in the example are defined by the summary relations associated with the relation edges `receiver`, `sender`, *and* `kids`. *Therefore, the query does not have to be evaluated on the complete object domain defined by the summary relations* `Letter` *and* `Person`.

After the initialization phase the summary relations have the following cardinality.

relation	*name*	*cardinality*
[Person,Person]	*kids*	*30*
[Letter,Person]	*receiver*	*500*
[Letter,Person]	*sender*	*500*
[Address]	*Address*	*80*
[Address,String]	*city*	*1*

7.4 The Chinese Postman Problem

The solution to the summary query must satisfy the constraints specified by the grounded and modified query graph. Each solution is identified by a combination

of object identifiers. If we denote these identifiers by o_i and $(o_1, \ldots, o_n)$ identifies a solution, then their associated objects satisfy the relations in the query graph. In other words there is a path connecting these objects, which passes through each edge of the graph at least once.

Obviously, given a graph there are many paths that traverse all the edges. However, as traversing the path implies finding the associated set of objects at each node, there is a cost involved in the traversal. In general the traversal of edges require joining the current set of objects with the binary relation associated with the edge. In the worst case the join result is the Cartesian product of these two. We assume that the cost is dominated by the cardinality of the binary relation associated with each edge. This assumption is true if the resulting object set is smaller or has approximately the same cardinality as the original after traversing the edge. Otherwise, the cost is dominated by producing the result.

If the join condition is an equi-join and represents a $1-1$, or $1-n$-ary relationship, the result cardinality will be at most the maximum of the involved relations. Furthermore, as the summary data base is relatively small the query cost is considered acceptable.

The query problem reduces under this assumption to finding a route from one node to another through a connected graph that uses each edge at least once. This problem is also known as the *(undirected) Chinese Postman Problem* (CPP) in Graph theory. The solution to this problem is given in [GM84][pp.340-344]. For brevity we will only give an outline of the algorithm and refer to the textbook for a complete discussion and correctness proof.

Algorithm 7.1 The basic idea of the algorithm is to find an Eulerian cycle in the graph. If the graph is not Eulerian, then edges are added between nodes of odd degree to make the graph Eulerian. The algorithm is performed in the following steps:

1. *If the graph is Eulerian, the CPP cycle is simply the Euler cycle through the graph. By definition the Euler cycle traverses each edge only once.*

2. *If the graph is not Eulerian, the graph is made Eulerian by adding chains between pairs of nodes of odd degree. Let $X \subseteq N$ denote the set of nodes of odd degree.*

3. *Use the shortest path algorithm to find the shortest path between each pair of nodes in X (and its length).*

4. *Construct the complete graph $\mathcal{K}(X)$, where each edge is labeled with the length of the shortest path connecting the nodes.*

5. *Determine the perfect matching with minimum weight on $\mathcal{K}(X)$. The resulting set of edges correspond to chains in the original graph. Furthermore, the chains are the shortest chains that can be added to make the graph Eulerian.*

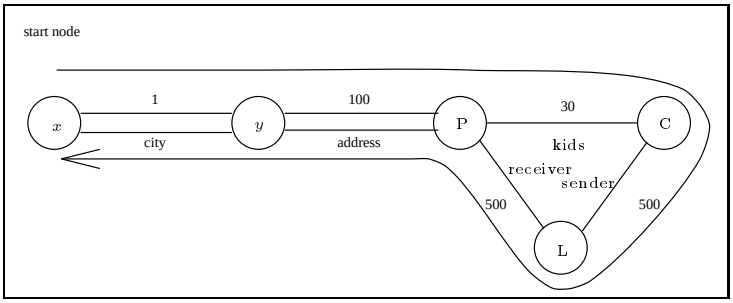
6. *Determine the Euler cycle through the modified graph.*

With this algorithm a path through the query graph can be found that uses each edge at least once. We refer to it as the CPP path.

Example 7.3 In the initialized Mail query graph the selection on city is performed and the graph is adjusted accordingly.

1. *First we note that the graph is not Eulerian. Thus in the CPP path some edges will have to be traversed more than once.*
2. *The set X of nodes of odd degree is given by x and P. To make the graph Eulerian, a chain consisting of existing edges connecting these nodes must be added to the graph.*
3. *Calculating the shortest path between each pair of nodes in X is trivial. This path is given by the edges* CITY *and* ADDRESS.
4. *The complete graph $\mathcal{K}(X)$ consists of a single edge connecting the nodes P and x with length* $100 + 1$.
5. *The perfect matching with minimum weight on $\mathcal{K}(X)$ is trivially the edge (x, P). The query graph is made Eulerian by duplicating the edges* CITY *and* ADDRESS *corresponding to the shortest path (x, P).*

This algorithm results in the following CPP path:



The CPP path fixates the evaluation order for the operations required to calculate the summary query, but the choice of the starting edge is free. Obviously the edge corresponding to the smallest relation is chosen as the starting edge.

Note that once in the evaluation all the edges have been traversed, the remainder of the CPP path can be skipped. For instance, in the previous example once the evaluation has reached node `P` for the second time all the edges have been traversed once. Consequently, traversing the relation edges `address` and `city` a second time does not affect the query result found thus far.

In the following sections two evaluation algorithms are presented, which use the CPP path to calculate the fragment combinations that potentially contribute to the query result.

Both algorithms do not use the query graph immediately, but rather they transform it first into a join-index graph, which is a more convenient representation of the problem. In this transformation relation edges are mapped onto individual nodes. Furthermore, each pair of adjacent edges on the CPP graph

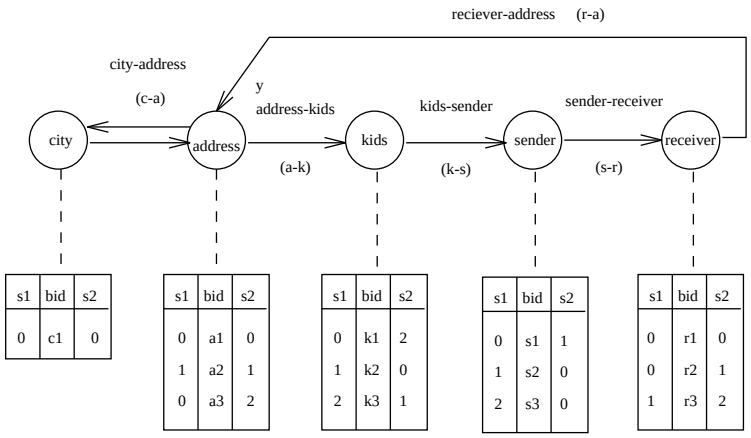


Figure 7.3: The join-index graph for the Mail query

is represented by an edge in the join-index graph. The nodes are labeled by the name of the original edge and the new edges are labeled by the concatenation of the original edge labels. Condition edges in the query graph are mapped to edges in the join-index graph. Because selection conditions have already been removed from the query graph the condition edges are always adjacent to two relation edges.

The summary query result consists of fragment combinations that (potentially) contribute to the query result. These combinations are represented by a set of binary relations, called *pivot* relations, which consist of a *pivot attribute* and a fragment identifier.

Example 7.4 The join-index graph for our example is illustrated in Figure 7.3. The join indices are initially undefined and are generated during summary query evaluation by joining the summary relations of two adjacent nodes. This will be illustrated in the next section.

7.5 Batch task generation

The batch algorithm is divided into two phases: the *initialization* phase and the *pivot* phase.

In the *initialization phase* the edges and nodes in the join-index graph are associated with binary relations by traversing the CPP path. The nodes in the join-index graph are associated with the pivot relations identifying the subset of the summary relation satisfying the query constraints. Furthermore, join indices are constructed and associated with the edges in the join-index graph. An entry in these join index relations represents a fragment combination with "equal" partitioning information.

The CPP path is traversed starting with the node associated with the smallest summary relation, the start node. While traversing the path a pivot relation is

constructed for each node indicating which fragments from the start edge and the current relation lie on the part of the CPP path traversed thus far.

Each pivot relation is basically a projection of the n-ary relation representing the query solution on a unique tag -the pivot- and the fragment identifier. Initially, the solution to the query is represented by the pivot relation associated with the start node. This pivot relation is constructed using the *mark* operation μ, which extends a relation with a unique tag field called *tag*. Thus if x represents the start node the initial pivot relation $P_0(x)$ is constructed as follows:

$$P_0(x) = \pi_{[tag,bid]}\mu R(x)$$

Example 7.5 After this first step the join-index graph of the Mail query is extended with a pivot relation $P(city)$. This pivot relation consists of a single entry `(1,c1)` *identifying a fragment of the* `city` *relation.*

The remaining pivot relations are constructed in two steps. First a join index is constructed for the current relation edge and the next relation edge. Secondly, this join index is joined with the current pivot relation. Let x and y represent the current relation node and the relation node to be visited next, respectively. Then given the pivot relation $P(x)$ for relation node x, the join index $R(x.y)$ and pivot relation $P(y)$ are constructed as follows:

$$\begin{aligned} R(x.y) &= \pi_{[x.bid,y.bid]}(R(x) \overline{\bowtie}_{x.s2=y.s1} R(y)) \\ P(y) &= P(x) \bowtie R(x.y) \end{aligned}$$

Example 7.6 The first edge that is traversed in the join-index graph is the `c-a` *edge. The join-index is constructed by joining the summary relations* `city` *and* `address` *on their attributes* `city.s2` *and* `address.s1`*. The resulting join-index consists of two tuples: $R(c\text{-}a) = \{(c1, a1), (c1, a3)\}$, and the pivot relation $P(address) = \{(1, a1), (1, a3)\}$.*

If the edge connecting the two nodes x and y corresponds to a condition edge $C(x, y)$, the join-index is calculated using the join condition. Note that the join condition is defined on the attribute ranges of the summary relations $R(x)$ and $R(y)$. The pivot relation is calculated similar to the previous case.

$$\begin{aligned} R(x.y) &= \pi_{[x.bid,y.bid]}(R(x) \overline{\bowtie}_{C(x.s2,y.s1)} R(y)) \\ P(y) &= P(x) \bowtie R(x.y) \end{aligned}$$

If the traversed edge corresponds to a $1 - n$ relation, many fragments will be associated with the same start fragment. Conversely, if an $n - 1$ relation edge is traversed, the pivot relation will associate many start objects with the same object. To maintain the property that each pivot relation represents a projection of the query result, the duplicate pivot attributes in the constructed pivot relation must be renumbered. This transformation can be represented by

a transformation relation T, which associates a unique new pivot attribute tag' with each duplicate attribute in the constructed pivot relation. This transformation relation is then used to renumber all the pivot relations constructed thus far. Thus after construction of the new pivot relation $P(y)$ the following actions are performed:

$$\begin{aligned} X &:= \mu P(y) \\ P(y) &:= \pi_{[tag',bid]} X \\ T &:= \pi_{[tag',tag]} X \\ \forall_{x \in N} & \quad (P(x) := \pi_{[bid',x]}(T \bowtie P(x)) \end{aligned}$$

Example 7.7 Consider relations $P(x) = \{(tag_1, x_1), (tag_2, x_2)\}$ *and* $R(x.y) = \{(x_1, y_1), (x_1, y_2), (x_2, y_1)\}$. *Then the new pivot relations* $P(x)$ *and* $P(y)$ *are calculated as follows:*

$$P(y) := P(x) \bowtie R(x.y)$$

Resulting in $P(y) = \{(tag_1, y_1), (tag_1, y_2), (tag_2, y_1)\}$. *This pivot relation does not have a unique pivot attribute and must therefore be renumbered:*

$$\begin{aligned} X &= \mu P(y) \\ P(y) &= \pi_{[tag',bid]} X \\ T &= \pi_{[tag',tag]} X \end{aligned}$$

After this renumbering operation the pivot relation looks like:
$P(y) = \{ (tag'_1, y_1), (tag'_2, y_2), (tag'_3, y_1)\}$ *and the transformation relation:* $T = \{(tag'_1, tag_1), (tag'_2, tag_1), (tag'_3, tag_2)\}$. *This relation* T *is then used to renumber the already defined pivot relations:*

$$P(x) := \pi_{[tag',bid]}(T \bowtie P(x))$$

Such that $P(x) = \{(tag'_1, x_1), (tag'_2, x_1), (tag'_3, x_2)\}$. *Note that the resulting pivot relations can indeed be considered to be a vertically fragmented solution to the query represented by* $R(x) \bowtie R(y)$.

Example 7.8 After two steps along the CPP path in the Mail query the pivot relations for `city`, `address`, *and* `kids` *are constructed. The renumbering operation ensures that the common pivot attribute is unique. This is illustrated in Figure 7.4. Note that the summary relations for* `city` *and* `address` *are dropped, because the required information is stored in the pivot relations and join-indices.*

In the *pivot phase* the pivot relations are used to construct all the fragment combinations by joining the pivot relations on the pivot attribute.

Given the pivot sets $P(x_1), \ldots, P(x_n)$, the complete set of tasks $T[bid_1, \ldots, bid_n]$ is found by joining the pivot sets on the unique tag and projecting on the BAT identifiers of the pivot relations.

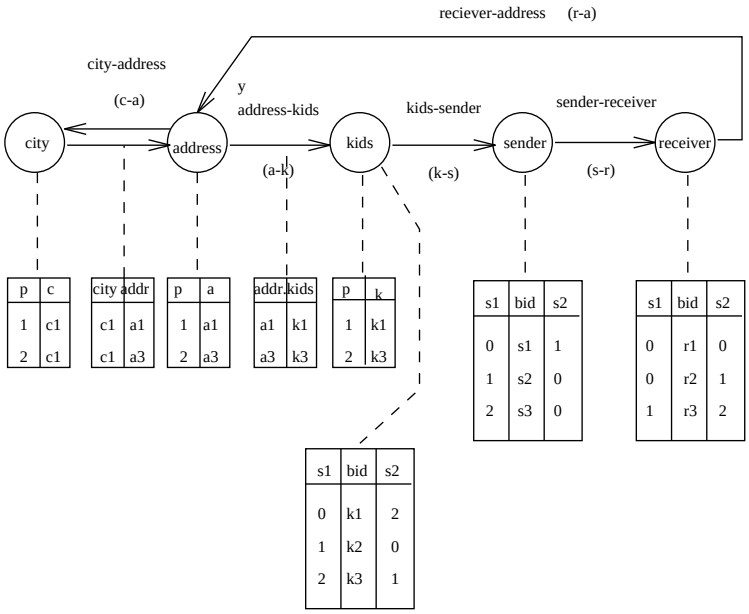


Figure 7.4: The join-index graph after two edge traversals.

$$T \quad := \quad \pi_{[bid_1,\ldots,bid_n]} P(x_1) \bowtie \cdots \bowtie P(x_n) \tag{7.1}$$

Example 7.9 Finally, when all the edges are traversed, all the nodes are associated with a pivot relation. In the Mail example this results in the pivot relations for `city`, `address`, `kids`, `sender`, *and* `receiver`. *This is illustrated in Figure 7.5. Notice that the uniqueness of the pivot attribute has the effect that the calculation of the join expression 7.1 is reduced to a simple lookup operation.*

Summarizing we arrive at the following algorithm:

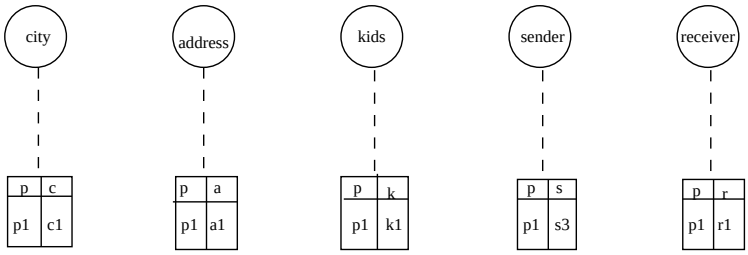


Figure 7.5: The summary query result for the Mail example.

Algorithm 7.2 Given the join-index graph $J = \langle N, E \rangle$ associated with the query graph. Let the CPP path through this graph be defined by the node sequence $(x_1, \ldots, x_n)$. Furthermore, let $|R(x_1)| \leq |R(x_i)|$ for $i \neq 1$. Then the following algorithm calculates the pivot sets for each node.

1. *The node x_1 represents by definition the smallest relation edge. The special operation μ creates the initial pivot relation from the set of fragment identifiers of the summary relation. This operation associates a unique tag value with each fragment identifier.*

$$P(x_1) \quad := \quad \pi_{[tag,bid]} \mu R(x_1)$$

2. *Traverse edges of the CPP path once from the starting node until the end node is reached. For each visited node the pivot set is calculated. In the first traversal a join index is associated with each edge. Let (x_i, x_{i+1}) denote the edge to be traversed, then the following must be taken into account to construct the join index $R(x_i, x_{i+1})$ and pivot set $P(x_{i+1})$:*

   - *The edge (x_i, x_{i+1}) can be associated with a condition edge.*
   - *The pivot set $P(x_{i+1})$ is already defined, because the node has already been visited. In this case the new pivot set is is the intersection of the previous and new one.*

   Thus the following actions are performed in the construction of the pivot sets:

$$\begin{aligned} R(x_i, x_{i+1}) \quad &:= \quad \begin{cases} \pi_{[x_i.bid, x_{i+1}.bid]}(R(x_i) \overtimes_{x_i.s2 = x_{i+1}.s1} R_{x_{i+1}}) \\ \pi_{[x_i.bid, x_{i+1}.bid]}(R(x_i) \overtimes_{C(x_i.s2 = x_{i+1}.s1)} R_{x_{i+1}}) \end{cases} \\ P(x_{i+1}) \quad &:= \quad P(x_{i+1}) \cap P(x_i) \bowtie R(x_i, x_{i+1}) \end{aligned}$$

Example 7.10 After the initialization the tasks are produced using the join expression 7.1 and added to a task table for execution.

7.6 Navigational task generation

The algorithm used in batch task generation produces the tasks in the final phase of the algorithm. For a parallel query processing architecture it is better to perform the task generation and task execution in parallel. For this reason the *navigational task generation* is designed. Basically, it recursively traverses the CPP path and uses the summary relations to create and test possible fragment combinations.

Similar to the object graph associated with the data base, a summary graph can be associated with the summary database. The nodes in this graph correspond to the relation fragments and the edges correspond to the associated fragments, i.e. those fragments that have overlapping attribute domains.

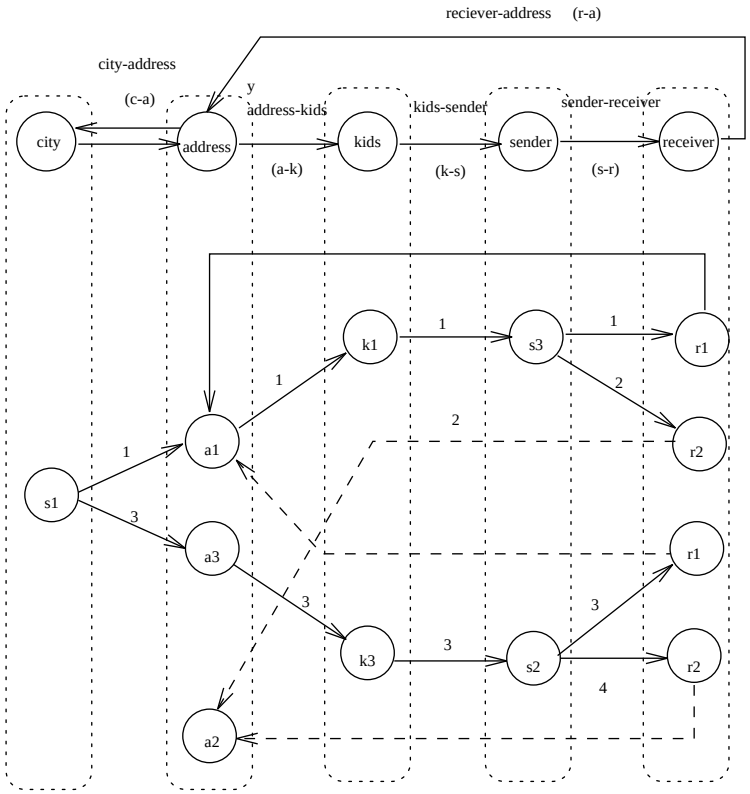


Figure 7.6: The summary graph for the Mail query

The navigational algorithm constructs the query result incrementally by traversing all possible CPP paths through the summary graph. Each fragment in the summary graph, can be associated through a single edge with zero, one, or more fragments. Furthermore, similar to the batch algorithm a distinction must be made between relation and condition edges.

For each path through the summary graph the reachable fragments at each node are maintained. Each time the path revisits a node, the same fragment should be encountered. Let b_i denote the fragment associated with the node x_i, the set of fragments B_{i+1} reachable in the summary graph is determined as follows:

$$B_{i+1} = \begin{cases} \pi_{[bid]}\sigma_{s_1=b_i}R(x_{i+1}) \\ \pi_{[bid]}\sigma_{C(s_1,b_i)}R(x_{i+1}) \end{cases}$$

Example 7.11 The complete summary graph for the Mail example is illustrated in Figure 7.6. The graph is constructed incrementally by traversing the CPP path for each possible fragment combination. The first path starts with fragment `c1`*. From this node, the* `address` *relation fragments* `a1` *and* `a3` *can be reached. The algorithm will now recursively test each combination. As long as a fragment combination is successful, the path is drawn as a solid line. Note that most combinations fail when the* `receiver-address` *edge is traversed. Only a single fragment combination remains.*

7.7 Conclusion

In this chapter we discussed a new and generally applicable query optimization technique. The task generation algorithm produces a series of fragment combinations or tasks that are sufficient to calculate a query result. It is based on simulating the query on a summary data base containing the fragmentation information.

Processing a query in two phases, a query on the summary data base and a query on the data fragments is useful both for query processing in disk-based single processor environments and in a parallel processing environments. In the first case, the summary query is used both as an indexing technique and the tasks produced can be scheduled such that the available buffer space is used optimally. An indication of its advantage can be found in the MCH task allocation algorithm of Chapter 9.

In a parallel environment the profit stems from the reduction of the number of tasks to be executed and from the fact that task generation can be executed in parallel with task execution using the navigational task generation algorithm.

The batch task generation algorithm is more suited for single processor environments as it uses cheap set-oriented operations to determine the tasks.

The DSM storage model made it possible to translate the query processing problem to the graph-theoretical Chinese Postman Problem. The efficiency of this approach relies on the assumption that the relations are properly partitioned, such that the cost for evaluating the primitive (join) operations is not larger than one of its operands and therefore linear in the size of its operands.

The validity of this assumption depends on the partitioning of the relations. At best it results in a $1-1$ relationship between summary relations. For instance, binary relations representing tuple attributes should be partitioned on their tuple OID. A proper partitioning is of utmost importance to keep the number of tasks and, therefore, the summary query cost low.

Chapter 8

Task elimination

8.1 Introduction

The predominant approach towards query evaluation in DBMS is to map a static query evaluation plan onto a processor pool for (data driven) execution. One of the major obstacles for improved performance is the lack of techniques to predict and to avoid resource congestion, which leads to underutilized hardware platforms. A possible solution is to use the standard query optimization techniques to generate a revised query execution plan from scratch at query evaluation time. To control the optimization overhead, the threshold technique can be used to trigger the optimization [BR88, Ngu81]. Alternatively, a range of different query schedules could already be prepared before the query evaluation. The measured query statistics then determine the final query schedule [GW89]. For this purpose *choose-plan* operators are included in the query execution plan.

A novel approach is to use a *dynamic query processing* scheme, which partitions the data base such that the query result is the union over all sub-queries. The sub-queries are distributed for execution by a *Query Scheduler* as independent tasks on a pool of processors. This approach can be seen as a generalization of *associative join processing* [OV92][pp. 470-477], where through horizontal partitioning of the operand relations a join expression is transformed to the union of join expressions on their composing fragments:

$$R \bowtie S \longrightarrow \bigcup_{i,j} R_i \bowtie S_j$$

This scheme has the advantage that information obtained after the execution of each task can be returned to the *Query Scheduler* for load balancing and query

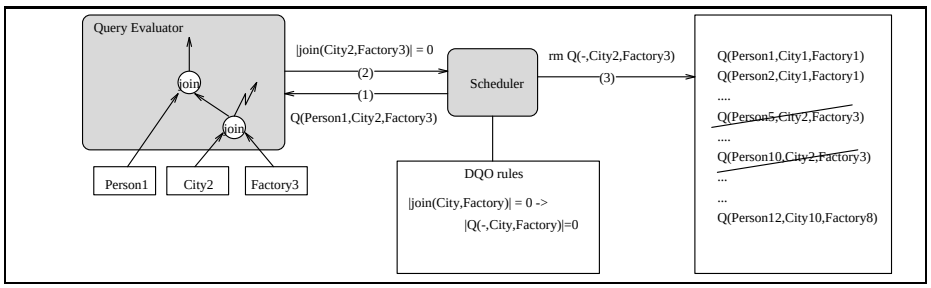


Figure 8.1: Dynamic Query Optimization

optimization. Load balancing is achieved by controlling the allocation of tasks. The query optimization scheme is generally known as Dynamic Query Optimization (DQO) and is defined as the process of modifying the query schedule based on the measurements taken by the *Query Evaluator*.

In this chapter, however, we present a run-time optimization called *task elimination*. This optimization is based on the assumption that the tuples, which partake in the query result are, generally not uniformly distributed over the product space of the relations involved. Instead, they often exhibit some clustering and data skew. As a result the query schedule need not be evaluated for all the fragment combinations. For example consider the following query:

```
SELECT *
FROM Person P, City C, Factory F
WHERE P.address= C.name and C.name = F.location
```

Since the number of factories in a city is variable, there are many (City, Factory) pairs that do not contribute to the query result. Consequently, a large number of (Person, City, Factory) combinations do not have to be considered either. Let $Q(x, y, z)$ denote the sub-query that calculates the example query for the fragments x, y, and z. Then this knowledge can be represented by the following optimization rule:

$$|join(City, Factory)| = 0 \quad \longrightarrow \quad |Q(-, City, Factory)| = 0 \tag{8.1}$$

This rule expresses the fact that if the result of the join between a `City` fragment and a `Factory` fragment is empty, then the query, that uses this combination of fragments is empty *regardless* the contents of the `Person` fragment. The *Query Evaluator* reports the occurrence of empty intermediate result to the scheduler. The Query Scheduler can then perform logical optimizations using this rule, i.e. not taking tasks that contain this combination of fragments into execution (See figure 8.1).

Unlike in static query processing, the execution order of the joins are not fixed in our DQO scheme. This means that the choice between the execution

orders $(Person \bowtie City) \bowtie Factory$ and $Person \bowtie (City \bowtie Factory)$ is made at run-time. Thus, apart from the previous rule the following rule is also provided:

$$|join(Person, City)| = 0 \quad \longrightarrow \quad |Q(Person, City, -)| = 0 \tag{8.2}$$

Whether Rule 8.1 or Rule 8.2 is actually used depends on the execution order chosen at run-time. If both joins are evaluated simultaneously the effect of these optimization rules is combined to reduce the amount of work even further. In Section 4 we investigate the effect of taking all pairwise joins into account. This evaluation strategy is called parallel bottom-up evaluation.

8.2 Relational algebra properties

Dynamic query processing is a novel approach to which old techniques based on algebraic equivalence can be reused. This section focuses on these issues and places these properties in the context of dynamic query processing. In the next section a new technique is introduced and analyzed in detail.

Given the relational algebra expression and semantics preserving rewrite rules, a series of equivalent expressions can be produced for any query expression. In the following we summarize the significance of the communicative, associative, and distributive properties of the relational operations on dynamic query optimization.

8.2.1 Commutative operations

For a commutative operation the result of the operation does not depend on the order of the left or right operand. Interchanging them does not affect the outcome of the operation. An example of such operations are the join, intersect, and union operators.

$$\begin{aligned} A \cup B &\equiv B \cup A \\ A \cap B &\equiv B \cap A \\ A \bowtie B &\equiv B \bowtie A \end{aligned}$$

For dynamic query optimization in a main-memory context this property can be used to reduce the cost for operation evaluation. For instance, in [Bra84] it was shown that hash-based implementation of the relational operations have a superior performance over comparison based implementation schemes. In general these algorithms consist of two phases: a *hash* phase and a *probe* phase.

In the *hash phase* a hash table is built on the first operand. The size of the hash table and the quality of the hash function determines the length of the collision lists associated with each hash entry.

In the *probe phase* the hash table is used by calculating for each elements of the second operand its hash value and evaluating the appropriate action when a matching element in the collision list is found.

Obviously, the cost of the hash phase and probe phase is different and depends on the size of respectively the first and second operand relation. For instance

it turns out that in general it is more efficient to build a hash table on the largest operand and iterate over the smaller operand in the probe phase (See also Chapter 11).

8.2.2 Associative operations

An operator is called associative if the result does not depend on the execution order. Of the relational operations, the join, union and intersection operation are associative. This property is summarized in the following equations:

$$\begin{aligned}(A \cup B) \cup C &\equiv A \cup (B \cup C)\\ (A \cap B) \cap C &\equiv A \cap (B \cap C)\\ (A \bowtie B) \bowtie C &\equiv A \bowtie (B \bowtie C)\end{aligned}$$

The associative property of the join operation leads to the most important issue in query optimization: join order. The join operation is a relative expensive operation. Furthermore, the processing cost depends on the operand size and is in the worst case of the order $O(n^2)$. To determine an optimal join order it is therefore necessary to have reliable estimates of the intermediate results. In the DQP scheme these estimates are made regularly (after each operation) and are therefore more reliable than the estimates that are made only once in the SQP scheme.

In dynamic query processing the associative property of the join operation leads to a design, where the join order is determined at run-time. Consequently, the query specification is general enough to allow many execution orders to be produced.

8.2.3 Distributive operations

The distributive property expresses the fact that an operation can be *distributed* over a specific other operation. In the following equations we see that intersection operation is distributive over the union operator and visa-verse. Furthermore, the join operation is distributive over the union and intersection operation.

$$\begin{aligned}(A \cap B) \cup C &\equiv (A \cup C) \cap (B \cup C)\\ (A \cup B) \cap C &\equiv (A \cap C) \cup (B \cap C)\\ (A \cap B) \bowtie C &\equiv (A \bowtie C) \cap (B \bowtie C)\\ (A \cup B) \bowtie C &\equiv (A \bowtie C) \cup (B \bowtie C)\end{aligned}$$

Query parallelization heavily relies on this distributive property of the join operation. Both in SQP and DQP architectures, a query can be parallelized by horizontally fragmenting the operand relations. Consider for instance the situation where relation A is fragmented into n pieces. The query is then transformed into the following:

$$A \bowtie B = (A_1 \cup A_2 \cdots \cup A_n) \bowtie B$$

Taking advantage of the distributive property it is translated into n join operations to be executed in parallel (once a copy of the relation B is present on each processor).

$$(A_1 \cup A_2 \cdots \cup A_n) \bowtie B \quad = \quad (A_1 \bowtie B) \cup \cdots \cup (A_n \bowtie B)$$

8.2.4 Projection and selection

The projection and selection operation are relatively cheap to perform. Furthermore, they have in common that they reduce their operand relations, either in size (selection) or in the number of attributes (projection).

Consequently, it is generally considered to be a good heuristic to perform projection and selection operations as early as possible. The equations below state a few equivalence relations, that can be used to *push* the projection and selection operators down the query tree. The terms $[f]$ and $[g]$ denote sets of projection attributes. The symbols p and q are used for selection or join predicates. The function $attr$ applied to a predicate or relation identifies the set of attributes used in the predicate or relation.

$$\begin{aligned}
\sigma_p(\sigma_q, R) &\equiv \sigma_{p \wedge q} R \\
\pi_{[f]}(\pi_{[g]} R) &\equiv \pi_{[f]} R \wedge f \subseteq g \\
\sigma_p \pi_{[f]} R &\equiv \pi_{[f]} \sigma_p R \wedge attr(p) \subseteq f \\
\sigma_p(R_1 \cup R_2) &\equiv (\sigma_p R_1) \cup (\sigma_p R_2) \\
\sigma_p(R_1 \bowtie_q R_2) &\longrightarrow R_1 \bowtie_{p \wedge q} R_2 \\
R_1 \bowtie_p R_2 &\longrightarrow \sigma_q R_1 \bowtie_r \sigma_s R_2 \wedge attr(q) \subseteq attr(R_1) \\
&\qquad\qquad \wedge attr(s) \subseteq attr(R_2)
\end{aligned}$$

8.2.5 Semantic properties

The key to dynamic query optimization form the semantic properties of the four basic binary operations: union, intersect, join and difference. In particular, their relationship with the empty set. These properties give rise to two different kinds of optimization techniques: *task simplification* and *task elimination.*

In the task simplification technique, operations are omitted from the task execution schedule, if it is clear that they will not add or remove tuples from an intermediate result. This technique is based on the following properties of the union and set difference operation:

$$\begin{aligned}
A \cup \emptyset &\longrightarrow A \\
A \setminus \emptyset &\longrightarrow A
\end{aligned} \tag{8.3}$$

The Query Scheduler decides upon detection of an empty intermediate result to simplify the query tasks involving these segment combinations, which produce the empty result. This results in a reduction of the task processing time and in a reduction of segment transfer.

Example 8.1 Consider the task T on fragments P, Q, R, and S defined by the following equation:

$$T(P,Q,R,S) = P \bowtie Q \setminus R \bowtie S$$

If during processing it turns out that for a specific combination of fragments (r_1, s_1), that $r_1 \bowtie s_1 = \emptyset$, then all remaining tasks identified by (p_x, q_y, r_1, s_1) can be calculated using the following simplified task expression:

$$T(P,Q) = P \bowtie Q$$

Thus saving both fragment I/O and CPU time to calculate $r_1 \bowtie s_1$.

In practice this technique improves the average task execution time only marginally, as the cost of evaluating these operations, when one of the operands is an empty set, is relatively low. We will not discuss this technique any further.

The following properties 8.4 form the basis of the task elimination technique.

$$\begin{aligned} A \bowtie \emptyset &\longrightarrow \emptyset \\ A \cap \emptyset &\longrightarrow \emptyset \end{aligned} \tag{8.4}$$

Task elimination uses the occurrence of empty intermediate (partial) results to reduce the number of outstanding tasks. For instance, if for the query $S \bowtie R \bowtie T$ the Query Evaluator discovers that the intermediate result $s_1 \bowtie r_2$ is empty, it reports this observation to the Query Scheduler, which then removes all remaining tasks $s_1 \bowtie r_2 \bowtie t_i$ from the task table. Note that in a static/pipelined query processing environment $s_1 \bowtie r_2$ need not be a combination which is executed.

In the remainder of this chapter we will discuss and analyze the effectiveness of this technique in detail.

8.3 Task elimination

In this section we determine the potential savings that can be obtained by task elimination. The effectiveness of this technique is determined by the fraction of empty intermediate results. As the detection of empty intermediate results depends on the evaluation order of the operations in the query, we have also examined the effect of using a left/right-deep tree and our own parallel bottom-up evaluation technique on the elimination factor. The results of this exercise can be found in Section 8.4.

The fraction of empty intermediate results or *elimination factor (e)*, strongly depends on the relational operation and the attribute distribution of the participating relations. The expected value of the elimination factor (e) for a binary operation $\otimes$ between two fragmented relations A and B can be expressed in the probability distribution $P(i,j)$ for empty intermediate results and the number of fragments of the relations n_A and n_B. $P(i,j)$ is defined as the probability that the result of $A_i \otimes B_j$ is empty. Thus:

$$P(i,j) = Prob\{|A_i \otimes B_j| = 0\} \quad (8.5)$$

$$E[e] = \frac{1}{n_A n_B} \sum_{i,j} P(i,j) \quad (8.6)$$

A parameter of importance is the fragment size, because it strongly influences the elimination factor. This can easily be seen by considering the extreme cases. If the fragment size equals the relation size, the elimination factor is zero, because the join result is not empty. If on the other hand each fragment consists of a single tuple, the elimination factor equals $\max(|A|,|B|)/|A| \cdot |B|$.

On one hand we expect the elimination factor to increase as the fragment size decreases, because the probability of an empty result increases, but on the other hand it also increases the number of tasks, which has a decreasing effect on the elimination factor.

Furthermore, the fragment size determines the processing cost of a task and the communication cost for transporting the fragments between processors. Because it is not possible to present a general cost model for relational queries, we have determined the total processing cost for the specific, commonly used, equi-join query.

Before we derive the processing cost for a k-way equi join, we first determine the elimination factor for a single join operation $A \bowtie_{A.a=B.b} B$. Without loss of generality we assume for the remainder of this chapter that the join attribute domain is a subset of the natural numbers. Attribute a is a key attribute of relation A and assumes values in the range $[1, \cdots, c_A]$. The relation A is range partitioned over its key attribute a into n_a fragments A_i containing p tuples each, so that the key attribute a of fragment A_i ranges over the values $[pi + 1, \cdots, p(i+1)]$. The relation B is also range partitioned on its key attribute. We assume that the distribution of the key attribute of B and its non-key attribute b are independent. The fragments B_j also contain p tuples. The attribute value b is distributed according to a certain probability distribution function $\pi(b)$.

To determine the elimination factor $E[e]$, we first express the probability distribution $P(i,j)$ in the probability distribution $\pi(b)$. For this we first determine the *match* probability P_m, which is the probability that the join result of two fragments A and B is not empty. A match occurs if an attribute value b of fragment B lies within the range of key attributes $[pi+1, \cdots, p(i+1)]$ of fragment A. Because the attribute values b are uniformly distributed over the B fragments is the match probability $P_m(i,j) = Prob\{|A_i \bowtie B_j| > 0\}$ independent of the choice of the B fragment. The match probability can then be expressed as follows:

$$P_m(i,j) = P_m(i) = \sum_{b=pi+1}^{p(i+1)} \pi(b) \quad (8.7)$$

Because $P_m(i)$ is the same for all the fragments of B, we find the following expressions for $P(i,j)$ and $E[e]$:

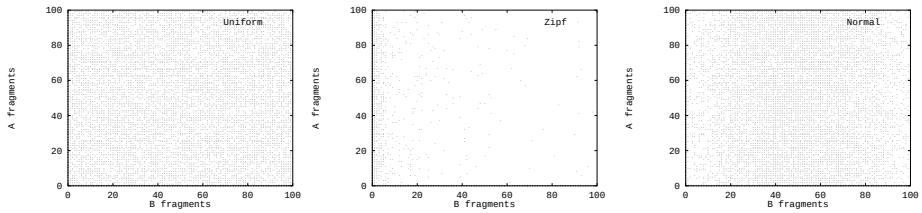


Figure 8.2: Distribution of non-empty join tasks for respectively Uniform, Zipf(0.5) and Normal(5,000;2,500) attribute distributions

$$P(i,j) = P(i) = (1 - P_m(i))^p \tag{8.8}$$

$$E[e] = \frac{1}{n_A n_B} \sum_{i,j} P(i,j) = \frac{1}{n_A n_B} n_B \sum_i P(i) = \frac{1}{n_A} \sum_i P(i) \tag{8.9}$$

In the following paragraphs we calculate the elimination factor for the situation where the foreign key attribute $B.b$ follows the *Uniform*, *Normal* and *Zipf* distribution. Because the query is an equi-join operation on a key attribute, the query result has the same cardinality as the referencing relation B. Therefore the elimination factor can be used to compare the optimization technique for different data distributions.

To show the clustering property of the data distributions, we have calculated equi-join queries for these data distributions on two relations containing 10.000 tuples divided over 100 fragments, and presented the result in scatterplots (Figure 8.2). Each dot represents a non-empty task result. These graphs immediately provide visual evidence of the potential savings of the task elimination for Zipf and Normal join attribute value distributions.

Uniform distribution

The uniform distribution is used to find the worst case behavior for the dynamic query optimization. The reason is that the data contributing to the query result is not clustered, which implies a low task elimination factor for moderately sized fragments. The probability distribution function of the uniform distribution is a constant $\pi(x) = \frac{1}{c_A}$. From this distribution we can derive the following:

$$P_m(i) = \frac{p}{c_A}$$

$$P(i) = \left(1 - \frac{p}{c_A}\right)^p$$

$$E[e] = \left(1 - \frac{p}{c_A}\right)^p$$

Normal distribution

The Normal distribution is also used by Schneider and DeWitt [SD89] in their performance analysis of join algorithms. This attribute distribution is chosen for our analysis, because it could occur in scientific databases for attributes that represent measurement data. The normal distribution $N(\mu, \sigma)$ is defined by:

$$\pi(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp \frac{-(x-\mu)^2}{2\sigma^2}$$

Because this is a distribution of a continuous function we determine the probability $P_m(i)$ as follows:

$$P_m(i) = \int_{pi}^{p(i+1)} \frac{1}{\sigma\sqrt{2\pi}} \exp \frac{-(x-\mu)^2}{2\sigma^2} dx$$

Zipf distribution

In actual databases, the attribute distribution will more likely follow the Zipf distribution [ST89, KNT89]. The Zipf probability distribution function $Z(c)$ for attribute values in the range $[1, \cdots, c_A]$ is defined as:

$$\pi(x) = {H_c}^{-1} x^{-c}$$

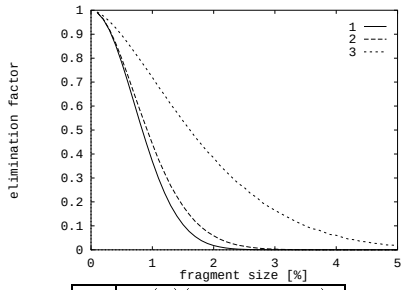
$$H_c = \sum_{k=1}^{c_A} k^{-c}$$

The c parameter is called the decay factor of the distribution. For $c = 0$ the distribution is uniform, if $c = 1$ the distribution equals the classical Zipf distribution. The distribution of personal income follows $Z(0.5)$.

Data distribution comparison

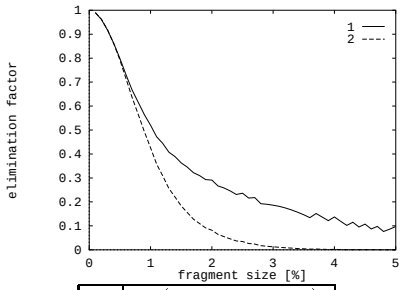
Given the Normal, Zipf and Uniform probability distribution functions and equations (8.8) and (8.9) we have calculated the elimination factor as a function of the fragment size for different distribution parameter settings (See Figures 8.3 and 8.4). The Uniform distribution is included in Figure 8.3, because it is equal to a $Zipf(0)$ distribution.

The graphs show that the elimination factor is a monotonically decreasing function of the fragment size. Furthermore, even the worst case distribution (Uniform) has a potential to reduce the number of tasks for fragment sizes smaller than 2.5 % of the relation size. Finally, we find that the elimination factor is sensitive to the parameters of the distribution. As the attribute distribution becomes more clustered, the task elimination technique becomes more effective over a larger range of fragment sizes (Cf. $Z(0.5)$ and $Z(1.0)$).



1	$Z(0)(= Uniform)$
2	$Z(0.5)$
3	$Z(1.0)$

Figure 8.3: Elimination factor (Zipf)



1	$N(0.5c_A; 0.15c_A)$
2	$N(0.5c_A; 0.25c_A)$

Figure 8.4: Elimination factor (Normal)

8.4 Multiple join evaluation

In a multiple join operation, the occurrence of an empty partial join result will also result in the removal of tasks. In this section the total task elimination E_k of an k-way equi-join is determined given the elimination factors e_i of the $(k-1)$ partial joins. First an expression for the elimination factor for the multiple join is formulated which is then used to calculate the total processing cost for a specific 3-way and 4-way equi-join.

The evaluation order of the join operations has a strong influence on the total elimination factor. We considered two different evaluation methods: *sequential evaluation*, which corresponds to the traditional left-deep and right-deep query tree, and our own method *parallel bottom-up evaluation*.

In the following paragraphs formulas are derived for a general k-way equi-join query. In the analysis each joined relation R_i is partitioned into n_i fragments. For each method we derive a formula for the number of tasks N_k that are removed by the task elimination technique. The total task elimination factor of the join query is obtained through division by the total number of tasks N_{task}:

$$E_k = \frac{N_k}{N_{task}} \tag{8.10}$$

$$N_{task} = \prod_{i=1}^{k} n_i \tag{8.11}$$

Sequential evaluation

In the sequential evaluation method the query is either represented by a left-deep or a right-deep join tree. The intermediate result at each stage of evaluation can be empty (Figure 8.5).

Thus the query evaluator sends the query scheduler information that combinations of two, three, or more fragments result in an empty query result. For a

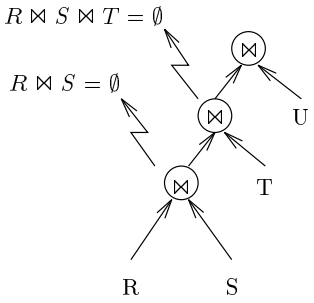


Figure 8.5: Sequential evaluation

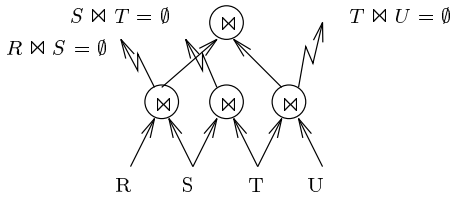


Figure 8.6: Parallel bottom up evaluation

combination of two fragments a large number of tasks can be removed. However, if the combination is more specific, less tasks can be removed. For instance, for a 4-way join, the event $|R_1 \bowtie S_1| = 0$ results in the removal of $n_3 n_4$ tasks[1]. Whereas the event $|R_1 \bowtie S_1 \bowtie T_1| = 0$ reduces the number of tasks only with n_4 tasks. The number of eliminated tasks for a 3-way and 4-way join operation can be determined using the elimination factors of the partial joins e_1 and e_2:

$$\begin{aligned} N_3 &= e_1 n_1 n_2 (n_3 - 1) \\ N_4 &= e_1 n_1 n_2 (n_3 n_4 - 1) + (1 - e_1) e_2 n_1 n_2 n_3 (n_4 - 1) \end{aligned}$$

Generally, of a k-way join $e_1 n_1 n_2$ tasks result in empty $R_1 \bowtie R_2$ combinations, because of the first join operation. This results in $e_1 N_{task}$ task eliminations. The next operation results in $(1-e_1)e_2 N_{task}$ eliminations, caused by $(1-e_1)e_2 n_1 n_2 n_3$ empty task results. Summing all terms until the $(k-2)$-th join operation we find for N_k, the number of tasks that are not evaluated:

$$\begin{aligned} N_k = & \sum_{i=1}^{k-2} \left(\prod_{j<i} (1 - e_j) \right) e_i \prod_{l=1}^{k} n_l \\ & - \sum_{i=1}^{k-2} \left\{ \left(\prod_{j<i} (1 - e_j) \right) e_i \prod_{l=1}^{k-1-i} n_i \right\} \end{aligned} \tag{8.12}$$

Parallel bottom-up evaluation

In the parallel bottom-up evaluation method, all possible join combinations are evaluated in parallel and the results are subsequently combined (Figure 8.6). The scheduler is informed if the result of a join for any combination of two fragments is empty. If such an event occurs, the scheduler removes the tasks containing this fragment combination. The number of eliminated tasks a 3-way and 4-way join operation is thus given by:

[1]Note that at least one task had to be executed to generate this event

$$
\begin{aligned}
N_3 &= (e_1 + (1-e_1)e_2)n_1n_2n_3n_4 - \max(n_1n_2, n_2n_3) \\
N_4 &= e_1n_1n_2n_3n_4 + (1-e_1)e_2n_1n_2n_3n_4 \\
&\quad + (1-e_1)(1-e_2)n_1n_2n_3n_4 - \max(n_1n_2, n_2n_3, n_3n_4)
\end{aligned}
$$

If we generalize this for the k-way equi-join we find the following expression for N_k, the number of eliminated tasks:

$$
N_k = \sum_{i=1}^{k-1} \left(\prod_{j<i} (1-e_j) \right) e_i \prod_{l=1}^{k} n_l - \max(n_1n_2, \cdots, n_{k-2}n_{k-1}) \qquad (8.13)
$$

Because all join combinations are evaluated, more work is done than actually required. However, the idea is that the additional work invested in a single subquery evaluation will result in a higher total elimination factor and, thereby, in a reduction of the total amount of work.

Comparison of the evaluation techniques

Using Equations (8.10) and (8.11) and the expressions for the number of eliminated tasks (8.12) and (8.13) we have calculated the elimination factor for a 4-way equi-join for the Normal, Uniform, and Zipf distribution for both evaluation techniques (See Figures 8.7 and 8.8). These graphs show that for all distributions the parallel bottom-up evaluation results in a larger elimination factor than sequential elimination. The reason for this is that in the parallel bottom up evaluation all the possible join combinations are tried, so that empty join results are detected at an early stage, leading to a larger number of eliminated tasks.

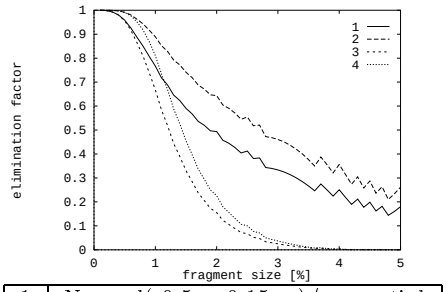
For instance, for a fragment size of 2 % an improvement of 15 % can be observed for a Zipf(1.0) distribution. However, the gain becomes smaller as the fragment size increases.

Calculation of the elimination factor for other multi-join queries show that the range of fragment sizes for which the task elimination is effective does not depend on the number of joins, but only on the distribution parameters. However, within this range, the elimination factor increases with the number of joins.

8.5 Multiple join processing cost

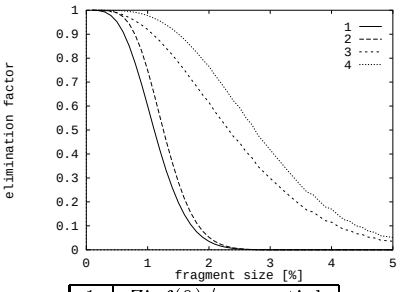
The total elimination factor can now be used to calculate the total processing cost for a multiple join query. In the cost model below the assumption is made that the tasks are evaluated by a single processor. Therefore, it gives an upper bound on the total query cost. When more processors are used by the Query Evaluator, tasks can be evaluated in parallel, which results in a lower response time [2]. The following simple cost model can therefore be used to measure the effectiveness.

[2] Adding processors influences the effectiveness of the dynamic query optimization technique, because it could be that a processor is processing a task that would otherwise be eliminated by a task, which is executed in parallel. However, this effect is negligible, because of the small probability on such an event.



1	Normal($0.5c_A$;$0.15c_A$)/sequential
2	Normal($0.5c_A$;$0.15c_A$)/parallel
3	Normal($0.5c_A$;$0.25c_A$)/sequential
4	Normal($0.5c_A$;$0.25c_A$)/parallel

Figure 8.7: E_4 Normal distribution



1	Zipf(0)/sequential
2	Zipf(0)/parallel
3	Zipf(1)/sequential
4	Zipf(1)/parallel

Figure 8.8: E_4 Zipf distribution

The total query processing cost C_{query} for this architecture is determined by the number of tasks remaining after task elimination $(1-E)N_{task}$ and the task processing cost C_{task}.

Since all tasks are executed on a single processor, each task execution for a k-way join requires at most k fragment transports $C_{com}(p)$ to the processor [3] and a single multi-join execution $C_{join}(p)$. These latter factors depend on the fragment size p.

$$\begin{aligned} C_{query} &= (1-E)N_{task}C_{task} \\ C_{task} &= kC_{com}(p) + C_{join}(p) \end{aligned}$$

Fragment transport requires a constant cost for network access and OS overhead C_{access} and a cost linear in the size of the fragment C_{copy} for copying the data from the network to the processors memory.

For the execution cost of the join operation we only give an upper bound. Each of the $k-1$ equi-join operations results in at most p tuple combinations. Assuming a hash join algorithm implementation we find that the join cost is also linear in the fragment size. In the first phase of the algorithm a hash table is constructed for one of the join operands, and in the second phase this hash table is probed for each join attribute value of the second operand.

$$\begin{aligned} C_{com}(p) &= pN_{bytes}C_{copy} + C_{access} \\ C_{join}(p) &= (k-1)pC_{hash} \end{aligned}$$

In Table 8.1 the parameter setting for our target architecture is given, consisting of MicroVax workstations, using the Amoeba distributed operating system.

Evaluation of the formulas for these two evaluation methods on a 4-way equi-join operation results in the total query processing cost as shown in Figures 8.9

[3] If fragments are properly cached by the processor(s), at most 1 fragment transport is required.

C_{access}	operating system overhead	1 msec
C_{copy}	data transfer rate	1msec/1k
C_{hash}	hash join cost	100μsec
N_{bytes}	tuple size	0.2 k

Table 8.1: The parameter setting for MicroVax systems running Amoeba

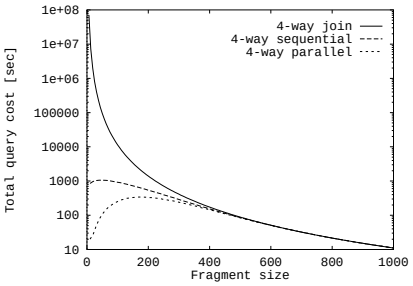


Figure 8.9: C_{query} for a 4-way join and Zipf(1.0) distribution

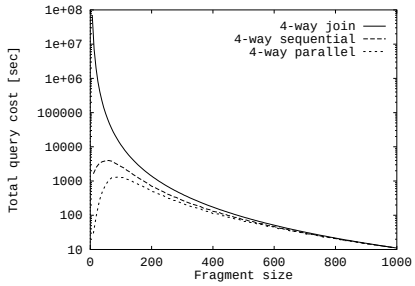


Figure 8.10: C_{query} for a 4-way join and Normal(5,000;1,500) distribution

and 8.10. These graphs present the total query processing cost as a function of the fragment size for the Normal(5,000;1,500) distribution and the Zipf(0.5) distribution. The elimination factors were obtained using the formulas of Section 8.4, and the cardinality of the relations was set to 10,000.

The combination of task elimination and the cost model illustrate the performance gain to be expected from dynamic query processing. The top curve in these graphs represents the total processing cost without task elimination. The result of the calculation shows that within the effective range of the task elimination technique a reduction of the total query cost can be obtained as long as the fragment size is small enough. Outside the effective range the total query processing cost decreases as the fragment size increases. Therefore, for this simple cost model if enough memory is available for query evaluation, it is better to choose a large fragment size outside the effective range of task elimination. However, if parallel query execution is considered, large fragment sizes lead to long communication delays at the query processors, which result in a larger response time. Therefore, more research has to be done to study the effect of parallel execution on the effectiveness of the task elimination optimization technique.

Furthermore, calculations on other multi-join queries show that task elimination becomes more effective as the number of joins increases and the attribute distribution becomes more clustered.

8.6 Conclusion

In this chapter, we presented the opportunities provided by the semantic properties of the relational operations for dynamic query processing. Furthermore, we presented a detailed analysis of a dynamic query optimization technique, called task elimination. A probabilistic model has been used to estimate the potential gains for different data distributions.

This analysis shows that the task elimination technique can lead to significant reductions in the amount of query tasks that have to be processed. The effectiveness of the task elimination technique depends on the attribute distribution and the fragment size. For a real-life distribution, like the Zipf distribution the results are promising. If enough memory is available the fragment size should be chosen as large as possible. However, when only a limited amount of memory is available, task elimination and a small fragment size will reduce the total processing cost. Finally, when using task elimination, it was shown that the parallel bottom up evaluation method increased the effectiveness of the task elimination method.

Future research has to generalize the task elimination technique for other relational operations. Furthermore, the effect of fragment size and fragment caching on the response time in a parallel Query Evaluator must be considered.

Chapter 9

Task allocation

9.1 Introduction

Load balancing is a major issue in the design of a distributed computing system. It deals with the question how a limited number of resources can be used to process a workload, such that a global cost function is minimized. In a parallel data base system the resources involved are the processors, the memory, and the network bandwidth. The load balancing issues address in this case the allocation of sub-queries to processors, the buffer management policy on each processor and global data allocation. The goal of the load balancing mechanism is to minimize the query response time or to maximize the query throughput. In this chapter we consider the first objective: to minimize the response time.

In a parallel data base system, the response time can be reduced by executing sub-queries in parallel. Unfortunately, the response time cannot be infinitely reduced by reducing the task granularity and executing them in parallel, because part of the computation is inherently sequential. For instance, the data re-partitioning and the scheduling overhead are often sequential. Therefore, the amount of *effective parallelism* that leads to a reduced response time is limited.

Furthermore, query execution in a parallel system soon becomes I/O bound as more processors are used to evaluate the query [Mur89] against data that is not locally cached. The throughput is limited further by the rate by which the query result can be delivered to the application.

In the proposed DQP architecture a query is executed by fragmenting the data first and to calculate the query on all these fragment combinations. As the tasks refer to the same global set of fragments, large fragment buffers can considerably reduce the amount of communication required by retaining fragments in local

memory. As the available memory is limited, a buffer manager must decide at run-time, which fragments to keep and remove.

Equally important for the effective use of the data buffers is the allocation of tasks to processors. Ideally tasks that have a number of fragments in common, are executed on the same processor, shortly after one other to maximize the cache hit ratio. In this chapter we present a pilot study of several task allocation algorithms and buffer management strategies.

9.2 The I/O bottleneck

In Chapter 8 the relation between the fragment size and the total number of tasks was investigated taking into account the effect of task elimination. Furthermore, a simple cost model for a k-way join operation was introduced to get an impression of the total query execution time. In this section, this cost model is extended to incorporate the effect of buffer management and task allocation on the average task execution time.

Before discussing specific task allocation and buffer management strategies we introduce the cost factors to be considered for task execution. We define the task evaluation time T_{task} as the time spent by a processor to execute a single task. The task evaluation time can be decomposed into the following cost factors:

$$T_{task} = (k+m)T_{com} + T_{exec},$$

where k is the number of input fragments per task, m the number of result fragments per task, T_{com} the time to retrieve or transfer a fragment and T_{exec} the task execution time.

Assuming that a task can only be executed when all data used is locally available, T_{com} consists of the time to collect the data from memory at the remote processor, transmit the data through the network, and the time to store the data in the local buffer. This cost is generally modeled by a constant delay T_a, which models the network access time and the scheduling overhead and a cost factor linear in the fragment size, which models the cost for copying the data to and from the data buffer.

In the following T_a, ρ_c, and S denote respectively the network access time, the network throughput and the fragment size.

$$T_{com} = T_a + \frac{S}{\rho_c}$$

Note that in a shared-nothing multiprocessor, unless the processors are interconnected through a point-to-point network, data transfers share a single communication medium. Consequently, the data transfers are serialized and limit the amount of effective parallelism. Ultimately, in such a system the network will form the bottleneck of the data base system and determine the minimum response time.

In the expression for the communication time T_a is determined by collisions on the network and process scheduling at both sender and receiver. Therefore

the network access time depends on the network traffic. If, for the sake of argument, we disregard this effect then $T_{com} \approx \frac{S}{\rho_c}$. Assuming that the network is the bottleneck then the upper bound on the maximum task throughput ρ_{task} is given by:

$$\rho_{task} = \frac{(k+m)S}{\rho_c}$$

For a given partitioning the fragment size and the number of tasks to be executed is fixed. The query response time is then determined by the task throughput. Because the factors m and S are fixed, only two approaches can be considered to increase the task throughput of the system:

- Increase the network throughput ρ_c; for instance by using a shared memory architecture or customized communication hardware and a point-to-point network.
- Reduce the number of fragment fetches per task k, through data replication or buffering and using a large local fragment buffer. This way the transmission cost is amortized over a large number of task executions.

The first approach represents a hardware approach, which is both costly and considered to have a limited lifespan. As illustrated by the database machines developed in the FGCS project. After a few years, the performance of parallel data base systems built out of off the shelf components could compete with these dedicated systems. In this chapter, we address the second alternative. By locally storing frequently used fragments, the number of fragment requests can be reduced significantly, so that the amount of parallelism for a specific query can be improved and the response time reduced.

Generally speaking there are two aspects, which determine the effectiveness of a buffer management scheme:

- What is put into the buffer?
- What is removed from the buffer, when the buffer is full.

The former issue is determined in the Goblin architecture by the task allocation algorithm of the Query Scheduler. If the scheduler assigns a task to a query processor, it needs to retrieve the fragments referred to by the task. The latter issue is called the buffer replacement policy. In the following sections we first consider buffer replacement policies and then compare two task allocation algorithms.

9.3 Buffer management

The buffer manager controls a pre-allocated amount of memory on each processor. Furthermore, as Goblin is a main memory architecture a distinction is made between fragment replicas and fragment copies. To ensure persistency the fragment replicas cannot be deallocated, without creating a replica on another

site. Fragment copies are created during query processing and represent either copies of persistent fragments or intermediate results.

Fragments can either have a maximum size, contain a maximum number of tuples, or be variable sized. The first option facilitates memory management both for query processing and fragment I/O. The second option is useful for analysis of the buffer management scheme. The latter option simplifies the implementation and design of the relational operations. In this section we assume that fragments contain a maximum number of tuples and occupy a maximum amount of memory.

The equivalent of a buffer manager in general purpose operating systems is the page manager. The page manager decides on the basis of the reference behavior of the processors workload, which pages can be removed from memory when a running program references a page that is not in memory. From analysis of the reference behavior of programs it turns out that in general programs display a certain locality of page references. This is considered to be a consequence of the imperative programming model. The set of pages at a certain moment in the execution of the program is called the working set and its size differs from program to program. An optimal paging scheme tries to determine the size of the working set and keep the working set in memory to minimize the number of page misses during the program's execution.

In our case the situation is a little different. The scheduler controls the reference pattern of the workload and thereby the amount of locality. An optimal buffering scheme should therefore consider, which fragments (cf. pages) can be replaced from memory and the task allocation algorithm.

In the following section we first determine the best possible buffer management and task allocation technique, and then compare the results with other buffer replacement and task allocation algorithms.

9.3.1 Optimal buffer management

The complete query is expressed as the union of all task results, where each task executes the query for a specific fragment combination. To simplify the analysis we assume that the complete query requires to execute the task for all possible fragment combinations. Each task on k operands can be identified by the combination of k fragment identifiers $(f_1, \ldots, f_k)$ (and its query graph).

Example 9.1 Consider a query which uses relations R, S, and T and each relation consists of 2 fragments. Then the query execution involves evaluation of the tasks identified by (r_1, s_1, t_1), (r_1, s_1, t_2), (r_1, s_2, t_1), (r_1, s_2, t_2), (r_2, s_1, t_1), (r_2, s_1, t_2),(r_2, s_2, t_1), and (r_2, s_2, t_2).

The basic problem addressed in this section can now be stated as follows. Given a collection of tasks $T = \{(f_1, \cdots, f_k) | f_i \subset R_i\}$ and a fragment buffer. Find a task allocation algorithm and a buffer replacement policy that minimizes the number of buffer misses.

The optimal buffer management scheme proposed in this section minimizes the average number of misses required for executing a query consisting of a large number of tasks. The objective of the scheme is to maximize the number of tasks that can be executed for a given buffer size and query.

Theoretically the minimum number of misses can be computed, given the buffer size, the number of relations, and the number of fragment partitions. An important notion in the analysis is the term *buffer volume*, which is defined as follows:

Definition 9.1 The buffer volume V is the number of tasks that can be evaluated with the fragments stored in the buffer.

By definition the buffer volume depends on the number of relations referenced by the query, and the distribution of the buffer slots over these relations.

Example 9.2 If the buffer contains for a 3-way join query 4 fragments of the first relation, 3 fragments of the second relation and 1 fragment of the last relation, in total $4 \times 3 \times 1$ fragment combinations can be formed, resulting in a total number of 12 task evaluations and by definition a buffer volume of $V = 12$.

If in our example the 8 buffer slots are divided differently over the relations, for instance as $3 \times 3 \times 2$, for the same buffer size the buffer volume would be 18. In other words in the first situation 12 tasks can be executed per 8 fragment I/Os, while in the second case 18 tasks, an improvement of 50%.

The basic idea of the optimal buffer management scheme is to divide the available buffer slots equally over the operand relations so that the buffer volume is maximized. The effect of this scheme is summarized in the following theorem:

Theorem 9.1 Assuming that:

- *The set of tasks consists of all fragment combinations.*
- *The task allocation algorithm first allocates the tasks that can be formed with the current buffer content.*
- *The buffer replacement algorithm maintains the maximum cache volume for the current buffer size.*

Consider a task with k- operand relations and relation R_i is partitioned into n_i relations. Then the number of buffer misses M per task for the optimal buffer management scheme and a buffer size C is given by:

$$M = \frac{C}{V} N_{task}$$

where:

$$V = \lceil \frac{C}{k} \rceil^{C \bmod k} \cdot \lfloor \frac{c}{k} \rfloor^{k - c \bmod k}$$

$$N_{task} = \prod_{i=1}^{k} n_i$$

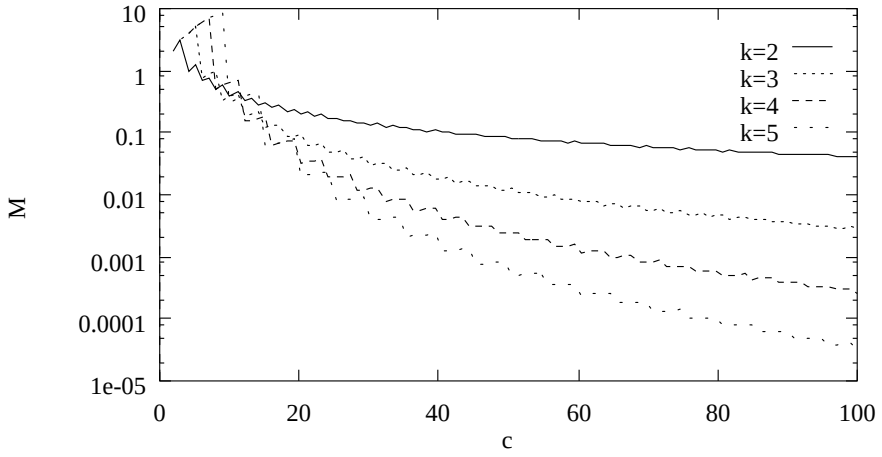


Figure 9.1: The average number of buffer misses per task for optimal buffer management.

Proof 9.1 The first assumption ensures that each fragment combination that can be formed with the fragments in the buffer is an eligible task. The second assumption implies that each task belongs to a distinct buffer volume. Therefore the complete task set can be covered by non-overlapping buffer volumes.

By maximizing the buffer volume the average number of misses per task can be minimized. For each distinct buffer volume C fragment fetch operations are required. The maximum buffer volume can be obtained by dividing the buffer slots over the relations as evenly as possible: $k - C \bmod k$ relations are assigned $\lfloor \frac{C}{k} \rfloor$ slots and the remaining $C \bmod k$ relations are assigned $\lceil \frac{C}{k} \rceil$ slots. For such a buffer slot assignment in total $V = \left(\lfloor \frac{C}{k} \rfloor\right)^{k-C \bmod k} \cdot \left(\lceil \frac{C}{k} \rceil\right)^{C \bmod k}$ tasks can be executed without fetching a new fragment. As a result the average and minimal number of misses per task is given by:

$$M = \frac{C}{\lfloor \frac{C}{k} \rfloor^{k-C \bmod k} \cdot \lceil \frac{C}{k} \rceil^{C \bmod k}}$$

Figure 9.1 displays the average number of misses per task for queries with 2 to 5 operands. Note that there are two extreme cases, that are not properly covered by the previous cost formula:

- The buffer can hold only the fragments for a single task.
- The buffer can hold all the fragments for the complete task set

In the first case, the buffer is initially filled with the appropriate number of fragments. For each following task it is sufficient to replace a single fragment. Thus the average number of buffer misses per task is given by:

$$M = \frac{k + N_{task} - 1}{N_{task}}$$

In the second case, each fragment needs to be loaded in the buffer at most once. Thus the average number of buffer misses is given by:

$$M = \frac{\sum_{i=0}^{k} n_i}{N_{task}}$$

To mimic the optimal buffer management scheme presented in this section a buffer replacement strategy and task allocation algorithm have been designed, called the Maximum Cache Volume (MCV) and Maximum Cache Hit (MCH) allocation algorithm. In the following sections first the buffer replacement and task allocation algorithms are presented and finally the results from a simulation study are presented.

9.4 Buffer replacement techniques

Before a task can be completed, it is required that all its fragments are locally available. Each time a fragment is not available, it is requested from the buffer manager on a remote site and stored in a slot allocated in the local buffer. If all the slots in the buffer are already occupied, the buffer replacement must decide which fragment to be removed. To avoid loss of valuable data some fragments are *fixed*, either permanently for persistent fragments, or temporarily for fragments that are required for the current task evaluation.

All the buffer replacement algorithms considered have in common that they do not remove fragments fixed in memory. Instead, they select a victim among the set of non-fixed fragments.

9.4.1 Random replacement

The random replacement strategy randomly selects a fragment from the set of non-fixed fragments. A disadvantage of this approach is that fragments can be removed that are required for the next task evaluation. Furthermore, the random behavior makes it impossible for the scheduler to predict which fragments are stored in the buffer at a certain moment. Consequently, the task allocation algorithm cannot take the buffer content into account to prevent fragment I/O.

9.4.2 LRU replacement

The LRU buffer replacement policy is well known from OS page replacement algorithms. The idea is based on the locality principle of application programs. A program references only a limited set of pages during a time interval. This set is called the working set. The pages from the working set will only change gradually during execution. If the buffer is large enough to contain the working set, the LRU algorithm performs well. However, if the working set is slightly larger than the buffer, there is a situation where the algorithm performs badly,

namely when the program sequentially references all the pages in the working set, the LRU algorithm removes the page it needs in the near future.

In the database application the reference pattern is determined by the task allocation algorithm. For the LRU algorithm to perform well it is necessary that the locality principle holds for the reference pattern. It is therefore to be expected that the LRU algorithm performs best with a task allocation algorithm that displays locality.

9.4.3 Maximum Cache Volume replacement

The MCV replacement algorithm is derived from the optimal buffering scheme. It tries to keep an equal number of fragments of the relations in the buffer. The idea is that by keeping an equal number of fragments for each relation in the buffer, the amount of tasks that can be formed is maximized. Thus increasing the probability of hits for the next task execution. For instance, given a buffer size of 12 and a query on three relations, a maximum of $4 \times 4 \times 4 = 64$ tasks can be evaluated, using the MCV policy, while the minimum amount of tasks is obtained by allocating the buffer slots in an unbalanced fashion, like $10 \times 1 \times 1 = 10$ tasks.

This method is designed to be combined with the the maximum cache hit task allocation algorithm, which keeps track of the buffer contents.

9.5 Task allocation

When a task arrives at a Query Processor the task is queued for execution. First the task operands are retrieved from the local buffer manager. When all operands are available the task waits in the ready queue for execution. Therefore, a certain amount of time the task has to wait, either for requested fragments to arrive, or until the query processor is free to process the task[1].

A load balancing algorithm tries to minimize the average waiting time for a task. In general load balancing algorithms use task allocation and task migration to obtain a good load distribution. Furthermore, as the waiting time also depends on the amount of communication required, the second objective of the load balancing algorithm is a reduction of communication overhead.

As the tasks are relatively small we do not consider task migration algorithms, but consider only task allocation algorithms. The following three task allocation algorithms are considered:

- Random task allocation.
- Sequential task allocation.
- Maximum Cache Hit allocation.

The following sections present these algorithms in more detail. To reduce the waiting time they share that they assign tasks to query processors with the lowest load first. The feedback mechanism continuously reports the load of a processors to the scheduler.

[1]The task execution algorithm described in Chapter 10 overlaps task execution with fragment I/O.

9.5.1 Random allocation

In random task allocation, the query scheduler selects a task randomly from the task table and assigns it to the processor with the minimum load. The advantage of this algorithm is that the query scheduler only requires information on the current load of the query processor.

Because the algorithm does not take into account the buffer contents of the target query processor, the average number of misses per task is relatively high for all the buffer replacement strategies.

9.5.2 Sequential allocation

In sequential task allocation the tasks are allocated to a processor in an incremental fashion. This means that two successively allocated tasks differ in only a single fragment identifier. This allocation mechanism results in a certain locality of reference at the query processor and is therefore expected to produce less misses than random task allocation.

It requires the query scheduler to order the tasks before they are assigned. If the complete product space of fragment identifiers must be traversed this is a relatively simple problem that can be solved by numbering the fragments of each relation. If the first operand specifies the least significant digit and the last operand the highest significant digit, then each task specification is uniquely identified by a number.

Example 9.3 For a query on relations R, S and T, which are partitioned in respectively $20, 30$ *and* 10 *fragments, the task* (r_{14}, s_4, t_9) *represents the number* $(14 * 30 + 4) * 10 + 8 = 4248$*, the next task will be either* 4249 *or* 4247*, which corresponds to tasks* (r_{14}, s_4, t_9) *and* (r_{14}, s_4, t_7)*, respectively.*

9.5.3 Maximum Cache Hit allocation

Under the Maximum Cache Hit task allocation algorithm, the scheduler keeps track of buffer contents of the local Buffer Managers. If a QP is available for handling a new task, the least expensive one (w.r.t. I/O) is selected from the task table.

Thus the query scheduler will first assign tasks that can be formed with the current buffer content. If these tasks are not available, it assigns a task that requires only a single fragment retrieval.

Example 9.4 Consider the same query as in the previous example. In the MCH allocation scheme the QS knows the buffer contents for each QP. Let the buffer of a QP contain the following fragments: $\{r_1, r_2, s_1, s_2, t_1\}$*. With this buffer contents the following tasks can be formed:*

Zero fragment requests
(r_1, s_1, t_1)
(r_1, s_2, t_1)
(r_2, s_1, t_1)
(r_2, s_2, t_1)

When all these task have been executed, the scheduler selects a task that requires a single fragment transport:

Single fragment request	
(r_1, s_1, t_x)	$x \neq 1$
(r_x, s_1, t_1)	$x \neq 1 \wedge x \neq 2$
(r_1, s_x, t_1)	$x \neq 1 \wedge x \neq 2$

By this task assignment the buffer content changes so that new tasks can be formed that require zero fragment requests.

9.6 Performance comparison

We have designed a simulation experiment to compare the previously discussed task allocation and buffer replacement techniques. In this experiment the average number of buffer misses per task are measured for each buffer replacement - task allocation method combination (r, t), where $r \in \{MCV, LRU, RND\}$ and $t \in \{MCH, SEQ, RND\}$.

The measured misses per task indicates the average number of fragment fetches per task. In the experiment we assume that the buffer is initially empty. Furthermore we assume, similar to the optimal buffer management algorithm, that the complete task set is defined as the Cartesian product of all fragment combinations.

The experiment is run on tasks consisting of k operand relations, where each relation is partitioned into n fragments. The simulation executes the resulting n^k tasks by selecting a task from the task set using the task allocation method and simulating the task execution on the buffer. For each simulated task execution the buffer replacement algorithm determines the number of misses, and for each fragment to be retrieved the fragments to be removed from the buffer. Finally, the ratio of the total number of misses and tasks is returned.

In runtime overhead these methods can be ordered increasingly for the task allocation algorithms as: $RND \leq SEQ \leq MCH$, and for the buffer replacement techniques as $RND \leq LRU \leq MCV$. Thus the combination (RND, RND) results in the lowest run-time overhead and the combination (MCH, MCV) in the highest. Obviously, this overhead must be set against the performance gain resulting from a more efficient use of the buffer.

9.6.1 Cache miss per task ratio

In the experiments we have examined all the possible combinations of buffer replacement and task allocation algorithms. For each combination we varied the number of operands for each task k, the number of fragments for each operand n and the buffer size c represented by the maximum number of fragments it can contain.

To visualize the results, the miss ratio is expressed as a function of the *relative buffer size* ρ_c. The latter is a derived factor which expresses the buffer size for a measurement as a percentage of the total number of fragments required. Thus: $\rho_c = c/nk$. For a task on k operands consisting of n fragments the total number of fragments is nk. If the relative buffer size is 100% each fragment is retrieved once, independent of the number of tasks. Thus the miss ratio at

a 100% relative buffer size will be the same for each task allocation - buffer replacement combination.

The results from the experiment on tasks with three operand relations, each partitioned into 25 fragments are presented in Figures 9.2 - 9.4.

In Figure 9.2 the allocation algorithms are compared in combination with the MCV replacement strategy. As expected result the MCH and SEQ allocation algorithms in less buffer misses than the RND algorithm. Furthermore, we see that under MCV replacement the SEQ allocation is a linear function of the relative buffer size, because the probability that a fragment is stored in the buffer increases linear with the relative buffer size. Finally, we observe that the MCH algorithm exploits large buffer sizes best.

Figure 9.3 confirms the last observation. However, it does not confirm our expectation that the (MCH,MCV) is the best possible combination.It turns out that the (MCH,LRU) combination performs better, probably because LRU replacement removes the fragment that is least recently used in a task execution. The knee shaped curve for the combination of the sequential allocation algorithm is caused by the enumeration property of the algorithm. Each time the buffer size is a multiple of the number of fragments per relation, the LRU replacement algorithm ensures that the most recently used fragments are kept in memory. Because the task allocation enumerates the tasks using the fragment identifiers for the operands as base, the least significant digits (or the fragment identifiers for the first operands) are used most frequently and lead therefore to a buffer holding all the fragments associated with the first few operands.

With a random replacement strategy (Figure 9.4) all the allocation algorithms perform equally bad. This observation underlines the importance of a proper buffer replacement strategy.

Figure 9.5 shows that the influence of the replacement strategy for the MCH allocation algorithm on the miss ratio is marginal. Both LRU and MCV replacement strategies show a good performance.

9.6.2 Task allocation and buffer replacement overhead

Because the QS is responsible for the task allocation is the task allocation overhead the most important factor for the system performance. Therefore, we have measured its CPU cost. For the experiment the average number of cycles spent in each of the task allocation and buffer replacement algorithms was determined. For this purpose code was added to do a basic block count. These basic block counts were combined by a profiler to determine the total number of cycles spent in each of these algorithms. The results of these experiments are summarized in Table 9.1. This is a small but representative sample of all experiments.

The measurements show that the overhead of the MCH algorithm is directly related to the buffer volume. In the current implementation of this algorithm the tasks that can be formed for a certain buffer content are constructed at each iteration of the algorithm. The overhead can be reduced by generating these tasks incrementally. Nevertheless, the cost will be linearly related to the buffer volume $V = \frac{c}{n}^k$.

The overhead for the sequential and random allocation algorithms decreases as the buffer size increases. Furthermore, it is an order of magnitude less than

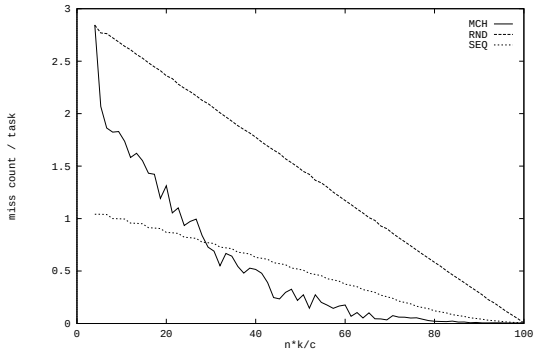


Figure 9.2: MCV buffer replacement: k=3, n=25

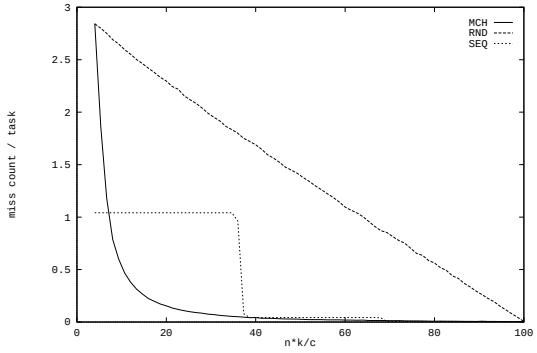


Figure 9.3: LRU buffer replacement: k=3, n=25

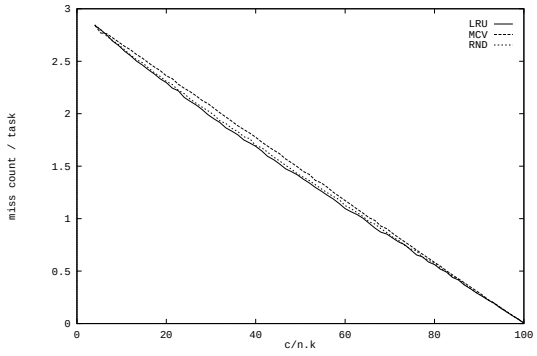


Figure 9.4: Random buffer replacement: k=3, n=25

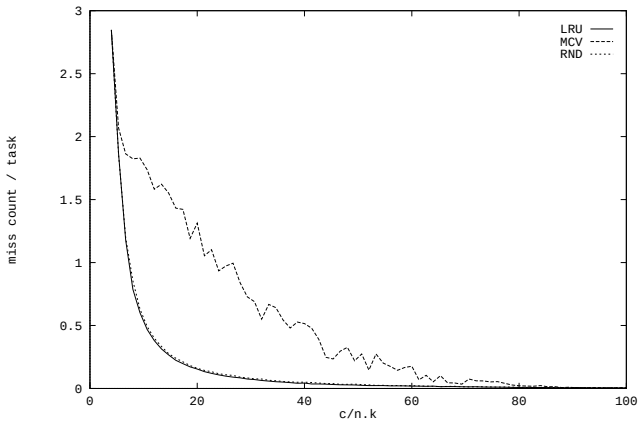


Figure 9.5: MCH task allocation: k=3, n=25

LRU buffer replacement, k=3, n=10		
algorithm	buffer size	cycles
MCH	10	2418
MCH	20	8135
MCH	30	21325
SEQ	10	438
SEQ	20	96
SEQ	30	27
RND	10	774
RND	20	695
RND	30	27

Table 9.1: The task allocation overhead

the overhead for MCH.

Consequently, if we consider both the effectiveness of the task allocation algorithms and its overhead the results are obtained by using the SEQ allocation algorithm in combination with the LRU or MCV buffer replacement algorithm. The choice between LRU and MCV depends on the buffer size in relation to the partitioning degree.

9.7 Conclusion

In this chapter a few important issues related to load balancing were discussed. In particular the importance of data distribution and data declustering were left out. Instead the discussion concentrated on buffer replacement and task allocation algorithm.

We argued that an important difference with buffer management for general-purpose operating systems is that in data base systems it is possible to introduce data locality in the query evaluation process. The optimal buffer management scheme presented exploits this feature.

Finally, for a selection of buffer replacement and task allocation algorithms the effect on the average buffer misses per task was measured through simulation. The elaborate MCH - and simple SEQ task allocation algorithm in combination with the LRU buffer replacement algorithm turned out to result in the lowest average miss average.

Due to the overhead of the MCH algorithm and its influence on the total system performance, the SEQ task allocation algorithm is by far the preferred task allocation algorithm.

Chapter 10

Task evaluation

10.1 Introduction

In this chapter we present the dynamic query optimization algorithm used by the Query Processor to evaluate tasks. The prime characteristic of the task evaluation technique is that each task is executed on a single processor using a main-memory buffer to store intermediate results and relation fragments. The task allocation algorithm ensures that tasks consecutively allocated to a processor differ in only a few fragments enabling the effective use of the main-memory buffer. Therefore the main cost factor to be taken into account for task evaluation is CPU cost.

The graph representation of the query (See Chapter 6) forms the basis of the algorithm. The task evaluation process proceeds by iteratively reducing the query graph until a single node remains. At each iteration an edge, called the *target* edge, is selected and removed from the graph. At the same time the relations associated with the edges are changed accordingly, such that the constraints represented by the target edge propagate to the neighboring edges and their associated relations in the remaining graph.

In the following section we first describe an old dynamic query evaluation technique based on graph reduction. The Goblin algorithm can be seen as a refinement of this algorithm. We present it in global terms to put the Goblin dynamic query evaluation algorithm in a context. The bulk of this chapter discusses the Goblin algorithm in detail.

10.2 The Wong-Youssefi algorithm

The Wong-Youssefi algorithm used to process QUEL queries in INGRESS is based on graph reduction. The following gives a flavor of the algorithm. The details can be found in [WY76]. The algorithm solves project-select-join queries of the following form:

$$\pi_{\alpha}\sigma_{F_1\wedge\cdots\wedge F_n}(R_1 \times \cdots \times R_k)$$

The query is transformed into a hyper-graph representation of nodes and hyper-edges, i.e. sets of nodes. The nodes in the graph represent the relation attributes involved. The edges are used for two purposes: *relation edges* group the attributes of a single relation and *condition edges* to group the attributes of the selection conditions F_i. The relation associated with a *relation edge* E is denoted by $R(E)$ and the selection condition associated with a *condition edge* is denoted by $C(E)$.

A well-known optimization heuristic, - used in this algorithm -, is to perform projections as soon as possible in the evaluation process. The notion of *distinguished nodes* is used to implement this heuristic. Informally at each point in the execution this set contains the attributes minimally required to construct the query result. The initial set contains the nodes of the projection attributes α. During query evaluation join-attributes are added to the set of distinguished nodes when required for sub-query evaluation.

To describe the QUEL query optimization algorithm we define the procedure EVAL. This procedure takes a hyper-graph $\mathcal{G}$ and a set of distinguished nodes $\mathcal{D}$ and recursively compiles a query program to calculate the relation defined by $(\mathcal{G}, \mathcal{D})$. By definition the program delivers the result relation in RESULT($\mathcal{G}$). The procedure EMIT constructs the query program by adding statements to it.

Depending on the situation the following three actions are performed to produce a query program (See also [Ull89][pages 676-692]):

1. If the graph $\mathcal{G}$ consists of k disjoint hyper graphs $\mathcal{H}_1, \ldots, \mathcal{H}_k$, the procedure EVAL is called recursively on each of the hyper-graphs $\mathcal{H}_i$, where the set of distinguished nodes is restricted to the nodes of each sub-graph NODES($\mathcal{H}_i$).

   Because the disjoint hyper-graphs represent independent relations, the query result is defined by the Cartesian product of the result obtained from the sub-graphs $\mathcal{H}_1, \ldots, \mathcal{H}_k$.

   ```
   FOR i FROM 1 TO k DO
       E_i := D ∩ NODES(H_i)
       EVAL(H_i, E_i)
   DONE
   EMIT RESULT(G) := RESULT(H_1) × ··· × RESULT(H_k)
   ```

2. If removal of a relation hyper-edge E decomposes the hyper graph into $k \geq 1$ disjoint sub-graphs, EVAL is called recursively on the sub-graphs and the

query result is found by joining the result of each call and projecting onto the distinguished attributes. Each relation hyper edge F, which intersects E, is semi-joined with $R(E)$ as an optimization step.

FOREACH F IN $\mathcal{G}$ DO
  EMIT $R(F) \ltimes R(E)$
DONE
FOR i FROM 1 TO k DO
  $\mathcal{E}_i$:= $(E \cup \mathcal{D}) \cap$ NODES$(\mathcal{H}_i)$
  EVAL$(\mathcal{H}_i, \mathcal{E}_i)$
DONE
EMIT RESULT(G) := $\pi_{\mathcal{D}}(R(E) \bowtie$ RESULT$(\mathcal{H}_1) \bowtie \cdots \bowtie$ RESULT$(\mathcal{H}_k))$

3. If a condition hyper-edge E is removed, the query result is found by calling EVAL on the resulting sub-hypergraph and evaluating the selection condition $C(E)$ on the result. Let $\mathcal{H}$ denote this subgraph, then the query is evaluated as follows:

$\mathcal{E}$:= $(E \cup \mathcal{D}) \cap$ NODES$(\mathcal{H})$
EVAL$(\mathcal{H}, \mathcal{E})$
EMIT RESULT$(\mathcal{G})$:= $\pi_{\mathcal{D}}\sigma_{C(E)}$(RESULT$(\mathcal{H})$)

At each graph reduction step, if the graph does not consist of the union disjoint graphs, a *target edge* is selected and one of the above actions is performed when appropriate. The authors consider the selection of the target edge a crucial issue in the optimization strategy. Their solution is based on the following heuristics rules that are applied in decreasing order of priority:

- Relation edges that are *small* (in cardinality) and intersect only one or more relation hyper-edges receive the highest priority for removal. These relations are semi-joined with their intersecting relations and potentially reduce their size.
- Relation edges that represent cut edges of the graph are preferred. This heuristic favors programs that use efficient decomposition joins [Ull89][page 676] over programs that use sequences of two-way joins.
- Remaining relation edges that intersect only relation edges.
- Condition edges receive the lowest priority, because their removal can result in a set of disjoint graphs, leading to the calculation of a Cartesian product of the result of each sub-graph.

The use of heuristics is an important weakness of this algorithm. The assumptions on which these rules are based do not always hold and can therefore result

in a sub-optimal execution. The Goblin task execution algorithm is based on the graph reduction paradigm found in the Wong-Youssefi algorithm, but it is refined to take the actual cost of the operations and the Goblin architectural features into account.

In the following section we first point out the major differences between the two algorithms. Second, we present the Goblin task execution algorithm and illustrate the algorithm using the Mail example. Finally, we discuss the criteria that control the evaluation order.

10.3 Goblin task evaluation issues

This section discusses the main differences between the Wong-Youssefi algorithm and the Goblin task evaluation algorithm. They are related to the optimization strategy, the data model, and the reuse of intermediate results.

10.3.1 Cost based versus heuristic

The QUEL query optimization algorithm is driven by simple heuristics. This means that the the join order in expressions like $R(E) \bowtie \text{RESULT}(\mathcal{H}_1) \cdots \bowtie \text{RESULT}(\mathcal{H}_k)$ is selected randomly. This leads to possibly sub-optimal query evaluation plan.

In contrast, the Goblin task evaluation algorithm is based on up-to-date cost information. For each iteration step of the algorithm the best possible choice is made on the basis of cost estimates and data availability. The latter is a result from skewed data arrival in a distributed query processing architecture.

Furthermore, to overlap fragment I/O with query processing, a task is taken into execution even though not all the fragments are already available.

10.3.2 Query result representation

The Wong-Youssefi algorithm is designed to execute queries on n-ary relations and produce the query result as an n-ary relation. In Goblin the query result is represented by a set of binary relations, - pivot relations -, which are related by a single pivot attribute. In other words, the decomposed storage model is also used to represent the query result.

This choice eliminates projection cost from the task evaluation. If an application accesses only a sub-set of the projection attributes, it merely has to retrieve the associated pivot relations. Using these relations it can then easily reconstruct the data into the required format. This approach largely off-loads the reconstruction of a query result to the application site. It is based on the assumption that result reconstruction is cheap in main-memory.

Furthermore, the relational operations use and produce binary relations only. This allows for a more efficient implementation of these restricted operations than can be obtained for more general operations handling n-ary relations.

10.3.3 Reuse of intermediate results

Finally, the QUEL algorithm is designed to execute a single query, while in Goblin a large number of similar queries are executed concurrently. Therefore,

the former does not take the overlap that exists between query tasks into account. Yet, most tasks have some fragment combination in common, and, therefore, can use the same intermediate results.

The Goblin task evaluation maintains intermediate results in a buffer for reuse. During task evaluation the buffer content is first checked to see if the result of a reduction step is already available. For instance, the buffer is searched for the task fragments in the initialization phase.

To summarize, the Goblin task evaluation algorithm is a cost-driven dynamic query optimization scheme based on graph reduction which allows the reuse of intermediate results. The algorithm consists of an initialization phase, a graph reduction phase, and a pivot phase. In the following sections an overview is presented of the evaluation scheme and the different phases are discussed in detail.

10.4 Notation and terminology

Similar to the task generation process (Chapter 7), task execution is driven by a graph representation of the query. For task execution the query graph is extended with a cost function, which determines for each edge the cost for removing it from the graph. The query graph is thus represented by the five-tuple $G = \langle N, E, \mathcal{A}, card, cost \rangle$, with N the set of nodes to represent attributes and constants, E a set of (undirected) edges E, and A the set of projection attributes, or *attribute nodes*. The other nodes are *internal nodes* or *constant nodes* if they refer to constants. The functions $card : E \rightarrow I\!N$ and $cost : E \rightarrow I\!N$ associate a cardinality and *reduction cost* with the edges. Furthermore, each edge is either a relation edge or a condition edge. The relation and condition associated with an edge (x, y) is denoted $R(x, y)$ and $C(x, y)$, respectively. The number of edges connected to a node, i.e. its *degree*, is denoted as $d(x)$. To simplify the following expose we assume that the query graph consists of a single connected component.

A task is specified by its query graph and edge assignment. The edge assignment associates edges with fragments of the corresponding relations. If $\mathcal{F}$ denotes the set of fragments, the task is completely defined by the pair $T = \langle G, a \rangle$, where $a : E \rightarrow \mathcal{F}$ associates a fragment with each edge. Note that each sub-graph of G with its associated edge assignment specifies a sub-query.

The task result is represented by binary relations, called *pivot relations*, one for each attribute in $\mathcal{A}$. These pivot relations identify the possible values for the projection attributes and relate them through a *pivot attribute*[1]. Thus related objects (i.e. satisfy the constraints specified by the query graph) have the same pivot attribute.

If a node is associated with a pivot relation it is a *pivot node*, otherwise it is a *single node*. Two pivot nodes are *related* if their pivot relations use the same pivot attribute. The set of related pivot nodes is called a *pivot graph*.

Example 10.1 We use the Mail query graph to illustrate these concepts (See Figure 10.1). The graph contains two pivot graphs. In the pivot graph PG_q *con-*

[1] In the object oriented terminology it is called an object identifier

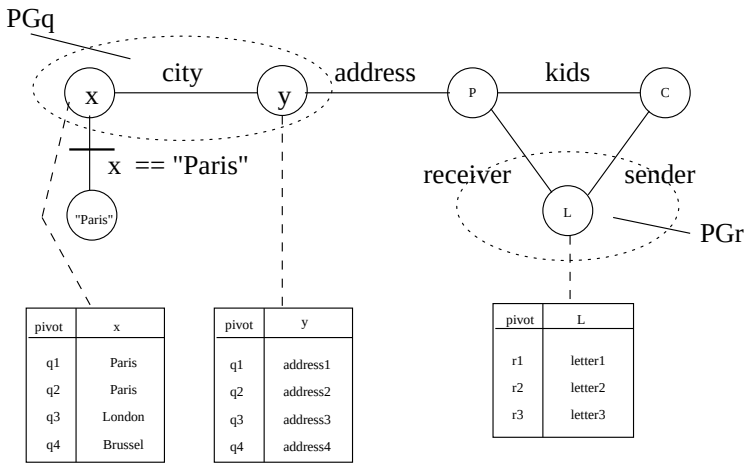


Figure 10.1: Pivot relations and pivot graphs

taining the pivot nodes x and y the pivot relations are related through the pivot attribute q, and in PG_r by attribute r.

We can now describe the Goblin evaluation algorithm in detail, starting with graph initialization followed by the graph reduction algorithm.

10.5 Graph initialization

The graph initialization encompasses binding of the graph with the actual task parameters and the initialization of the cost factors for the graph edges.

In Goblin the binary relation fragments are distributed over the query processor pool. During the initialization phase the query processor initializes the cardinality and cost function for the specific task by identifying the fragments (and intermediate results) locally available.

The remaining fragments are retrieved from remote query processors. The corresponding operators are inhibited until their operands arrive to enable task execution to overlap with fragment I/O.

The relation edges are labeled by the cardinality of the corresponding fragments. Furthermore, the CPU cost to remove each edge from the graph is estimated. This cost estimate directly reflects the reduction actions involved. Its discussion is therefore delayed to Section 10.8.

The binding algorithm discussed in Chapter 7 ensures that each projection attribute node is connected to at least one relation edge. This way the query graph contains all information necessary to solve the query.

10.6 Graph reduction

The graph reduction algorithm selects and removes a *target edge* from the query graph at each iteration. In the process it generates pivot relations for the nodes

connected to the target edge, the *target nodes*. The generated pivot nodes are related and belong therefore to the same pivot graph.

Intuitively, the pivot relations represent the solution to the sub-query identified by the pivot graph. The constraint represented by the target edge is materialized in the pivot relations. When two pivot graphs are combined, the constraint implied by the target edge connecting them is propagated to the pivot relations contained in the sub-graphs. The pivot relations are frequently renumbered to ensure that after merging two pivot graphs they have the same unique pivot attribute. This is done using the *mark* operation μ which extends its operand relation with a unique new pivot attribute.

The *target node type*, *target edge type* and the *presence of pivot relations* determine the operations to be performed when the target edge is removed.

The target node can either be an *attribute* node, an *internal* node, or a *constant*. Their pivot relations form the task result. The query graph is completely reduced when it contains only the attribute nodes. The internal nodes are part of the query graph but do not occur in the reduced query graph. The pivot relations created for these nodes are therefore dropped once their constraint is propagated to the pivot relations of all their neighbors. Constant nodes specify selection conditions. Once the condition is evaluated they are removed.

The target edge type specifies a constraint i.e. a condition- or a relation edge. Removal of a condition edge implies either a theta-join operation, if the edge connects two attribute nodes, or a selection, if the edge connects an attribute node to a constant node. Removal of a relation edge results in the evaluation of an equi-join operation.

In the following subsections we use the target edge type to classify the different actions performed by the algorithm. First we consider condition edges connected to a constant. Second, condition edges between attribute or internal nodes are considered. They lead to theta-join operations. Finally, we consider relation edges.

10.6.1 Selection

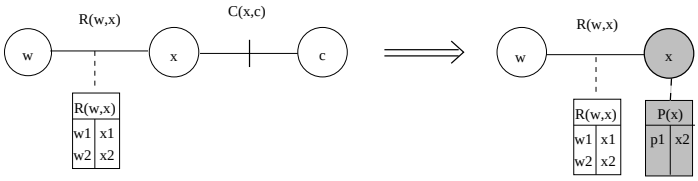
Consider the target edge between an attribute node and a constant node denoted by x and c, respectively. The condition for the target edge (x, c) is represented by $C(x, c)$.

The target edge puts a constraint on the domain of x. If x is a pivot node, i.e. a partial result, this domain is always made explicit in a pivot relation. If x is still a single node, the domain is determined by taking the intersection over the domains of the incident relation edges. Fortunately, it is not necessary to calculate this intersection. It suffices to select one relation edge, apply the selection condition, and use the result to construct the pivot relation for x. In the following we first consider the case that x is a single node, then we present the actions involved if x is a pivot node.

- If x is a single node, then there is at least one incident relation edge. This is guaranteed by the binding phase in the task generation algorithm. Let $R(w, x)$ represent this relation, then we can determine the pivot relation $P(x)$ as follows:

$$P(x) \quad := \quad \pi_{[p,x]} \mu \sigma_{C(x,c)} R(w,x)$$

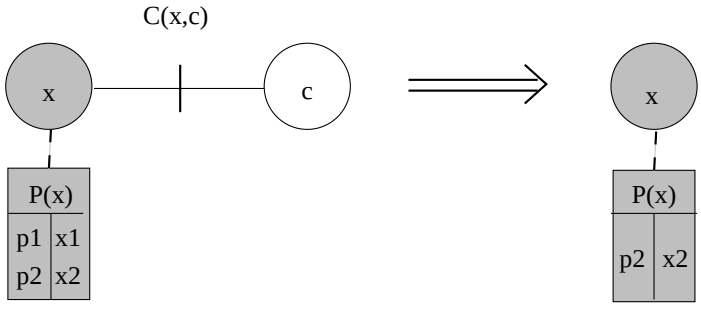
The graph rewrite is shown below. The constant node and target edge are removed from the query graph. The created pivot node is shaded and forms a pivot graph containing one node. Furthermore, the attribute values in the pivot relation $P(x)$ is a sub-set of the attribute values stored in the relation $R(w,x)$ that satisfy the selection condition.



- If x is a pivot node it belongs to a pivot-graph. The selection condition can immediately be applied to the pivot relation $P(x)$, which can then be used to restrict the relations in the pivot graph. To achieve this the relations are semi-joined on their pivot attribute with $P(x)$. If $x_1, \ldots, x_k$ are the other nodes in the pivot-graph then the following actions are performed:

$$\begin{aligned} P(x) &:= \sigma_{C(x,c)} P(x) \\ P(x_1) &:= P(x_1) \ltimes P(x) \\ &\vdots \\ P(x_k) &:= P(x_k) \ltimes P(x) \end{aligned}$$

The constant node and target node are removed from the query graph. The new pivot relation $P(x)$ is a sub-set of the original pivot relation. This is illustrated in the following figure:



10.6.2 Theta-join

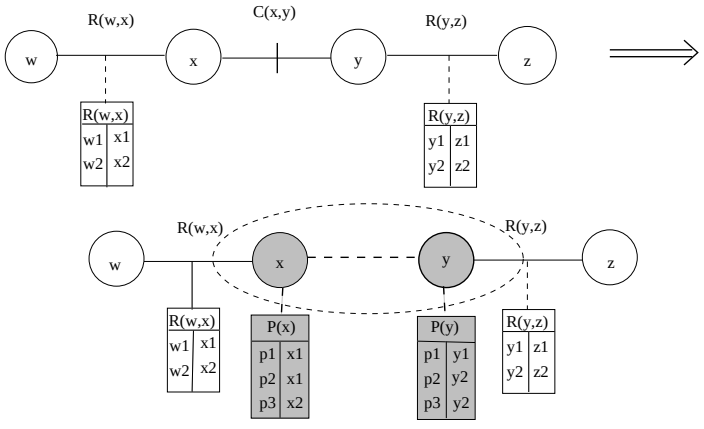
Consider that the target edge connects two attribute- or internal nodes by a condition edge denoted by x, y and $C(x, y)$, respectively. The condition requires a theta-join over the attribute domains.

Each node can again be a single node or a pivot node. For a single node the attribute domain is defined by an incident relation edge. The domain of the pivot node is defined by its associated pivot relation. In the algorithm three different cases must be considered: two single nodes, one single node and a pivot node, and two pivot nodes. These cases are discussed separately.

- If x and y are single nodes then their attribute domains are determined by the incident relation edges. The edge with the smallest relation is chosen to determine the attribute domain of the pivot relations. Let $R(w, x)$ and $R(y, z)$ represent the relations then we can determine the pivot relations $P(x)$ and $P(y)$ by joining the relations $R(w, x)$ and $R(y, z)$ on the condition $C(x, y)$. The join result is extended with a pivot attribute p and used to construct the pivot relations for x and y. It is possible to determine the pivot relations for node w and z at the same time. However, if these pivot relations already exist, they must be combined with the new pivot relations. For simplicity, we will just calculate $P(x)$ and $P(y)$.

$$\begin{aligned} T &= \mu(R(w,x) \bowtie_{C(x,y)} R(y,z)) \\ P(x) &= \pi_{[p,x]} T \\ P(y) &= \pi_{[p,y]} T \end{aligned}$$

The modification of the query graph is indicated in the following figure. The nodes x and y form a pivot graph. In the example it is assumed that the join condition is satisfied for the combinations $C(x1, y1)$, $C(x1, y2)$ and $C(x2, y2)$.

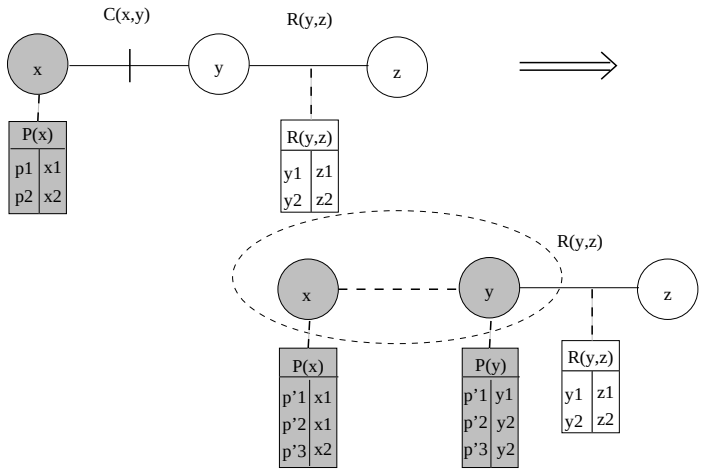


- If x is a pivot node and y a single node, then the pivot relation $P(x)$ and a relation edge connected to node y are used to determine the possible attribute value combinations. Let $R(y,z)$ denote the relation then the pivot relation $P(y)$ is found by joining $P(y)$ and $R(y,z)$ on the join condition.

  The join result is renumbered so that each solution is uniquely identified by a new pivot attribute p'. The pivot relations of the other nodes in the sub-graph of x must also be renumbered to reflect this change. Let $P(x_1), \ldots, P(x_k)$ denote these pivot relations then the following operations are performed:

$$
\begin{aligned}
T &:= \mu(P(x) \bowtie_{C(x,y)} R(y,z)) \\
P(x) &:= \pi_{[p',x]} T \\
P(y) &:= \pi_{[p',y]} T \\
P(x_1) &= \pi_{[p',x_1]} T \bowtie_{T.p=P(x_1).p} P(x_1) \\
&\vdots \\
P(x_k) &= \pi_{[p',x_k]} T \bowtie_{T.p=P(x_k).p} P(x_k)
\end{aligned}
$$

  This operation is illustrated in the following figure. Similar to the previous example we assume that the join condition is satisfied by the attribute combinations: $C(x1, y1)$, $C(x1, y2)$, and $C(x2, y2)$.



- If both x and y are pivot nodes, then their attribute domains are defined by pivot relations. The new pivot relations are determined by joining these two domains on the join condition, renumbering the result and projecting them on the new pivot attribute, x and y attribute.

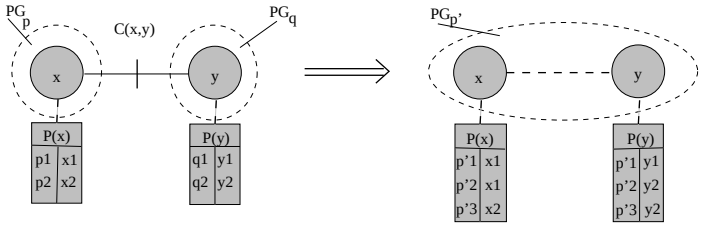
  The relations in each pivot-graph must be renumbered. Let the pivot relations of the pivot-graph containing x be represented by $P(x_1), \ldots, P(x_m)$

and, similarly, the pivot relations associated with node y by $P(y_1), \ldots, P(y_n)$. The pivot attribute for $P(x)$ and $P(y)$ is denoted by p and q, respectively. The new pivot attribute is represented by p'.

$$
\begin{aligned}
T &:= \mu(P(x) \bowtie_{C(x,y)} P(y)) \\
P(y) &:= \pi_{[p',y]} T \bowtie_{T.q=q} P(y) \\
P(y_1) &:= \pi_{[p',y_1]} T \bowtie_{T.q=q} P(y_1) \\
&\vdots \\
P(y_n) &:= \pi_{[p',y_n]} T \bowtie_{T.q=q} P(y_n) \\
P(x) &:= \pi_{[p',x]} T \bowtie_{T.p=p} P(x) \\
P(x_1) &:= \pi_{[p',x_1]} T \bowtie_{T.p=p} P(x_1) \\
&\vdots \\
P(x_m) &:= \pi_{[p',x_m]} T \bowtie_{T.p=p} P(x_m)
\end{aligned}
$$

If node x or y is an internal node and is not connected to an edge it will not be used in a future reduction step. Therefore, its pivot relation does not have to be constructed and it can be removed from the sub-graph.

The reduction step merges the two sub-graphs in a single graph. This is shown in the following figure.



10.6.3 Equi-join

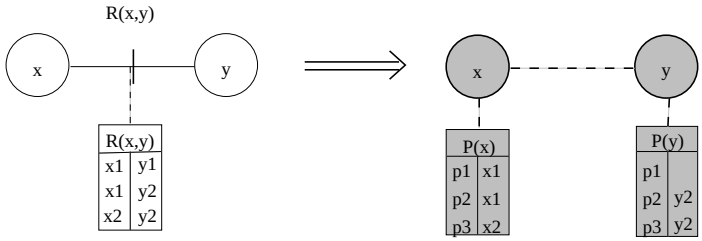
Because a relation edge can be connected to either a pivot node or a single node, the algorithm must consider three different situations: two single nodes, one single node and one pivot node, and finally, two pivot nodes. The following paragraphs define the actions that are performed to handle each of these cases.

- If both x and y are single nodes then the pivot relations $P(x)$ and $P(y)$ are undefined. Because they are connected by a relation edge the solution of the sub-query defined by the pivot-graph consisting of nodes x, y and edge (x, y) is simply $R(x, y)$. The pivot relations are constructed by first assigning a unique identifier to the tuples in $R(x, y)$ using the μ (mark) operation, and then by projecting on the pivot attribute p and the x or y attribute[2].

[2]In the implementation these three operations are typically combined in a single scan of the relation $R(x, y)$.

$$\begin{aligned} T &= \mu R(x,y) \\ P(x) &= \pi_{[p,x]} T \\ P(y) &= \pi_{[p,y]} T \end{aligned}$$

After the operation the two nodes form a pivot graph and the edge connecting them is removed.

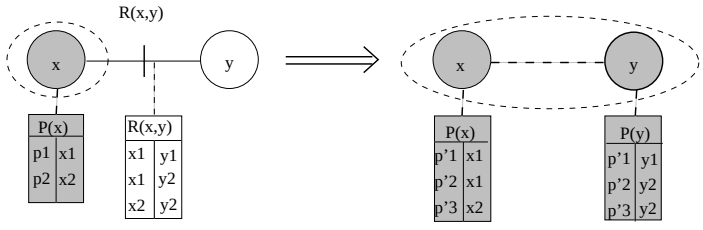


- If node x is a pivot node and y is a single node then the pivot relations $P(x)$ and $P(y)$ are determined by by joining the pivot relation $P(x)$ with $R(x,y)$. The node y is then merged with the pivot-graph associated with x.

  The join operation can invalidate the uniqueness constraint for the pivot attribute. All the pivot relations associated with the nodes of the subgraph must therefore be renumbered. Let $P(x), P(x_1), \ldots, P(x_k)$ denote the pivot relations associated with node x and p' a new pivot attribute, then removal of the target edge implies the following operations:

$$\begin{aligned} T &= \mu(P(x) \bowtie R(x,y)) \\ P(y) &= \pi_{[p',y]} T \\ P(x) &= \pi_{[p',x]} T \\ P(x_1) &= \pi_{[p',x_1]} T \bowtie_{T.p=P(x_1).p} P(x_1) \\ &\vdots \\ P(x_k) &= \pi_{[p',x_k]} T \bowtie_{T.p=P(x_k).p} P(x_k) \end{aligned}$$

This reduction is illustrated in the following figure.



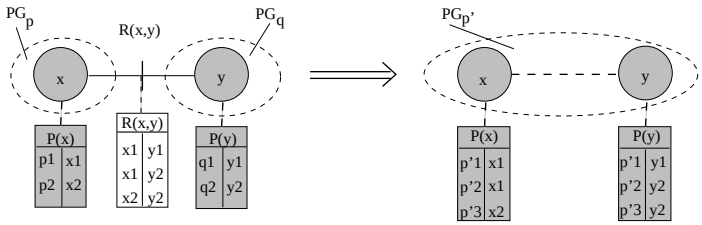
If the pivot relation $P(x)$ is not required in the query result and the node is isolated (i.e. $d(x) = 0$), then the pivot relation is removed. This reduces the renumbering overhead for the remaining query.

- If both x and y are pivot nodes, then node x is associated with a number of pivot relations $P(x), P(x_1), \ldots, P(x_m)$ and node y is associated with a number of pivot relations $P(y), P(y_1), \ldots, P(y_n)$.

  Let p and q denote the pivot attributes of $P(x)$ and $P(y)$, respectively. Then we have to find all the possible pairs (p, q) that satisfy the constraint expressed by the relation $R(x, y)$. In other words by joining the pivot relation with the relation $R(x, y)$ we find all possible combinations. By marking the join result each combination is assigned a unique pivot attribute p'. This result is subsequently used to renumber the pivot relations associated with node x and y.

$$
\begin{aligned}
T &:= \mu(P(x) \bowtie R(x,y) \bowtie P(y)) \\
P(x) &:= \pi_{[p',x]} T \bowtie_{T.p=p} P(x) \\
P(x_1) &:= \pi_{[p',x_1]} T \bowtie_{T.p=p} P(x_1) \\
&\vdots \\
P(x_m) &:= \pi_{[p',x_m]} T \bowtie_{T.p=p} P(x_m) \\
P(y) &:= \pi_{[p',y]} T \bowtie_{T.q=q} P(y) \\
P(y_1) &:= \pi_{[p',y_1]} T \bowtie_{T.q=q} P(y_1) \\
&\vdots \\
P(y_n) &:= \pi_{[p',y_n]} T \bowtie_{T.q=q} P(y_n)
\end{aligned}
$$

The reduction is illustrated in the following figure:



10.7 A SAMPLE TASK EXECUTION

In this section we illustrate the task execution algorithm using the Mail query example. In Chapter 7 this query has been used to clarify the task generation algorithm. This algorithm produce tasks $\langle G, \mathcal{F} \rangle$, that uniquely identify a subquery by a query graph G and a set of edge-fragment assignments. These tasks are allocated to the processors available, taking into account the processor load

relation	attribute 1	attribute 2	relationship
city	y [Address]	x [String]	1-1
sender	L [Letter]	C [Person]	n-1
receiver	L [Letter]	P [Person]	n-1
kids	P [Person]	C [Person]	1-n
address	P [Person]	y [Address]	1-1

Table 10.1: The fragment types used in the Mail query

and fragment distribution (See Chapter 9). The task execution algorithm described in this Chapter, finally, processes each task and reports the result to the Query Scheduler.

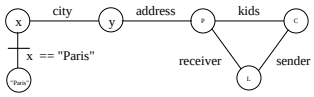
The first step in task execution is the initialization phase. In this phase the fragments referenced in the task assignment are located. If the BAT for the fragment is already available in the local buffer pool, the corresponding relation edge can be immediately bound. Furthermore, the buffer manager is requested to fix the BAT in memory, so that it is not removed during task execution.

If the BAT is not available, the Bat Buffer Manager retrieves it from the buffer of another processor. Task execution can proceed even though not all the relation edges are bound, because the graph reduction algorithm selects only target edges that can be reduced successfully. During task processing the BATs requested arrive and are bound to the graph, so that the query graph can be completely reduced.

The Mail query is represented in a cyclic query graph of five relation edges and a single condition edge. The relations associated with the edges are presented in Table 10.1, which maintains the name, attribute names, attribute types, and relationship.

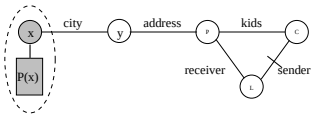
1. The graph reduction algorithm is cost driven. This means that the cost for removing the target edge should be minimal. In this example the first target edge is likely to be the condition edge, because the `City` relation has the smallest cardinality and the selection operation is cheap compared to the other operations.

$$P(x) \quad = \quad \pi_{[p,x]} \mu \sigma_{x='Paris'} City$$



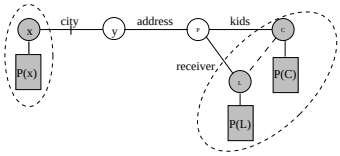
2. The next target edge is the relation edge `Sender`. The pivot relations for attributes C and L are created by simply numbering the tuples in the `sender` relation. The attribute node L and C form a pivot-graph.

$$
\begin{aligned}
T &= \mu Sender \\
P(C) &= \pi_{[p,C]} T \\
P(L) &= \pi_{[p,L]} T
\end{aligned}
$$



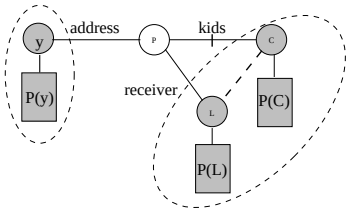
3. Removal of the `city` edge implies an equi-join of the pivot relation associated with node x and the `city` relation. After the join operation the result must be renumbered, so that each solution is identified by a unique pivot attribute. The node x represents an internal node and is not used in the final query result. After the graph reduction x and its associated pivot relation $P(x)$ can be removed, because the node is not connected to the remaining query graph.

$$P(y) = \pi_{[p,y]} \mu (P(x) \bowtie City)$$



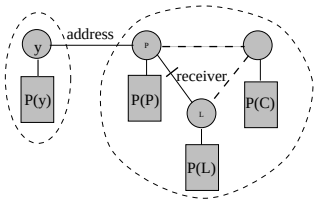
4. In this reduction step the pivot-graph consisting of node C and L is combined with the single node P by removing the relation edge `kids`. The pivot relation $P(P)$ is constructed by joining the `kids` relation with the pivot relation $P(C)$. The pivot relations $P(L)$ is renumbered by joining it with the relation T on the old pivot attribute. After this reduction the nodes P, C and L are merged in a single pivot-graph.

$$
\begin{aligned}
T &= \mu (P(C) \bowtie Kids) \\
P(P) &= \pi_{[p',P]} T \\
P(C) &= \pi_{[p',C]} T \\
P(L) &= \pi_{[p',L]} (T \bowtie P(L))
\end{aligned}
$$



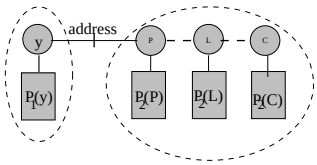
5. It is possible that not all the edges in a pivot-graph are removed. This is illustrated in this reduction step by removing the relation edge `receiver` between nodes P and L. According to the algorithm this reduction is solved by joining the pivot relations $P(P)$, $P(L)$ and `receiver`. The join result is renumbered and is then used to renumber the pivot relations in the sub-graph.

$$
\begin{aligned}
T &= \mu(P(P) \bowtie Receiver \bowtie P(L)) \\
P(P) &= \pi_{[p',P]}(T \bowtie P(P)) \\
P(C) &= \pi_{[p',C]}(T \bowtie P(C)) \\
P(L) &= \pi_{[p',L]}(T \bowtie P(L))
\end{aligned}
$$

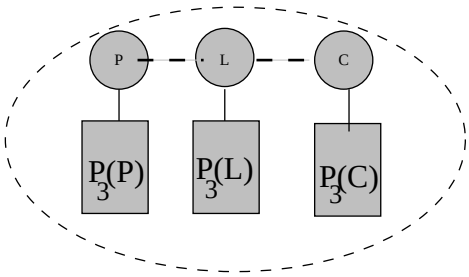


6. Finally the two pivot-graphs identified by node y and nodes $\{P, L, C\}$ are combined by removing the relation edge `address`. After the reduction the internal node y is removed from the pivot-graph.

$$
\begin{aligned}
T &= \mu(P(P) \bowtie Address \bowtie P(y)) \\
P(P) &= \pi_{[p',P]}(T \bowtie P(P)) \\
P(C) &= \pi_{[p',C]}(T \bowtie P(C)) \\
P(L) &= \pi_{[p',L]}(T \bowtie P(L))
\end{aligned}
$$



7. The final result is represented by the pivot-graph identified by nodes $\{P, L, C\}$. The associated pivot relations form the DSM representation of the query result.



10.8 Target edge selection

The goal of the dynamic query optimization algorithm is to minimize the total task execution time. As the selection and removal of the target edge involves join processing on the associated binary relation fragments, the choice of the target edge is critical to the task execution time. In the Wong-Youssefi algorithm, this choice is based on classification of relations in a *small* and *not-small*.

In Goblin the edge that incurs the least processing is selected for removal. For each edge this CPU cost is estimated at task initialization and it is updated at run-time using the profiles of the pivot relations produced. This cost consists of two components: the cost to calculate a new pivot relation and the cost for renumbering already existing pivot relations from a sub-graph.

The first component is based on operand size and the result of an operation. The operand size is known at run-time and the result size can be estimated for

a relational operation using statistics on the size, cardinality and distribution of the attribute values of the operands [SAC+79].

Once a cardinality estimate is found of the resulting pivot relation, the second component is easy to determine, because the pivot relations in the new pivot graph have by definition the same cardinality. Furthermore, their cardinality in the old pivot graph is known.

The operations used in the task execution algorithm are selection, theta-join, semi-join and equi-join. The following paragraphs present formulas which express the cardinality of their result in the cardinality and ordinality of the operand attributes. In these formulas we assume a uniform attribute value distribution to simplify the analysis. The effect of skewed data distributions on intermediate result size is studied in [ST89].

In the following we use R and S to denote binary relations, the symbols A and B to represent the attributes, and $card(R)$, $ord(A)$, $min(A)$, and $max(A)$ for the cardinality of relation R, the number of distinct attribute values, and the minimum and maximum value of attribute A, respectively.

10.8.1 Selection

The query graph allows the specification of selection conditions on a single attribute of the form $A\ \theta\ C$, where $\theta \in \{<, =, >\}$. For the equality predicate, the cardinality of the selection result is estimated by the number of distinct attribute values and cardinality of the relation.

$$card(\sigma_{A=C}R) = \frac{card(R)}{ord(A)}$$

For range selection predicates the formula uses the minimum and maximum attribute values.

$$card(\sigma_{A>C}R) = \frac{max(A) - C}{max(A) - min(A)} card(R)$$

$$card(\sigma_{A<C}R) = \frac{C - min(A)}{max(A) - min(A)} card(R)$$

10.8.2 Theta-join

Attributes in the query graph that are connected through a condition edge express a theta-join operation. These expressions are of the form $A\ \theta\ B$, where $\theta \in \{<, =, >\}$ and A and B represent the attributes. If the condition contains the equality predicate, an equi-join is performed. This case is discussed in the next section.

A reliable estimate of the cardinality of the theta-join result is difficult to make. In the worst case, the join result equals the Cartesian product of both operands. Of course this is generally a pessimistic estimate and does not provide a solid basis for cost driven query optimization.

A better approach is to maintain statistics on these join operations. In the Goblin architecture a processor executes many similar tasks. It is therefore

possible to determine the join selectivity for a theta-join on two specific fragments and use this join selectivity factor to estimate the cardinality for other fragment combinations. For a join of two relations R and S the join selectivity (σ_{RS}) is defined as:

$$\sigma_{RS} = \frac{card(R \bowtie S)}{card(R).card(S)}$$

Given this join selectivity the following formula gives a cardinality estimates for a theta-join on another fragment combination of the same relations:

$$card(R' \bowtie_\theta S') = \sigma_{RS}.card(R').card(S')$$

10.8.3 Equi-join

The equi-join operation is the most common operation in the task evaluation algorithm. Its frequent use is a consequence of the object representation model. The join condition is therefore always expressed on a key and non-key attribute, that often represent OID types. Assuming that the relations R and S are joined on the attributes A and B, where A is a key attribute of relation R, then the cardinality of the result is at most the cardinality of S, because each tuple of S joins with at most one tuple of R:

$$card(R \bowtie_{A=B} S) = card(S)$$

In some cases the type constructors of the data model can be used to get an even more accurate estimate. If the relations R and S store two attributes of a tuple then they are related by a $1-1$ relationship. In the Mail query graph this is exemplified by the `address` and `city` relation. An address is a tuple object and has a unique city attribute. Then the cardinality of the join result is determined by the cardinality of the smallest relation.

$$card(R \bowtie_{A=B} S) = min\{card(R), card(S)\}$$

10.8.4 Semi-join

The semi-join operation is used in the task evaluation algorithm to reduce a pivot relation to a sub-set of pivot attributes. Because the pivot attributes are unique, the cardinality of the result simply equals the cardinality of the smallest relation. Thus:

$$card(R \ltimes S) = min\{card(R), card(S)\}$$

Given these formulas to estimate the result size of an operation, the cost formulas for the operations and the actions performed for each graph reduction, it is possible to associate a cost estimate to each edge in the query graph. At each iteration of the graph reduction algorithm the edge with the minimal cost is selected as target edge.

10.9 Optimization issues

The presented graph reduction algorithm solves general queries represented by query graphs, but leaves still a lot of optimization issues open. In this section we will briefly introduce two optimization techniques that can further improve the task evaluation performance.

One technique exploits semantic constraints introduced by the data model and has already been mentioned shortly in the presentation of the example. If a target relation edge connected to a single node is removed and it is known from the data model that it expresses a $1-1$ relationship then the produced pivot relation will have a unique pivot attribute. Consequently, the other pivot relations do not have to be renumbered.

The other technique aims at reusing intermediate results. Basically, in this technique the pivot relations associated with a pivot-graph are maintained. The effectiveness of this technique strongly depends on the task allocation algorithm. It is only applicable if the task allocation algorithm assigns a task that refers to many fragments used in a task previously executed. In that case this task can share and reuse intermediate results.

To make this work, a naming scheme for intermediate results is required to identify at task initialization time, which results are available in the processor's buffer. From the description of the algorithm we know that sub-query results are uniquely identified by their pivot-graphs and the fragments associated with the edges. In the initialization phase, these sub-graphs and associated pivot relations replace their corresponding nodes in the query graph.

Reusing intermediate results has potentially a great effect on the average task execution time. For instance, given a query on five different relations and a processor pool consisting of five processors, it is possible to reduce the total amount of work by assigning these tasks as follows:

Site	Task sequence
P1	$T(P_1, Q_1, R_1, S_1, T_1); T(P_2, Q_1, R_1, S_1, T_1); T(P_3, Q_1, R_1, S_1, T_1); \ldots$
P2	$T(P_1, Q_2, R_1, S_1, T_1); T(P_1, Q_3, R_1, S_1, T_1); T(P_1, Q_4, R_1, S_1, T_1); \ldots$
P3	$T(P_1, Q_1, R_2, S_1, T_1); T(P_1, Q_1, R_3, S_1, T_1); T(P_1, Q_1, R_4, S_1, T_1); \ldots$
P4	$T(P_1, Q_1, R_1, S_2, T_1); T(P_1, Q_1, R_1, S_3, T_1); T(P_1, Q_1, R_1, S_4, T_1); \ldots$
P5	$T(P_1, Q_1, R_1, S_1, T_2); T(P_1, Q_1, R_1, S_1, T_3); T(P_1, Q_1, R_1, S_1, T_4); \ldots$

If the task reduction algorithm stores the appropriate intermediate results this task assignment has the effect that after the first task has been executed each processor can calculate the next task result by combining the stored intermediate result with the new fragment.

10.10 Conclusion

Query execution in Goblin is based on the dynamic query processing proposed in this thesis. This chapter discussed the task execution algorithm employed in the prototype. The dynamic features encompass adaptivity towards fragment size and the size of intermediate results, and adaptivity towards skew in the fragment arrival rate.

The first feature is the result of a dynamic query optimization scheme, which determines the join execution order at run-time, based on up-to-date relation fragment profiles. As tasks are evaluated in main-memory, tasks are optimized towards CPU cost. The task evaluation algorithm is like the Wong-Youssefi algorithm based on graph reduction. It includes, however, three new aspects.

First, it can incorporate a mechanism for multi-task optimization by reusing intermediate results. In the proposed dynamic query processing scheme, where a large number of similar tasks are executed by a query processor, this has a large potential.

Second, the execution order of the individual join operations can be decided at run-time. The query optimization is not based on heuristics, but based on the actual fragment profiles.

Thirdly, the algorithm allows overlapping of fragment I/O with the graph reduction process. This is useful, because the time required to retrieve a fragment from a remote processor is of the same order of magnitude as a single equi-join operation. Thus a task execution can proceed even though not all the fragments are available.

These features lead to an efficient, adaptive query processing mechanism, which is robust and adaptive to changes in the load distribution and data skew.

Furthermore, this approach has reduced the generally difficult optimization problem in parallel database system into two controllable and distinct smaller problems: a local (CPU) optimization problem at each of the Query Processors and a task allocation problem (IO) at the Query Scheduler.

Many aspects have not been fully addressed in this chapter and will be investigated in our future research. The graph reduction algorithm leaves room for further optimization. Especially exploitation of the relationship between the data model and the graph reduction algorithm shows promise. If it is known that two relations form a sub-set of each other, a semi-join operation can be avoided. Furthermore, in many cases it is not necessary to renumber the pivot relations in a sub-graph, thereby saving many join operations.

With respect to the reuse of intermediate results not all has been said. Specifically, the heuristics to decide what intermediate results must be maintained have to be developed. Furthermore, an analysis of its effectiveness is required to get insight into the tradeoff between the use of memory resources for storing intermediate results and the cost of their reconstruction.

Chapter 11

Goblin evaluation

11.1 Introduction

In the previous chapters many design decisions have been made and techniques have been explored for implementing an OODBMS. The Goblin prototype incorporates many of these techniques. In particular, the current version exploits partitioning information through a two-level query-processing scheme. The first level generates tasks by running the query on a summary data base. The second level evaluates these tasks for the particular fragment combinations in main-memory.

The system is designed to run both shared-memory and shared-nothing architectures. Currently, there are two target platforms: one is collection of SGI/Indigo workstations (34364.3 Dhrystones/sec) running UNIX and the other is a multi-processor system consisting of 8 Intel 80386 (7142.9 Dhrystones/sec) running the distributed operating system Amoeba [MvRT+90]. On both platforms the processors communicate through an Ethernet connection. A generic thread package and interprocess-communication package is defined to facilitate porting the architecture to other platforms.

The current implementation is not yet fully operational on the parallel platforms. However, all key algorithms have been implemented and can be run in isolation. Thus, even though the system is only partially implemented, we can obtain a fairly accurate performance prediction.

In this chapter we illustrate the performance of the key algorithms and components of the Goblin system. In the next section the relational operations provided by the Goblin kernel and the communication sub-system are timed. In the third section the effect of the two-level query-processing scheme is examined

for a simple query. In the fourth section we provide a performance prediction for the parallel system for the Wisconsin benchmark. We conclude with a summary of our findings.

11.2 The Goblin kernel

Goblin is designed as a main-memory parallel data-base system. This has a major influence on its design. In a main-memory parallel system the overall performance strongly depends on efficient processing and data communication. This functionality is provided by a small kernel, which incorporates a communication module and a processing module.

These modules will be discussed in further detail in the following sections.

11.2.1 Communication

In the Goblin system Query Schedulers and Query Processors are processes. These processes are created at system startup time and communicate with each other using message passing primitives provided by the communication module.

The Query Processors use the communication primitives to retrieve data fragments and report task results to the Query Scheduler. The data fragment messages are important for the overall query performance, because a task can only be completed until all the fragments it requires are locally available. Through clever buffer management and task allocation the average number of fragment requests can be reduced (See Chapter 9). However, due to a limited amount of buffer memory fragment I/O can not be completely avoided. An efficient implementation is therefore necessary. For query processing two factors are important: the response time, i.e., the time measured from the fragment request until its arrival, and the maximum throughput of the network i.e. the maximum number of fragments that can be sent between process pairs.

The Query Scheduler uses the communication primitives to control the task execution. It assigns from its task table a number of tasks to the query processor with a lower than average load. The load information on Query Processors is an example of the information feedback from the Query Processors to the Query Scheduler. For system performance a low response time for task assignment and for feedback information is essential. For instance, out-of-date load information on the Query Processors reduces the load balancing algorithm's efficiency.

We have determined the response time for data transfer on both platforms. In the experiment a client process sends a data request to a server process on another site which then returns the data.

The results for the SGI network are disappointing due to the network load and process scheduling delays. Its fast processor ensures that the measured system- and user-time for a data transfer of 100,000 bytes does not exceed 100 msec. The network and scheduling delay, however, is in the order of a few seconds.

The Amoeba operating system is designed to support distributed applications, which is visible in the results presented in Table 11.1. It shows the response time for a small message (500 bytes) and a large message (30,000 bytes). The first size corresponds to a typical control message and puts the communication

type	size	response time
control	500 byte	4 msec
data	30,000 byte	98 msec

Table 11.1: The communication response time on the Amoeba platform

overhead in a perspective. The large message represents a typical fragment retrieval operation.

Both the response time for control messages and data messages are reasonable compared to the processing time of the relational operations. A proper task allocation and buffer replacement scheme can reduce the average number of fragment requests for a three way join task to 0.5 –1.0 fragment requests (Chapter 9). Under these conditions the task spends only 50 – 100 msec on communication and 800 – 1000 msec on processing (Amoeba platform).

11.2.2 Processing

The task evaluation algorithm of the Query Processor determines at run time the execution order for the operations specified by the query graph associated with the task. At each reduction step of the algorithm the query processor executes a relational operation. These operations are uninterrupted by I/O and other operations. The total task execution time equals the sum of the response times of the individual operations including the overhead of the task evaluation algorithm. The performance of these relational operations is therefore important for the task execution time.

The basic operations called by the graph reduction algorithm are select, join and semi-join. These operations take binary relations as operands and return the result as a binary relation. Apart from these operations, the processing module contains operations for partitioning binary relations. Either on one or both attributes using a range-based or hash-based partitioning scheme. In the following we present the results of performance measurements of these basic operations on the SGI and Amoeba platform.

11.2.3 The join and semi-join operation

Both the join and semi-join operation use a hash-based algorithm. First a hash index is created on the join attribute of one operand relation. Entries having the same hash value are administrated in a collision list. The hash table size is chosen as a power of two, such that the average collision list-length does not exceed four entries. Each tuple of the other relation is then used to probe this hash index. As long as the hash function uniformly distributes the tuples of the first relation over the hash domain and the cardinality of the result size is less than cardinality of the largest relation, the response time is linear in the operand cardinality and given by:

$$t = t_h|R_1| + t_p|R_2| + t_c|S|$$

where t_h, t_p and t_c represent the time to create a hash entry, the time to probe the hash table and the time to construct a result tuple respectively.

The factors t_h and t_p are dominated by the function calls required to calculate the hash value or to compare two values. Assuming that this cost is approximately given by t_f, we can express the previous cost formula as follows:

$$t = t_f|R_1| + (t_f + kt_f)|R_2| + t_c|S|$$

where k is the average collision list length.

From this simple analysis we conclude that in main-memory data bases hash tables must be constructed on the *largest* relation. Furthermore, we expect the response time to show a saw-tooth characteristic as a function of the cardinality of the second operand R_2. This is because the hash table size assumes only values that are a power of two. The average collision list length will then increase until it reaches four.

These effects are illustrated by the measurements shown in Figures 11.1 and 11.2 for the join and semi-join operation on the SGI platform and in Figures 11.3 and 11.4 for the Amoeba platform. Each graph shows three situations: (1) both operands have the same cardinality $|R_1| = |R_2|$, (2) the cardinality of the hash table operands is 10% of the other operand $|R_1| = 0.1|R_2|$, and (3) the cardinality of the probe relation is only 10% of the hash relation $0.1|R_1| = |R_2|$.

The Figures show the predicted saw-tooth shaped curves resulting from the hash table size. For the join operation on equal sized relations discontinuities occur for relations having cardinalities 8000, 16,000, 32,000 and 64,000. Each point in the graph is represented by the average and standard deviation from many measurements.

The potential for dynamic query optimization is illustrated by the difference between the two execution orders. The curves show that by constructing a hash table on the largest relation, the performance can be improved by a factor of three. This optimization is implemented in the Amoeba version of the join algorithms. Furthermore, the collision list anomaly is also solved for this version as illustrated in Figures 11.3 and 11.4. In this implementation the hash table size is a linear function of the operand size.

The absolute join performance in main-memory is impressive. On a single processor 33 Mhz SGI/RS3000 a 100k $\times$ 10k join is performed within 520ms, and a 100k $\times$ 100k join in approximately 4.5 seconds. The result sizes for these joins are 10k and 100k, respectively. The performance characteristic for the semi-join operation is comparable to the join performance, but consistently lower because in constrast to the join operation it does not have to test all possible tuple combinations of its two operands.

11.2.4 The select operation

The basic select operation scans the tuples of its operand and evaluates a range condition on it. An alternative implementation uses an index on the selection attribute of the operand if it exists, thereby considerably reducing the processing time.

In the following we have only considered the execution time for a scan-based implementation of the select operation. For this operation the execution time is

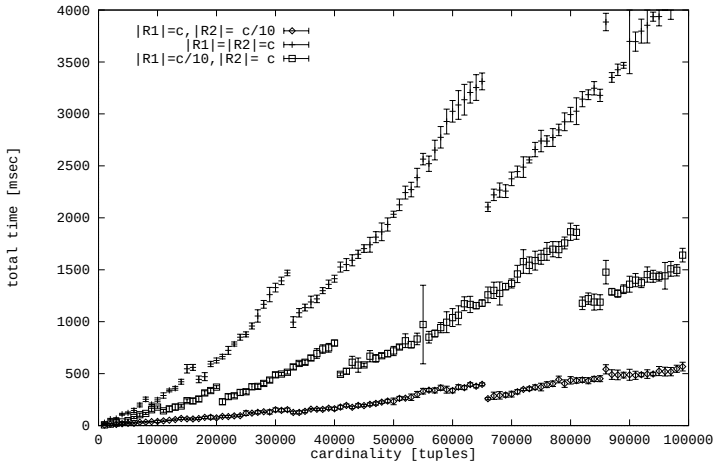


Figure 11.1: Join execution time on SGI as a function of the operand cardinality

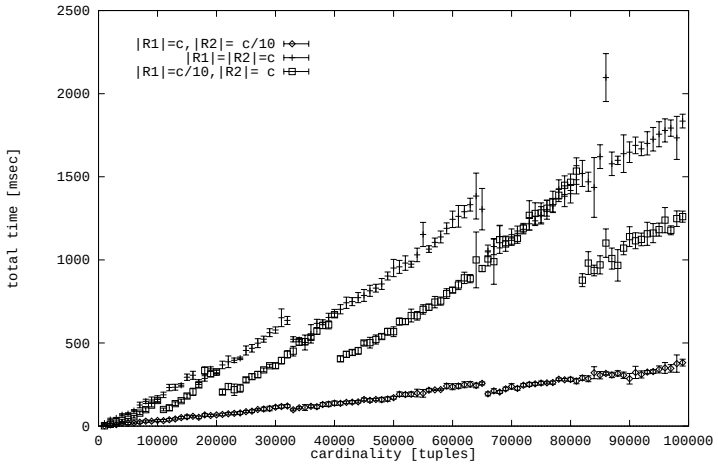


Figure 11.2: Semi-Join execution time on SGI as a function of the operand cardinality

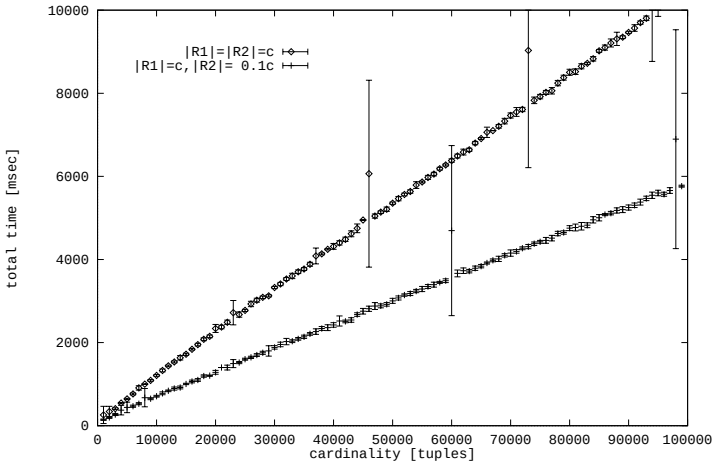


Figure 11.3: Join execution time on Amoeba as a function of the operand cardinality

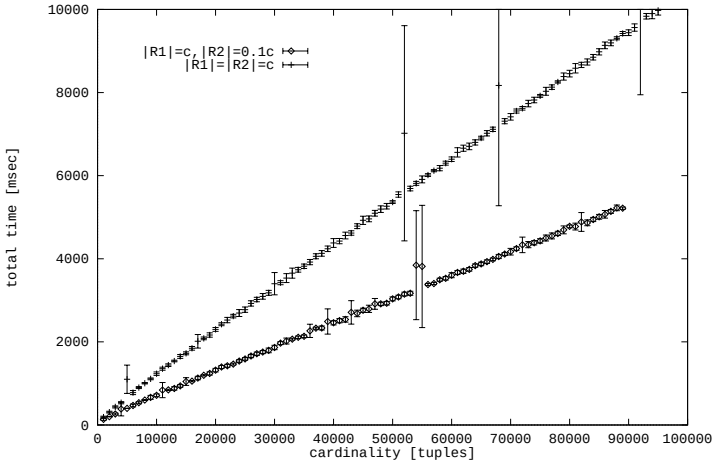


Figure 11.4: Semi-Join execution time on Amoeba as a function of the operand cardinality

simply a linear function of the cardinality of the input relation and the cardinality of the output relation:

$$t = t_f|R_1| + t_c\sigma|R_1|$$

where t_f, t_c an σ represent the time to respectively evaluate the comparison function for an input tuple, the time to produce a result tuple and the selectivity of the operation. Figures 11.5 and 11.6 show the execution time for the SGI and Amoeba implementation as a function of the input cardinality and selectivity as a percentage.

The 3-D graph shows that the time to construct a result tuple t_c is the dominant cost factor. For a selectivity of 0% the selection cost increases slowly to a maximum of 10 ms in the cardinality range of 1k - 100k tuples. At a 100% selectivity, the operation on a 100 Kbyte relation takes 600 ms.

11.2.5 The partition operation

Finally we look at the execution cost for the partition operation. As this operation distributes its input relation tuples over a number of output relations and does not reduce the number of tuples, the response time is a linear function of the input cardinality of the form:

$$t = t_p|R_1| + t_s$$

where t_p represents the partition overhead per tuple and t_s the constant start-up cost.

The measurements confirm that the execution cost is a linear function of the cardinality. Furthermore, the initialization overhead is negligible. For the SGI version we find that $t_p = 53.9$ msec/1000 tuples. This figure is quite high compared to the cost of a join and select operation and indicates that partitioning is best performed a priori.

11.3 Query processing

The two level query processing scheme consists of task generation and task execution. In Goblin the binary relations are a priori partitioned and distributed over the processors. With each binary relation a summary relation is associated which maintains for each fragment its partition information consisting of the fragment identifier and the hash values for the attributes.

The task generation algorithm executes the query on the summary relations to determine which fragment combinations contribute to the final query result. The tasks produced are stored in a task table and subsequently selected by the task allocation algorithm for execution on the Query Processors.

The partitioning degree and the fragment cardinality influence the query execution time. For a high partitioning degree the cardinality of the summary relations is high and the task generation overhead is high compared to the task execution time. If the relations are partitioned into a few large fragments, the task generation algorithm produces only a few tasks having a high execution time.

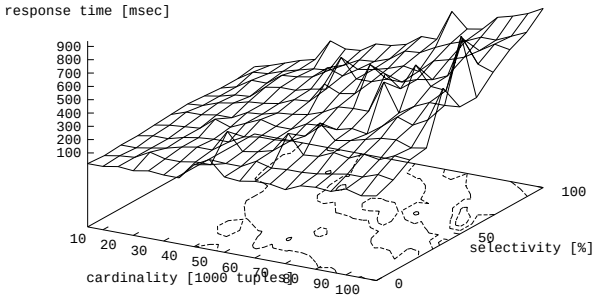


Figure 11.5: The select execution time on the SGI platform as a function of the input cardinality

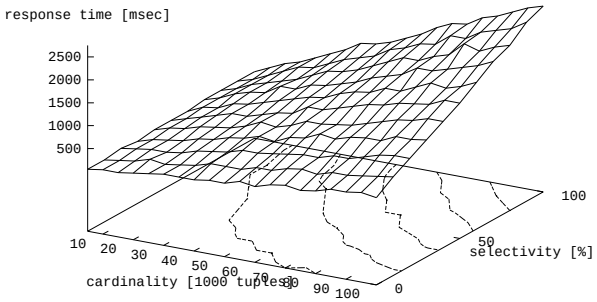
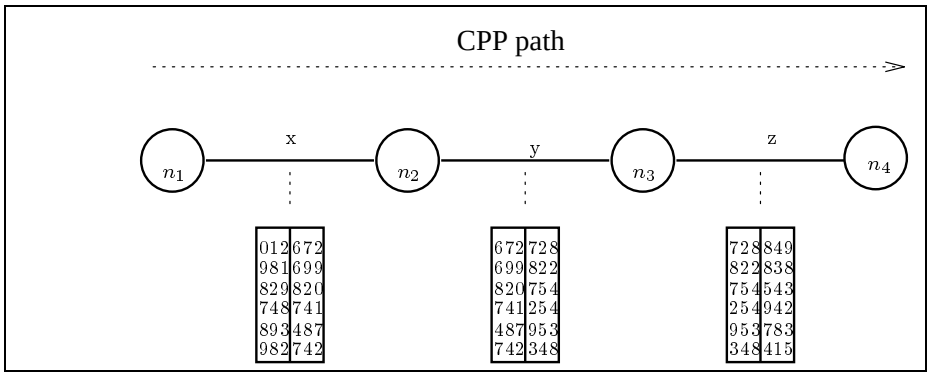


Figure 11.6: The select execution time on the Amoeba platform as a function of the input cardinality

In this section we show the influence of the partitioning degree on the total execution time and the minimal response time. The minimal response time is defined as the response time obtained if all the tasks were executed in parallel.

For this purpose we consider the query represented by the following query graph:



For this query graph we will first discuss the task generation in detail and then show the operations performed in the task evaluation process. Finally we show the overall performance by combining the task generation and task evaluation cost.

For the experiment the relations x, y and z represent binary relations with two integer attributes. The attributes are unique and randomly selected from the integer domain $[1, c]$, where c equals the cardinality of the relation. Before the query is executed the binary relations are partitioned into variable number of partitions. In the experiment the number of partitions ranges from 1 to 100.

11.3.1 Task generation

The nodes in the graph n_1, n_2, n_3 and n_4 correspond to object sets. The relationships between these objects are maintained in the binary relations x, y and z. The task generation algorithm determines a CPP path in the query graph and uses it to generate fragment combinations that possibly contribute to the query result. Because the query graph is a linear chain a possible CPP path is given by $\langle x, y, z \rangle$. Because all the summary relations have the same size the start node can be either x or z.

This edge sequence identifies a program which when run on the summary relations returns the fragment combinations contributing to the query result. In the following program (See Table 11.2) the join operation on two binary relations $P[a, b]$ and $Q[c, d]$ is defined as: `join(P,Q)` $\equiv \pi_{[a,d]}(P \bowtie_{b=c} Q)$. The `mark` operation applied to a binary relation $R[a, b]$ returns a relation $S[p, a]$, where attribute p is a unique pivot attribute.

The `remark` is similar to the `mark` operation. It invents for each tuple of its operand relation a new pivot attribute, but does this for both attributes at the same time. Applied to a relation $R[a, b]$ the `remark` operation returns the relations $S[p, a]$ and $T[p, b]$. This operation supports the renumbering of constructed pivot relations as described in Chapter 7.

```
p1-x = mark(x)
x-y = join(x,y)
p1-y = join(p1-x,y)
p2-p1 , p2-y= remark(p1-y)
p2-x = join(p2-p1,p1-x)
y-z = join(y,z)
p2-z = join(p2-y,y-z)
p3-p2 , p3-z = remark(p2-z)
p3-x = join(p3-p2,p2-x)
p3-y = join(p3-p2,p2-y)
```

Table 11.2: The summary query program for the example query

The solution to the summary query is represented in DSM format by the *pivot* relations p3-x,p3-y and p3-z. Each fragment combination is identified by a unique pivot value. The set of tasks can therefore be constructed by joining these three relations on their pivot attribute.

The binary relations are hash partitioned on both attributes. Thus if the hash function on the first attribute assumes n different values and the hash function on the second attribute assumes m values, the combination of them partitions the relation in $m \times n$ fragments. For simplicity we choose $m = n$ in the experiment leading to the quadratic partitioning degrees $p \in \{1, 4, 9, 16, 25, 36, 49, 64, 81, 100\}$.

With this partitioning the join attribute in each relation assumes $\sqrt{p}$ different values. If two summary relations of size p are joined each tuple in one operand relation will match $\sqrt{p}$ tuples in the other relations. The join result has therefore a cardinality of $p\sqrt{p}$. With the next join operation the result size grows again by a factor of $\sqrt{p}$. Consequently, the summary query cost grows as a power of the square root of the partitioning degree $O(\sqrt{p}^n)$, where n is the number of relations in the query graph.

Figures 11.8 and 11.7 show respectively the response time and the number of generated tasks as a function of the partitioning degree. Fortunately, even though the total number of tasks grows enormously, the absolute cost for summary query processing is acceptable for small partitioning degrees. Furthermore, in the figures the measurements coincide with the results from the model for the number of generated tasks and the summary query time, which are $f(p) = \sqrt{p}^4$ and respectively $f(p) = 3.5\sqrt{p}^4$.

In any case, the summary query cost must be balanced with the task evaluation cost. Consequently, the partitioning degree should depend on the cardinality of the binary relation. In the following section we consider the relation between task evaluation cost and partitioning degree.

11.3.2 Task evaluation

The task evaluation algorithm is based on reduction of the query graph. The algorithm selects at run time target node and edge combinations and propagates

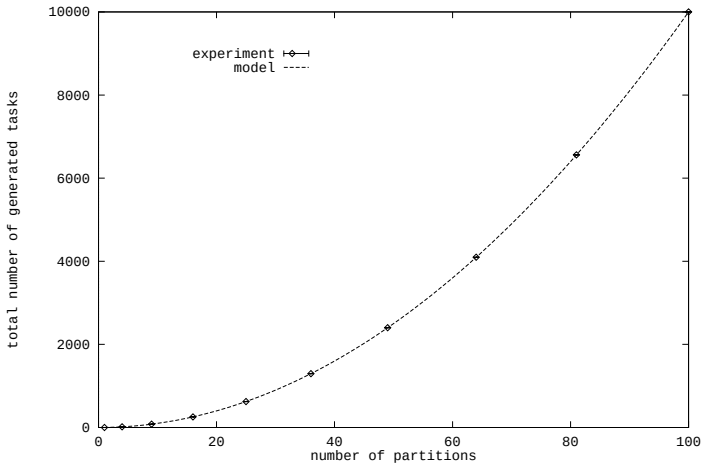


Figure 11.7: The number of generated tasks as a function of the partitioning degree

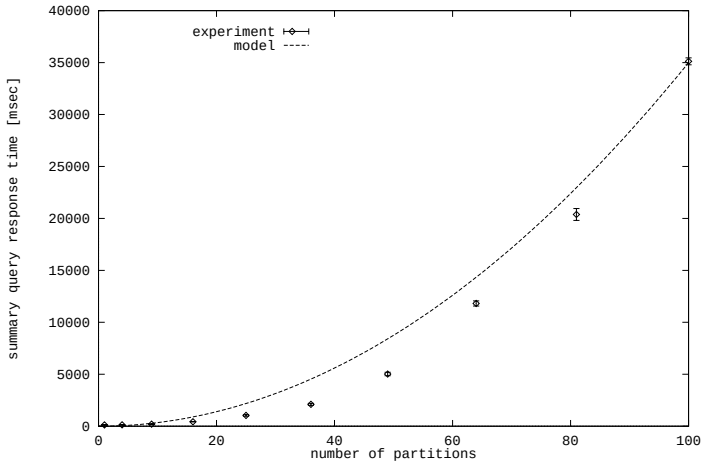


Figure 11.8: The summary query response as a function of the partitioning degree

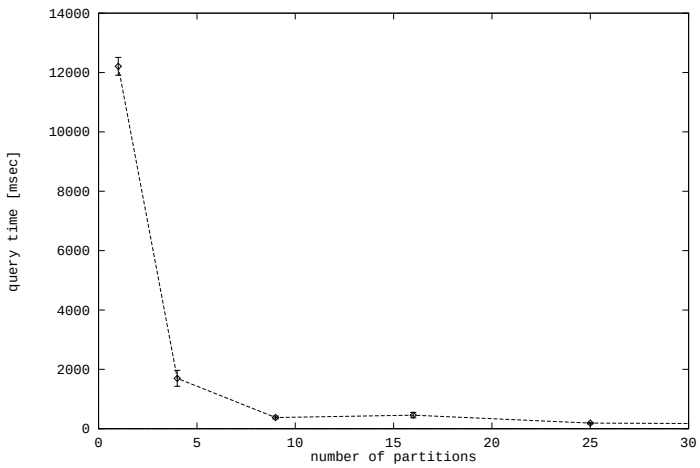


Figure 11.9: The average task execution time as a function of the partitioning degree.

the implied constraints to the connected edges until the graph is reduced to a single edge. This edge and its associated binary relation is then used to construct the pivot sets for the distinguished nodes[1].

Currently there exists only a Prolog implementation of the graph reduction algorithm. This program generates for a given query graph and set of distinguished nodes a task evaluation program. For the moment the dynamic optimization is and reuse of intermediate results is not included.

For task evaluation we consider the query on the relations x, y and z introduced in the previous example. Assuming that the distinguished nodes are n_1, n_2 and n_3. Then for this example a (possible) task evaluation algorithm is given by:

```
t1 = join(x,y)
t2 = join(t1,z)
n1 = mark(t2)
n2 = join(a1,t2)
n3 = join(a1,t1)
```

We expect the average task execution time to be inversely proportional to the partitioning degree. There are two reasons for this. First, because the join execution time is linear in its operand size and result size (See Section 11.2.3) and the partitioning degree is inversely proportional to the operand size. Secondly, because the cardinality of the total query result is independent from

[1] i.e. nodes associated with projection attributes

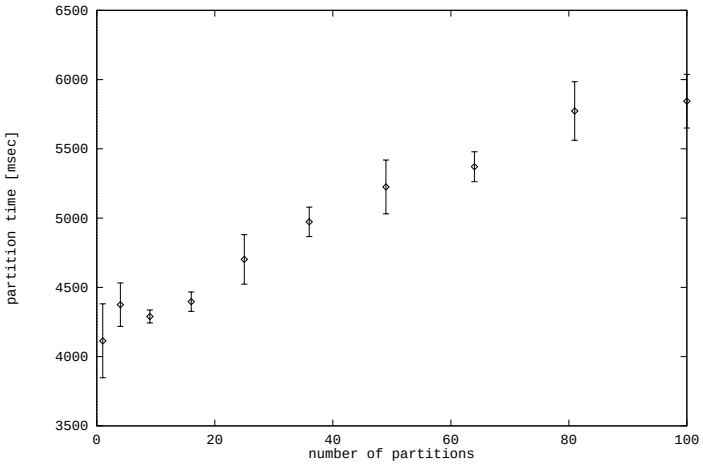


Figure 11.10: The partitioning time on the as a function of the partitioning degree

the partioning degree, the cardinality of the task result is inversely proportional to the number of tasks as presented in Figure 11.7 and therefore only adding to this effect.

In the experiment the average task evaluation time is determined as a function of the partitioning degree. The source relations contain 100,000 tuples each. Given a partitioning degree ranging from 1 to 100 the fragment cardinality ranges from $100,000$ to $1,000$ tuples. The results are shown in Figure 11.9. The average task execution time decreases fast as the partitioning degree increases. Already at a partitioning degree of 9 the task execution cost is an order of magnitude less than the original cost.

11.3.3 Partitioning overhead

For completeness we have also measured the time required for partitioning the three relations. In the experiment the relations are locally available and the resulting fragments are also stored locally. The summary relations are produced as a by-product of the partitioning operation.

The result is presented in Figure 11.10. The partitioning cost is surprisingly a linear function of the number of partitions. This could indicate a flaw in Goblin's memory allocation and requires our future attention.

A more important observation is that the partitioning cost is considerable compared to the task execution cost and summary query cost. Furthermore, the cost is high even for low partitioning degrees. For instance, it takes almost 6 seconds to partition the three relations into 100 fragments. Therefore, it is better to partition relations a priori.

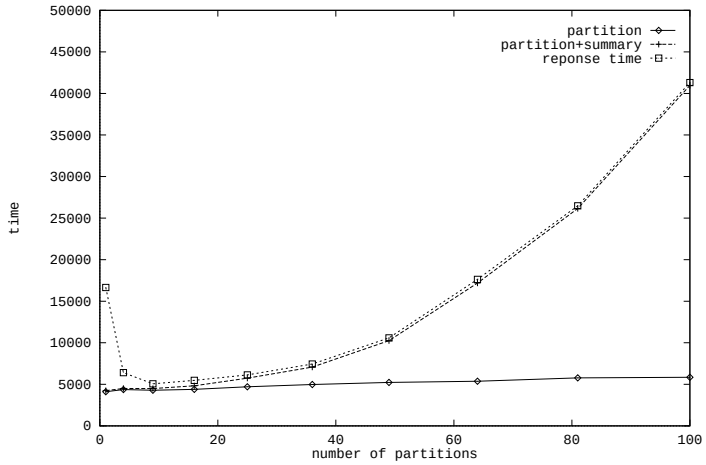


Figure 11.11: The minimum query response time as a function of the partitioning degree

11.3.4 Combining the results

In the previous subsections we presented the basic cost factors for running the example query in an operational system. By combining the partitioning cost, the summary query cost, the number of produced tasks and the average task execution time with the basic communication cost we can estimate the minimum response time (Figure 11.11) and total time (Figure 11.12) as a function of the partitioning degree.

The minimum response time is defined as the time obtained when all the tasks are executed in parallel. It is defined by the following formula:

$$t_{min} = t_{partition} + t_{summary} + t_{task} + 3t_{comm}$$

The partition time $t_{partition}$ offsets the response time with a relatively constant 5 seconds. The task execution time $t_{task}$is high between a partition degree of 1 and 9 but negligible for higher partitioning degrees; in the order of 25 msec. The summary query cost dominates the response time for partitioning degree higher than 9. The last factor is the communication cost for retrieving three fragments. The communication cost can be reduced to a single fragment retrieval by caching the fragments.

This minimum response time represents of course only a lower bound. The time required for distributing the tasks over the available processors and the fragments over the processors is not included. Although it is possible to distribute the tasks over the available processors in a single broadcast message and to distribute the fragments over the available processors in a maximum of $3.p$ broadcast messages, where p is the partitioning degree, it must be said that this scheme is not employed in the Goblin architecture.

The total time estimate sums the partitioning time, summary query execution time and total task execution time. It corresponds to the situation where the

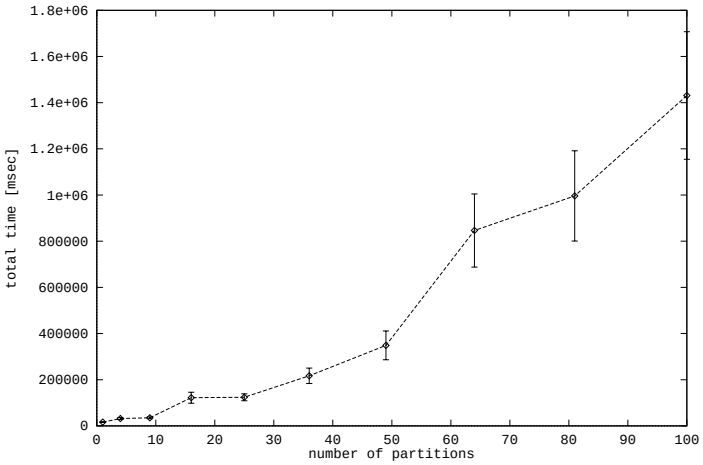


Figure 11.12: The total query processing time as a function of the partitioning degree

query is evaluated on a single processor. The following formula is used to obtain this measure:

$$t_{total} = t_{partition} + t_{summary} + n_{tasks} t_{task}$$

The results are shown in Figures 11.11 and 11.12. It shows that the optimal partitioning degree for the minimum response time for this example is reached for a low partitioning degree $p = 9$. For larger partitioning degrees the processing time for the summary query increases fast. Therefore, for obtaining better results for parallel query processing the summary query cost must be reduced. (For instance by executing the summary query in parallel.)

Note, however that the minimum response time without partitioning for this three way join query on 100,000 binary relations is a low as 580 ms and the total time is in that case 30.8 seconds.

11.4 The Wisconsin benchmark

The Wisconsin benchmark [DeW91] is a well known benchmark used to compare the performance of relational systems. To get a rough idea of the relative performance compared to other data-base systems, most new systems have run the `joinABprime` query of the benchmark.

The benchmark uses three synthesized relations `A`,`B` and `C` consisting of thirteen integer attributes and three 52-byte string attributes. The length of each tuple is therefore 208 bytes, assuming no storage overhead. In the original benchmark the cardinality of the relations is fixed. The small relation `C` contained 1,000 tuples and the two larger relations `A` and `B` 10,000 tuples.

The size of these benchmark relations (2 Mbyte and 200 Kbyte respectively) is relatively small compared to the physical address space of todays computers.

For this reason the benchmark relation cardinality is scaled up to compare the performance of data-base machines. The performance of these systems has been measured on relations containing 100,000 and 10,000 tuples (and more [DeW91]). Because the resulting relation size is at least 20 MByte it is unrealistic even for main-memory data-base machines to execute the Wisconsin queries in main-memory without partitioning the relations.

The two integer attributes `unique1` and `unique2` are uniformly distributed unique random values in the range $[0, max - 1]$, where max is the cardinality of the benchmark relation. In the *joinABprime* benchmark the relation `A` is joined with the relation `Bprime` on the attributes `unique1` and `unique2`. The `Bprime` relation is constructed by selecting 10% of the `B` relation. Because both join attributes are key attributes, the resulting relation contains 10,000 tuples.

In the DSM storage model the Wisconsin relation is represented by thirteen binary relations, one for each attribute. Each relation stores the association of a tuple identifier and an attribute value. Finding all the attribute values associated with a specific tuple requires joining these relations on the tuple identifier. Therefore the relations are hash partitioned on the tuple identifier to reduce this reconstruction cost.

Similar to the previous experiment the query is divided in two steps: task generation and task evaluation. The task generation algorithm traverses a CPP path through the query graph and runs the query on the summary relations. This implies a join expression over the 26 summary relations. This seems to be a prohibitive amount of work. Fortunately the summary relations are very small and the partitioning on the tuple identity leads to a small summary query result as well.

For instance, assume that both relations are partitioned in p fragments. Then all the fragment combinations of relation `A_unique1` and `B_unique2` potentially contribute to the query result. Furthermore, as each `A_unique1` and `B_unique2` fragment uniquely identify fragments for the remaining `A` and `B` attributes, respectively, a total of p^2 tasks is expected.

The result of this experiment is shown in Figure 11.13. Although the summary query is executed on a total of 26 relations, the partitioning on the tuple identifier leads only to a quadratic increase of the query execution time as a function of the partitioning degree.

The task execution time is measured separately. For a partitioning degree ranging from $[1, 100]$ the cardinality of the `A` fragment ranges from $100,000$ to $1,000$. The result of this experiment is shown in Figure 11.14.

If we combine the previous results to determine the minimal query execution time for this query we see that for a partitioning degree of 6 the query response time is 4000 ms (Figure 11.15). This graph includes estimated communication cost.

In Table 11.3 we compare the estimated joinABprime query execution time to the results obtained by other data-base systems. The absolute performance is of the same order of magnitude as the other main-memory systems. There is room for improvement, however.

First, we have observed that the summary query cost dominates the query execution time. Only 1.6 seconds of the estimated 4 seconds is used for com-

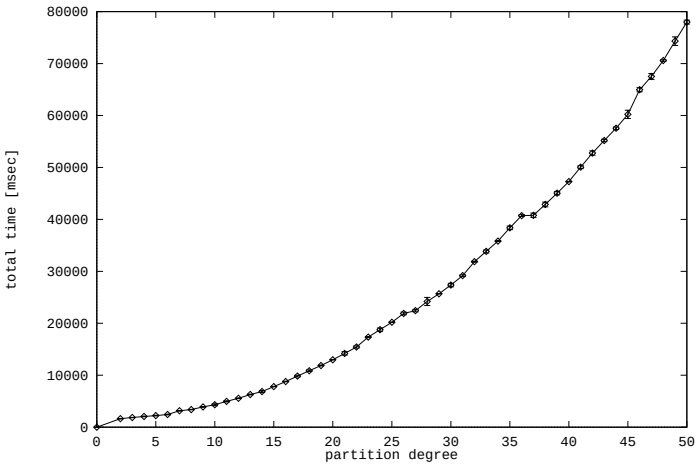


Figure 11.13: The joinABprime summary query execution time as a function of the partitioning degree

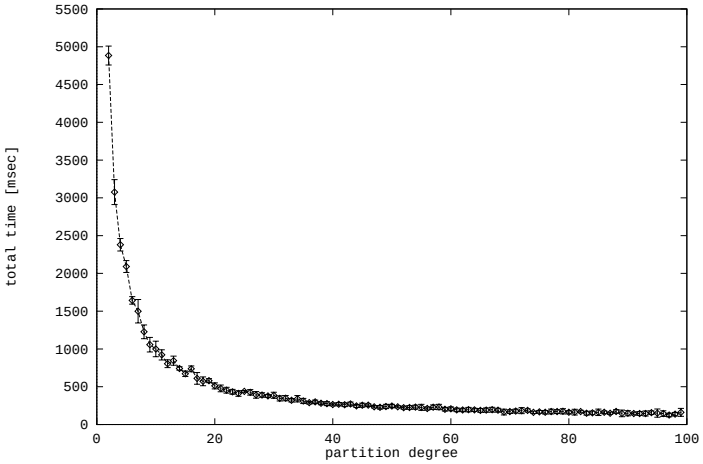


Figure 11.14: The joinABprime task execution time as a function of the partitioning degree

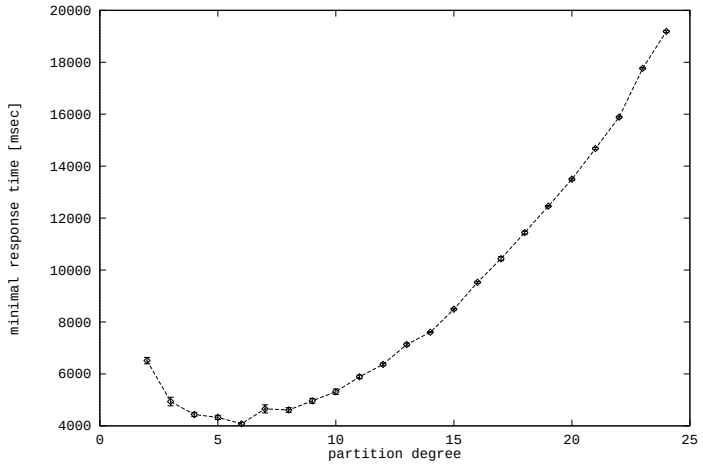


Figure 11.15: The joinABprime minimal response time as a function of the partitioning degree

System	#proc	response time	
Silicon DBM	3	23.900 sec	[LR88]
PRISMA	10	6.132 sec	[Wil93]
PRISMA	30	2.034 sec	
DBS3	10	1.8 sec	[BCV91]
Goblin/SGI	6	4 sec	

Table 11.3: The minimum response time for some parallel main-memory database systems for the `joinABprime` query.

puting the joinABprime tasks in parallel. We can amortize the summary query cost over multiple queries by storing its result. Each time the user evaluates the joinABprime query, the summary query result can be reused. Furthermore, the implementation of the summary query algorithm can be improved considerably. In the current implementation the final pivot phase, - where the tasks are produced- , is implemented by joining the pivot relations. Alternatively, this can be implemented using cheap lookup operations.

With these modifications we expect a response of 2.5 seconds.

11.5 Conclusion

The timing of the individual operations shows that the relational algebra operations on the binary relations are efficiently implemented.

The important performance factors in Goblin's two-level query-processing scheme are summary query cost, task evaluation cost and communication cost. On the basis of measurements of these factors we determined the minimum re-

sponse time as a function of the partitioning degree that can be obtained on a parallel platform .

It turned out that the minimum response time is dominated by the summary query cost. This cost increases exponentially with the partitioning degree and is only reasonable for small partitioning degrees (≈ 9). The task evaluation cost and communication cost are negligible compared to this factor.

Nevertheless, compared to other main-memory parallel systems we arrive at a reasonable response time estimate of 4 seconds for the `joinABprime` query of the Wisconsin benchmark, which can be reduced to 2.5 seconds by improving the implementation of the summary query algorithm.

Furthermore, the communication overhead for task distribution and task control is small compared to the task execution cost, which justifies a limited amount of parallel execution. The overall resource consumption can be controlled because the Query Scheduler can determine the number of processors used for the task evaluation.

The effect of load balancing is not covered by the experiments. This aspect will be studied in future experiments once the Goblin prototype is fully implemented.

Chapter 12

Summary and Future Research

12.1 Introduction

This thesis presents the design and analysis of a dynamic query processing architecture for the Goblin parallel OODBMS. The primary objective of this research was to design a query processing architecture that can effectively and efficiently cope with skewed data distributions and dynamically changing load distribution.

The design of an efficient parallel DBMS is a complex task, because many design issues that affect the performance depend on each other. For instance, a shared-memory multiprocessor requires a different query processing scheme than a shared-nothing architecture. Furthermore, load balancing is more difficult to achieve in a shared-nothing architecture.

To achieve this ambitious goal we fixated from the onset a few design decisions based on technological trends and requirements of the envisioned application domains. Consequently, the baseline for the Goblin design was that it should be a parallel main-memory OODBMS designed for a shared-nothing architecture. Once we made this decision we could concentrate our work on the storage model and on the query processing architecture.

The current design is the result of extensive testing and performance evaluation based on mathematical models, simulation models and benchmarks on prototype implementations of key algorithms. Although the prototype system is not yet fully implemented we concluded on the basis of the performance of key algorithms that this architecture shows a competitive performance compared to other main-memory parallel DBMS systems.

12.2 The main contributions

The research addressed many design aspects that have an impact on the performance of a parallel main-memory OODBMS. In global terms we can summarize the research contribution of our work in the following two points:

- The development, analysis and performance evaluation of a novel two-level dynamic query processing scheme.
- The design, implementation and performance evaluation of a storage model for a parallel main-memory OODBMS.

The results, their consequences and points for further research are discussed in the remainder of this chapter. Section 12.2.1 discusses the results and consequences for the storage model and section 12.2.2 the results and consequences for the query processing scheme. We conclude this thesis with an overview of the remaining research questions.

12.2.1 The Goblin storage architecture

The storage model is an important performance factor for a data base system. It is specifically designed to take advantage of the main-memory approach and to provide flexible storage of objects in a parallel system.

Object representation

We compared three alternative storage models for object representation for the main features of the Goblin architecture: the object-oriented data model, the parallel architecture and the main-memory design. The data model led to the consideration of two aspects: representation of *object sharing* and the efficient support of *object evolution*. The second feature, the parallel architecture, led to the requirement that the representation should allow coarse-grain parallelism and an easy declustering scheme. The main-memory assumption, finally, led to the requirement that the storage overhead should be low.

On the basis of a qualitative comparison of the three storage models we selected the decomposed storage model (DSM). This method maps the attributes of an object to binary relations. Consequently, join operations are required to retrieve all object's attributes, which is the main reason why it is seldomly used.

It turns out, however, that in a main-memory environment the join overhead is low and will reduce even further with the increase in processor speed. Furthermore, DSM has a low storage overhead compared to the normalized storage model, allows declustering, and provides efficient support of object sharing and object updates. Finally, a data base kernel can be optimized to support only operations on binary relations.

Therefore, our main conclusion is that DSM deserves to be reconsidered as the storage model for parallel main-memory OODBMS.

Two-level storage

Goblin declusters binary relations into fragments and distributes them over the processor pool to exploit parallelism. This means that a query on a binary

relation is mapped to sub-queries on the fragments. To perform this mapping correctly information on the partitioning must be maintained consisting of the partioning method used and the allocation of fragments.

We decided to maintain the fragmentation information for each binary relation in a *summary relation*, because this approach facilitates the use of fragmentation information in the query optimization process. It does not require symbolic evaluation of a query against a *fragmentation rule* to decide whether a fragment accessed by a query. Instead, the query is simply evaluated against the summary database using the a set of relational operations. The effect is that the number of sub-queries to be run on the data base is reduced.

The summary database maintains for each fragment of a binary relation its identification and, in case of range partitioned relations, the minimum and maximum values for each attribute involved. As the partitioning degree is limited by the number of processors in the system is the storage overhead for maintaining the summary database low. Furthermore, the summary query processing overhead can be controlled by varying the partitioning degree.

A disadvantage of this scheme is that to obtain correct answers to queries it is critical that the summary data base is consistent with the data base. A change in the partitioning information of a fragment must be reflected in the summary relation. Fortunately, this overhead is limited and can be avoided for frequently updated fragments by temporarily extending its partitioning information to cover the complete relation domain. In that case the fragment will be accessed by every query. Using this technique the summary relation can be updated by a process running in the background.

Our conclusion is that the summary data base provides a general indexing mechanism which facilitates the use of partitioning information in query optimization. The idea is also applicable to a centralized DBMS where it can limit the number of I/O operations.

12.2.2 Dynamic query processing architecture

Query optimization is a difficult and time consuming process. This is even more so for a parallel DBMS where both data parallelism and pipeline parallelism is exploited. In some systems the optimization process has been split into a logical optimization phase and a parallization phase, which takes the data allocation and load distribution at query start-up time into account. This functional decomposition is based on the assumption that the optimization decisions taken in the two phases are independent. Unfortunately, this assumption does not hold. Furthermore, the effectiveness of pipeline parallelism depends on the load distribution. A change in the load distribution can introduce bottlenecks in the query pipeline.

To overcome these problems we have designed a dynamic query processing architecture which can efficiently take load balancing decisions and perform query optimization at run-time. Its primary characteristics are that it exploits data parallelism only and uses a two-level query processing structure. We decided not to exploit pipeline parallelism, because it is difficult to combine with a load balancing scheme.

In the two-level query processing scheme a query is evaluated first on the summary data base and then on the stored fragments. A consequence of the declustering scheme used in the storage architecture is that a query is mapped into a union query over all possible fragment combinations, or *tasks*. These tasks are independent, run in parallel, and they are small enough to be executed in main-memory.

The two-level architecture is implemented by two kind of processes: *Query Schedulers* (QS) and *Query Processors*(QP). The QS generates tasks for a given query, distributes them dynamically over the processor pool and uses feedback information on the executed tasks to reduce the total amount of work (task elimination) and to adjust the task allocation for load balancing. The QP execute these tasks in main-memory and return execution information to the QS.

The main advantage of this two-level query processing scheme is that the optimization issues are separated. In the QS the *task generation* process drives the query execution by executing the query on the summary data base. Its *task elimination* process performs logical optimization using optimization rules and feed back information to reduce the number of tasks remaining. The *task allocation* process, finally, is mainly concerned with load balancing and reducing the I/O by taking into account the strong relation that exists between task allocation and buffer management in the QP.

The primary concern for the QP is to reduce the average task execution time through an effective use of memory and CPU resources. To achieve this its *task evaluation* algorithm uses a modified version of the INGRESS dynamic query optimization scheme. It is designed to exploit the similarity of tasks by storing and re-using intermediate results and to handle strong fluctuations in the fragment arrival rate.

These four processes, task generation, task elimination, task allocation and task execution have been studied in detail in this thesis. Through mathematical models, simulation and measurements on prototype implementations of these algorithms we have uncovered the critical performance parameters of the two-level query processing scheme, namely partitioning degree and the data model.

It turns out that the minimum response time is obtained for low partitioning degrees (≈ 10). The minimum response time is dominated by the task generation process which has to consider a large number of fragment combinations for execution. The number of produced tasks, and therefore the query cost, depends on the partitioning degree and underlying data model. If the attributes of two summary relations have a $m-n$ relationship, the number of tasks increases significantly. For $1-n$ and $1-1$ relationships the number of tasks can be limited with a proper partitioning.

It is possible to reduce the summary cost by improving the algorithm and by amortizing the cost of summary query processing over multiple executions by storing the summary query result. Therefore, we think that the two-level query processing architecture is a viable approach to the complex query optimization and load balancing task in parallel data base systems.

12.3 Future research

The design of a parallel OODBMS is a multi-year effort. Therefore we have not addressed all the issues in detail. In the previous chapters we have already encountered directions for further research. We will address them in the following sections.

The first direction is the further development of the binary storage architecture. In the discussion data placement and data replication have only been discussed in global terms. The main research question is how to make the data placement and replication adapt to the query workload and available memory resources. A technique to consider is to decluster binary relations on the basis of their partitioning, such that for query processing only a limited number of fragments need to be transported. Another technique worth considering is to maintain transport statistics for each fragment. If it turns out that a fragment is frequently copied to a certain processor site, it could be advantageous to move the fragment permanently to that site.

The second direction concerns the query processing architecture. In particular, optimizations for the summary query algorithm must be considered. Possible approaches are to choose the data partitioning on the basis of the underlying data model. For instance, partition attribute relations of tuple objects on their tuple OID to reduce the number of generated tasks.

The next important issue is to study the effect of the task allocation algorithms on the load distribution. In the beginning of the project simulation models have been constructed to study this aspect, but the experiments failed due to defective simulation software.

Finally, in the task evaluation algorithm the effectiveness and mechanisms for re-using intermediate results must be studied in detail. The main problem here is to guess which intermediate results should be maintained to make an effective use of buffer memory. This problem is related to browsing query optimization, but more restricted, because in this case the query remains the same and only the accessed fragments changes.

Bibliography

[ABD+92] M. Atkinson, F. Bancilhon, D. DeWitt, D. Maier, and S. Zdonik. *Building an Object-Oriented Database System: The story of O_2*, chapter The Object Oriented Data Base System Manifesto, pages 3–18. Data Management Systems. Morgan-Kaufman, 1992.

[AK92] S. Abiteboul and P. Kannelakis. *Building an Object-Oriented Database System: The story of O_2*, chapter Object Identity as a Query Language Primitive, pages 98–127. Data Management Systems. Morgan-Kaufman, 1992.

[AKO88] P.M.G. Apers, M.L. Kersten, and H.C.M. Oerlemans. PRISMA Database Machine: A Distributed, Main-Memory Approach. In *Proceedings of the International Conference on Extending Database Technology*, 1988. Venice, Italy.

[AKW+92] P.M.G. Apers, M.L. Kersten, A. N. Wilschut, P.W.P.J Grefen, C.A. van den Berg, and J. Flokstra. PRISMA/DB: A Parallel Main-Memory Relational DBMS. *IEEE Journal on Data and Knowledge Engineering*, 4(6):541–554, December 1992.

[BCV91] B. Bergsten, M. Couprie, and P. Valduriez. Prototyping DBS3, a shared-memory parallel database system. In *Proceedings of the First International conference on Parallel and Distributed Information Systems*, pages 226–235, 1991. Miami Beach, Florida.

[Bea88] F. Bancilhon and et al. The design and implementation of O_2, an object-oriented database system . In *Advances in Object-Oriented Database Systems*, 1988.

[Bea90] H. Boral and et al. Prototyping Bubba, a highly parallel database system. *IEEE Journal on Data and Knowledge Engineering*, 2(1):4–24, 1990.

[BG89] L. Becker and R.H. Güting. Rule-based optimization and query processing in an extensible geometric database system. Forschungsbericht 312, Fachbereich Informatik Universität Dortmund, Postfach 500500, D-4600 Dortmund, August 1989.

[BK89] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Journal on Data and Knowledge Engineering*, 1(2):196–214, June 1989.

[BR88] P. Bodorik and J.S. Riordon. A threshold mechanism for distributed query processing. In *Proc. of the 16-th Annual ACM Computer Science Conference*, pages 616–625, 1988. Atlanta, GA.

[Bra84] K. Bratbergsengen. Hashing methods and relational operations. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 323–333, August 1984. Singapore.

[Car84] L. Cardelli. *Semantics of datatypes*, volume 173 of *Lecture Notes in Computer Science*, chapter A semantics of Multiple Inheritance. Springer-Verlag, 1984.

[Cat93] R.G.G Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1993.

[CBSB92] C. Chachaty, P. Borla-Salamet, and B. Bergsten. Capturing parallel data processing strategies within a compiled language. *Applied Information Technology*, (13), 1992.

[CDRS86] M. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the Twelfth International Conference on Data Bases*, pages 91–100, August 1986. Kyoto.

[Dea90] O. Deux and et. al. The Story of O_2. *IEEE Journal on Data and Knowledge Engineering*, 2(1):91–108, March 1990.

[DeW91] D.J. DeWitt. chapter The Wisconsin Benchmark: Past, Present, and Future, pages 119–166. Morgan-Kauffman, 1991.

[DGS+90] D. J. DeWitt, S. Ghadeharizadeh, D.A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The GAMMA Database Machine Project. *IEEE Journal on Data and Knowledge Engineering*, 2(1):44–51, 1990.

[GGdM89] G. Gardarin, I. Guessarian, and C. de Maindreville. Translation of logic programs into funtional fixpoint equations. *Theoretical Computer Science*, 63:253–274, 1989.

[Gib91] G.A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage.* An ACM Distinguished Dissertation. The MIT Press, Cambridge Massachusetts, London England, 1991.

[GM84] M. Gondran and M. Minoux. *Graphs and Algorithms.* Interscience Series in Discrete Mathematics. Wiley, 1984.

[Gro87] The Tandem Database Group. *NonStop SQL: A Distributed High-Performance, High-Availability Implementation of SQL*, pages 113–137. Number 359 in Lecture Notes in Computer Science. Springer-Verlag, September 28-30 1987. Pacific Grove, CA.

[Gro90] EDS Database Group. Eds - collaborating for a high-performance parallel relational database. In *Proceedings of the ESPRIT conference*, November 1990. Brussels, Belgium.

[GW89] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proceedings of the 1989 SIGMOD conference*, pages 358–366, 1989.

[HO88] A. Hafez and G. Ozsoyoglu. The partial normalized storage model of nested relations. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 100–111, August 1988. Los Angeles, CA.

[HRD93] I. Herman, G.J. Reynolds, and J. Davy. Made: A multimedia application development environment. CWI Report CS-9369, CWI, 1993.

[HS93] A. Heur and M.H. Scholl, editors. *Fifth Workshop on Foundations of Models and Languages: Optimization in Databases*. Intitut für Informatik, Technische Universität Clausthal, september 1993.

[HZ87] M. Hornick and Z. Zdonik. A Shared Segmented Memory System for an Object Oriented Database. *ACM Transactions on Office Information Systems*, 5(1), January 1987.

[JR86] M.B. Jones and R.F. Rashid. Mach and matchmaker: kernel and language support for object oriented distributed systems. In *International OOPSLA '86 conference*, 1986. Portland, OR.

[KAM+87] M.L. Kersten, P.M.G. Apers, Houtsma M.A.W., E.J.A. van Kuyk, and R.L.W. van de Weg. A Distributed, Main-Memory Database Machine. In *Proc. of the Fifth International Workshop on Database Machines*, pages 353–369, October 1987.

[KBG89] W. Kim, E. Bertino, and J. Garza. Composite objects revisited. *ACM Sigmod Record*, 18(2):337–347, 1989.

[KBGW90] W. Kim, N. Ballou, J.F. Garza, and D. Woelk. Architecture of the ORION next-generation database system. *IEEE Journal on Data and Knowledge Engineering*, 2(1):109–124, March 1990.

[Ker91] M.L. Kersten. Goblin, a DBPL designed for Advanced Database Applications. In *Proceedings of the 2nd International Conference of Database and Expert System Applications*, 1991. Berlin.

[KNT89] M. Kitsuregawa, M. Nakayama, and M. Tagaki. The effect of bucket size tuning in the dynamic hybrid GRACE hash join

method. In *Proceedings of the fifteenth international conference on Very Large Data Bases*, pages 257–267, August 1989. Amsterdam, The Netherlands.

[KvdB91] M.L. Kersten and C.A. van den Berg. Parallel Processing of a Class of Geographical Queries. In *Proceedings of the International Workshop on Database Management Systems for Geographical Applications*, pages 274–287, 1991. Capri, Italy.

[KvdBS+93] M.L. Kersten, C.A. van den Berg, A.P.J.M. Siebes, Thieme C.J., and van der Voort M.H. The Goblin Database Programming Language. Technical Report CS-R9407, CWI, november 1993.

[LC86] H. Lu and M. J. Carey. Load balanced task allocation in locally distributed computer systems. In *Proceedings of the 1986 conference on parallel processing*, pages 1037–1039, 1986.

[LC87] Tobin J. Lehman and Michael J. Carey. A Recovery Algorithm for a High-Performance Memory-Resident Database System. In *Proceedings of the 1987 SIGMOD conference*, 1987.

[Loh89] G.M. Lohman. Is query optimization a 'solved' problem ? Computer Science technical report 89-005, Oregon Graduate Center, Beaverton, OR, 1989.

[LR88] M.D. Palmer Leland and W.D. Roome. The Silicon Database Machine: Rationale, Design, and Results. In *Database Machines and Knowledge Base Machines*, pages 311–324, 1988.

[LSL92] T.J. Lehman, E.J. Shekita, and Cabrera L.F. An Evaluation of Starburst's Memory Resident Storage Component. *IEEE Journal on Data and Knowledge Engineering*, 4(6):555–567, December 1992.

[Mur89] M.C. Murphy. Effective resource utilization for multiprocessor join execution. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 67–75, 1989.

[MvRT+90] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[Ngu81] G.T. Nguyen. Distributed query management for a local network. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, pages 188–196, 1981. Paris, France.

[Oll71] T.W. Olle. Feature Analysis of Generalized Database Management Systems. *Communications of the ACM*, 14(5), May 1971.

[OV92] Tamer M Özsu and Patrick Valduriez. *Principles of Distributed Databases*. Prentice-Hall, 1992.

[Pag92] J. Page. *Proc. of the 10th British National Conference on Databases*, volume 618 of *Lecture Notes in Computer Science*, chapter A study of a Parallel Database Machine and its Performance - The NCR/Teradata DBC/1012, pages 113–137. Springer-Verlag, July 6-8 1992. Aberdeen, Scotland.

[PMC+90] H. Pirahesh, C. Mohan, J. Cheng, T.S. Liu, and P. Selinger. Parallelism in relational database systems: Architectural issues and design approaches. In *Proc. of the 2nd International Symposium on Databases in Parallel and Distributed Systems*, pages 4–29, July 1990. Dublin.

[SAC+79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, May 1979. Boston, Ma.

[SD89] D. A. Schneider and D.J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proc. of the 1989 ACM SIGMOD conference*, pages 110–122, June 1989. Portland, Oregon.

[SKPO88] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. The Design of XPRS. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 318–330, August 1988. Los Angeles, CA.

[SRH90] M. Stonebraker, L. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Journal on Data and Knowledge Engineering*, 2(1):125–142, March 1990.

[ST89] S. Salza and M. Terranova. Evaluating the size of queries on relational databases with non uniform distribution and stochastic dependence. In *Proc. of the 1989 ACM SIGMOD conference*, pages 8–14, 1989. Portland, Oregon.

[Tee93] W.B. Teeuw. *Parallel Management of Complex Objects: The design and implementation of a Complex Object Server for Amoeba.* PhD thesis, Twente University, 1993.

[TF82] T.J. Teorey and J.P. Fry. *Design of Database Structures.* Prentice Hall, 1982.

[TN91] M. Tsangaris and J. Naughton. A stochastical approach for clustering in object bases. *SIGMOD Records*, May 1991.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume II. Computer Science Press, 1989.

[Val87] P. Valduriez. Join indices. *TODS*, 12(2):218–246, 1987.

[VBD89] F. Velez, G. Bernard, and V. Darnis. The O_2 Object Manager: an Overview. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 357–366, August 1989. Amsterdam.

[vdBK90] C.A. van den Berg and M.L. Kersten. *Logging and Recovery in PRISMA*, pages 229–241. Lecture Notes in Computer Science 503. Springer-Verlag, September 1990.

[vdBKSA91] C. A. van den Berg, M.L. Kersten, and S. Shair-Ali. Dynamic parallel query processing. Technical Report CS-R9112, CWI, February 1991.

[vdBvD93] C.A. van den Berg and F. van Dijk. DBO implementation options: data base requirements for Multi-Media support. MADE Report T/DBO-eval/S.2, CWI, 1993.

[vK93] E. van Kuijk. *Semantic Query Optimization in Distributed Database Systems.* PhD thesis, Twente University, 1993.

[VKC86] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation techniques of complex objects. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 101–109, August 1986. Kyoto.

[WA91] A.N. Wilschut and P.M.G Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proceedings of the First International conference on Parallel and Distributed Information Systems*, pages 68–77, 1991. Miami Beach, Florida.

[WFA92] A.N. Wilschut, J. Flokstra, and P.M.G. Apers. Parallelism in a main-memory system: The performance of PRISMA/DB. In *Proceedings of the 18th International Conference on Very Large Data Bases*, 1992. Vancouver, Canada.

[WG89] A. Wilschut and P. Grefen. XRA definition. PRISMA document P465, Twente University, September 1989.

[Wie83] G. Wiederhold. *Database Design.* McGraw-Hill, 1983.

[Wil93] A. Wilschut. *Parallel Query Execution in a Main Memory Database System.* PhD thesis, Twente University, 1993.

[WT90] P. Watson and Townsend. *The EDS Parallel Relational Database System*, pages 149–166. Lecture Notes in Computer Science 503. Springer-Verlag, September 1990. Noordwijk, The Netherlands.

[WY76] E. Wong and K. Youssefi. Decomposition- a strategy for query processing. *ACM Transactions on Data base Systems*, 1(3):223–241, 1976.

Index

Samenvatting

Snel beschikbare en betrouwbare informatie wordt steeds belangrijker binnen het bedrijfsleven en de overheid. Tevens ziet men dan dat de hoeveelheid gegevens die bijgehouden wordt jaarlijks met 25 % groeit. Gegevensbanken vormen een middel om onder deze omstandigheden toch aan de informatiebehoefte te voldoen.

Een belangrijk voordeel van een gegevensbank is dat men door middel van zoekvragen ruwe gegevens kan combineren om zo 'verborgen' informatie af te leiden. Denk bijvoorbeeld aan de bestrijding van uitkeringsfraude door loonbelastinggegevens te koppelen aan uitkeringsgegevens.

Naarmate de zoekvragen ingewikkelder worden en de hoeveelheid gegevens omvangrijker, is het voor een enkele computer niet mogelijk om een zoekvraag binnen een aanvaardbare termijn op te lossen. Met een zogeheten 'parallelle gegevensbank' waarbij verschillende computers samenwerken kan de verwerkingstijd teruggebracht worden. In dit proefschrift wordt een nieuwe techniek onderzocht om de zoekvraag door een aantal computers te laten verwerken.

In de gangbare aanpak wordt de zoekvraag opgedeeld in deelvragen en daarna verdeeld over de beschikbare computers. Het resultaat van een deelvraag wordt doorgestuurd naar een andere computer en gebruikt in de oplossing van zijn deelvraag. Zodoende wordt het resultaat van de zoekvraag als het ware aan een lopende band geconstrueerd. De centrale problemen in deze aanpak zijn de opdeling van de zoekvraag in deelvragen en de toekenning van de totale hoeveelheid werk over de beschikbare computers, zodanig dat de deelvragen binnen dezelfde termijn opgelost worden. Hiervoor moet de duur voor iedere deelvraag bepaald worden en rekening gehouden worden met de belasting van iedere computer. Helaas is het onmogelijk om deze factoren voorafgaand aan de verwerking precies te bepalen, zodat niet volledig gebruik gemaakt wordt van de totale capaciteit van het computersysteem waardoor niet de minimaal mogelijke duur bereikt wordt.

In de voorgestelde aanpak worden de gegevens in stukken verdeeld zodat de originele zoekvraag kan worden opgelost door een groot aantal identieke deelvragen of taken uit te voeren op een gedeelte van de gegevensbank. Een centrale component construeert, coördineert en verdeelt deze taken tijdens de verwerking, rekening houdend met de belasting van iedere computer. Een voordeel van deze aanpak is dat het niet nodig is om voorafgaand aan de verwerking een schatting te maken van de duur van iedere deelvraag, omdat de verdeling van het werk

tijdens de verwerking nog aangepast kan worden. Deze techniek wordt dynamic query processing genoemd.

Echter, het dynamisch aanpassen van de verwerking aan de feitelijke werklast kost extra tijd. In dit proefschrift proberen we voor deze nieuwe verwerkingsmethode inzicht te krijgen in deze kosten. Bovendien worden nieuwe technieken voorgesteld en geanalyseerd die de totale werklast terugbrengen gebruik makend van de informatie in de gegevensbank.

Eén van deze technieken is het zogenaamde "two-level query processing" waarbij de zoekvraag op twee abstractieniveaus wordt verwerkt. De centrale component lost de zoekvraag op over een samenvatting van de opgeslagen gegevens en construeert aan de hand van het resultaat taken die parallel op de werkelijke gegevens worden uitgevoerd. Aan de hand van de informatie op de geabstraheerde gegevens kan de centrale component bepalen welke zoekvragen uiteindelijk kunnen bijdragen aan het resultaat.

Op basis van wiskundige analyses en simulaties van deelaspecten van deze "dynamic query processing"-techniek is een parallel object-georiënteerde gegevensbank ontworpen en geimplementeerd. Uit metingen aan het geimplementeerde systeem blijkt dat, ondanks de tijd die de centrale component nodig heeft voor de generatie van deelvragen, het systeem een zoekvraag binnen eenzelfde tijd kan beantwoorden als traditionele parallelle gegevensbanken. Dit is bemoedigend aangezien er in de implementatie nog vele verbeteringen aan te brengen zijn.

Uit vervolgonderzoek zal de werkelijke kracht van het systeem moeten blijken zodra er met meer ingewikkelde zoekvragen geëxperimenteerd gaat worden. De verwachting is dat traditionele systemen bij het oplossen van deze zoekvragen de beschikbare computers minder effectief kunnen gebruiken.

Curriculum Vitae

Carel van den Berg was born in Bergen, North-Holland, The Netherlands on August 14, 1963. In 1975 he started secondary school at the Murmellius Gymnasium in Alkmaar and passed his final exam gymnasium-β in 1981.

He studied computer science at the University of Amsterdam and got his B.Sc. in 1984, and M.Sc. in 1986 with physics as subsidiary subject. He specialized in computer architecture with image processing as research subject and wrote his M.Sc. thesis on the design and implementation of an efficient image-memory interface for an array processor. "An image processing system using a mesh connected array of binary processors and bit/pixel accessible image memory". His advisors were Prof. Dr. L.O. Hertzberger and Dr. W. Duinker.

After his graduation he joined the data-base research group of Dr. M.L. Kersten at the Centre for Mathematics and Computer Science (CWI) in Amsterdam to work on the PRISMA-project which is a joint research effort of industry (Philips), dutch academia, and the CWI on the design and implementation of a parallel main memory relational database machine using a parallel object oriented language. His tasks were twofold: to participate in the design and implementation of this system and to perform research on parallel data-base machines. This effort resulted in a prototype implementation and a number of international publications. Furthermore, it triggered the conception of the research subject of this thesis: dynamic query processing in a parallel object-oriented data-base system.

In the fall of 1990 he was assigned to the SION project "Starfish" and started his Ph.D. research. In this project the University of Amsterdam, Free University, Twente University and the CWI cooperate to develop applications for a distributed operating system (Amoeba). In this project he concentrated on the design and development of a query processing architecture and data representation for a parallel object-oriented database system.

Currently, he continues his research on dynamic query processing and takes part in the Esprit project "MADE" to investigate data-base support for multimedia applications.