Quality of Service and Optimization in Data Integration Systems

Reinhard Braumandl

DISSERTATION

Accepted by the Department of Mathematics and Informatics of the University of Passau.

First Referee: Professor Alfons Kemper, Ph.D.
Second Referee: Professor Dr. Donald Kossmann

Date of final doctoral examination: 28.02.2002

Acknowledgments

First of all, I have to thank my advisors Prof. Alfons Kemper and Prof. Donald Kossmann for their support. They gave me the opportunity to participate in an ambitious and visionary project. I could learn a lot from their insight and experience in doing research work. Their advices provided invaluable guidance for my work.

I also wish to express my gratitude to all my colleagues at the University of Passau for many helpful discussions and for the pleasant working atmosphere: André Eickler, Jens Claussen, Natalija Krivokapić, Konrad Stocker, Markus Keidl, Stefan Seltzsam, Christian Wiesner, Bernhard Zeller and Bernhard Stegmaier.

André Eickler was the advisor for my master thesis and introduced me to the topic of name services in object-oriented database systems. Together with Jens Claussen the work on functional joins was done and his query processor for the Merlin project influenced my work on the ObjectGlobe query processor. Natalija Krivokapić and I shared an office for several years and we always had a really pleasant working atmosphere.

My doctorate work was done in the context of the ObjectGlobe project and the other project members Konrad Stocker, Stefan Seltzsam, Markus Keidl and Christian Wiesner helped a lot in making this project and thus my work, too, so dynamic. Konrad was our specialist on optimizer technology and always had a lot of hints on work in this area. Stefan developed the security system of ObjectGlobe and was responsible for the perfectly working computer pool. The meta data service of ObjectGlobe was the work of Markus and he also provided the rest of us with the latest KDE versions which he installed in our computer pool. Christian built his Hyperquery system on top of ObjectGlobe and thus had to suffer a little bit from the bugs I had left in the code.

Bernhard Zeller provided all colleagues with a big portion of black humor and Bernhard Stegmaier was my hardware dealer of choice and the first person to ask for all kinds of problems with PC hardware.

Stefan Pröls and Alexander Kreutz did a perfect job in augmenting the ObjectGlobe prototype with all the missing parts so that we could demonstrate the prototype to a broader audience. In the scope of his master thesis, Martin Siller implemented the algorithms for the query optimization techniques which are discussed in this work. Alexandra Schmidt provided support in all kinds of administrative and non-technical tasks and thus helped in concentrating on the 'real' work.
At last, I would like to thank my parents for their great support.

Tittling, June 2002

Reinhard Braumandl

Contents

Chapter 1

Introduction

Over the past years, we have seen a substantial growth of the Internet with respect to the propagation of participating sites and the capacities for data transfer. This development was driven by the desire of private individuals and commercial and non-commercial organizations for a global platform for electronic communication. The applications initially used on the Internet were rather simple and particularly targeted on personal information exchange. As the network capabilities and the understanding of the possibilities a world-wide communication network provides were improving, more advanced applications were developed. Today, we can find, for example, a large number of online shops, multimedia learning courses, distributed scientific data analysis applications and all kinds of information services (e.g., for travel planning) on the Internet. Many of these applications have in common that they offer data and in some cases also limited data processing capabilities.

1.1 Status Quo for Data Processing on the Internet

Altogether, a huge amount of data can be accessed on the Internet. Therefore, many researchers in the field of data integration systems claim that in some sense the Internet can be seen as a global database [LKK+97]. The goal of research in data integration systems is to develop techniques, which allow to use this global database in a similar way as usual database systems. The diversity of data sources on the Internet, which is shown below, causes this task to be a rather difficult one.

The data sources on the Internet belong to many different domains:

- Some well known online stores provide catalogs for books, CDs, DVDs, software, etc.
- Car manufacturers provide information about the car models they are offering.
- Realtors provide descriptions of houses and flats they have under offer.
- Travel agencies inform about possibilities to travel by plane, train and ferry. They also inform about available hotel rooms and rental cars.

- The results produced or gathered, for example, by earth observation, high energy physics or genome research are interesting for a large number of scientists working in the corresponding fields.
- Financial data like exchange or stock rates or economic data are provided by companies which offer online stock trading.

This list could be continued nearly indefinitely. Naturally, each of these domains has its own set of associations and rules which can be expressed in domain specific functions and operations.

We can further differentiate the data sources with respect to the applications which are used to manage and deliver their data. One of the simplest data source would be a WWW-server with static `HTML` pages. A more elaborate data source could deliver `HTML` or `XML` documents which are generated dynamically from data managed by a database system. More powerful data sources could offer a `SOAP` interface to access a server application or even a `JDBC` interface to access a complete database system. Of course, these solutions differ vastly in their performance and the flexibility they offer for retrieving the underlying data.

Two related aspects for Internet data sources are access control and payment. Most data sources on the Internet are still publicly accessible. But there is also a growing number of data sources with a restricted access from the Internet. Obviously, sites which charge for their information services only accept users which have paid for these services. The corresponding data sources normally represent huge values for their providers due to the efforts of creation or maintenance. Information services which are directly attached to the trading systems of stock exchanges are examples for such information services since the delivery of real-time stock rates is a costly task. Other data providers may restrict the access to their data sources to a closed group of users. For example, a company which wants to support the customer relationship management (CRM) systems of its business customers, may provide access to parts of its business data only to these customers.

1.2 A Framework for Query Processing on the Internet

All the Internet data sources from such a diverse set of domains could be used to satisfy various information needs of private users and commercial and non-commercial organizations. Unfortunately, these data sources cannot be used in concert to provide answers to user-defined queries if the usual ways to access them are used. These systems are normally not able to perform queries which try to find correlations between data in different sources. The reason for this is quite simple: The providers of data normally develop their services independently from each other and at the moment there is no agreed upon standard which can be used to perform inter-site query processing. Furthermore, the vastly differing demands and capabilities of data providers, as we have seen above, complicate such a task enormously.

In this work we will concentrate on query processing in a wide area environment such as the Internet. The demand for query processing in the way sketched above has already been identified in the literature before (for example, see [Wie93]). Several solutions were proposed for the architecture of so-called data integration systems which allow database like query processing

on data sources on the Internet. This work introduces a new architecture for data integration systems. This architecture is the foundation of the ObjectGlobe system which is our prototype implementation used for assessing the techniques developed in this work. In contrast to most of the previous architectures, our system utilizes a distributed architecture where distributed service providers can cooperate in the processing of queries. ObjectGlobe allows different service providers to concentrate in the contribution of data, CPU power, or code for query operators. For example, a specific query could use CPU power from provider A, the data from provider B, C, D and E, and a domain specific query operator from provider F.

The resulting data integration system is open with respect to the providers which participate in such a system. This means that new providers can be integrated in our system very easily. Therefore, CPU power and application code can be made available according to the specific requirements and this vastly increases the scalability of such a system.

In the end, this openness of our system can serve as a basis for an information economy where commercial providers offer their services in a free market-economy. In such a market, the customers which pose queries against a data integration system are interested in specifying and controlling the quality of the service they get offered. For example, when providers charge for their services and a query execution requires the services of several providers scattered on the Internet, the users are certainly interested in restricting the cost and time consumption of their queries. Thus, a data integration system has to support the customer in the enforcement of appropriate quality of service (QoS) constraints. The setup of a corresponding QoS management in our data integration system and the necessary techniques which are used to enforce QoS constraints will also be covered in this work.

QoS management in our setting has to trade off performance, costs and result quality aspects of a query execution. Therefore, if the data integration system itself could improve performance by the usage of better algorithms, the task of QoS management would be easier. Here, we especially deal with data sources which only offer access by point queries and thus need to be processed within a query execution by so-called functional joins. Since such data sources appear quite often in data integration and most implementations for functional joins result in time-consuming executions especially in the presence of nested data structures, we introduce a new alogrithm for functional joins which is named $P(PM)^*M$. This algorithm minimizes the number of accesses to a data source and nevertheless retains the nested structure of input data as far as possible.

1.3 The Outline of this Work

Chapter 2 introduces basic techniques and terminology in the areas of data integration systems and query processing. In Chapter 3 the basic ideas of the ObjectGlobe system are explained. We list the main requirements which guided the development of ObjectGlobe and explain how these requirements are met by our architecture. An overview of this architecture is given in the last section of this chapter. Chapter 4 presents the main architectural issues of the ObjectGlobe implementation in detail. This includes the description of special techniques which were developed for the query generation, distribution and execution phases of query processing. Chapter 5

assesses these techniques with respect to their performance effects.

In the Chapters 6, 7 and 8, we concentrate on the QoS techniques developed for the ObjectGlobe system. In Chapter 6 the relevance of QoS aspects in distributed data integration systems is demonstrated. After that, related work in multimedia-, network-, and database literature is listed. Then, we introduce a query processing specific quality of service model and explain the basics of integrating the support for this model in our system. Chapter 7 explains this integration in more detail for the plan generation, instantiation and execution phases. Afterwards, we discuss possible adaptations of query evaluation plans and their application for the enforcement of quality constraints during query execution. The results of experiments in the area of QoS management are summarized in Chapter 8.

In Chapter 9 we show the importance of functional join implementations for data integration systems. The application of functional joins is shown in the context of object-oriented and object-relational database systems. We also list related work regarding advanced implementations of functional joins. In Chapter 10 we give a short overview of physical and logical object identifier (OID) implementation techniques. Then, we list the known algorithms for functional joins and introduce our new $P(PM)^*M$-algorithm. Chapter 11 explains the integration of our algorithm into an iterator-based query engine and gives an initial performance comparison. After that, we present a more comprehensive performance analysis based on a cost model. Chapter 12 provides a conclusion.

Chapter 2

Data Integration Systems

In this chapter, we first examine the conventional way of answering queries with the technology used on the Internet today. With that technology, more advanced queries cannot be evaluated in a practical way. Therefore, data integration systems have been proposed in the literature. Some basic techniques used in such systems are explained below. We do not provide a complete overview of data integration systems here but concentrate on those techniques which are needed to understand the remainder of this work.

2.1 Executing Queries on the Internet

Today, virtually everybody can publish a document by generating `HTML` (or `XML`) and placing it on some Web server; likewise, it is more or less standard to make data stored in relational (or other) databases publicly available on the Web by establishing form-based interfaces and by using CGI scripts or Servlets. WWW clients can retrieve individual documents by a simple "click" and they can get specific information from a database (behind the Web server) by filling out a form. In other words, WWW clients today can easily execute "point queries" (i.e., given URL, return document) and they can execute queries that can be handled by a single database behind a Web server.

Assume that we have such a WWW data source for real estate offers and another for information about airports. A user looks for real estate offers with a distance to an airport less than 10 km. If a WWW client is used for this task, each data source would have to be browsed separately. Furthermore, the search for qualifying pairs of real estate and airport data items would have to be performed manually.

In Figure 2.1 such a scenario is depicted. We assume that the data sources for real estate offers and airports are accessible through HTML forms. In the usual way of answering queries in such a scenario, users have to fill out the corresponding HTML forms of the data sources and find the correlating answers from the independently retrieved results. Therefore, users have to perform the most tedious part for the processing of this query themselves. In essence, a user would have to inspect every element of the cross product of the results of both sources and would have to check if this element qualifies. This operation is called *join* (logically a Cartesian product

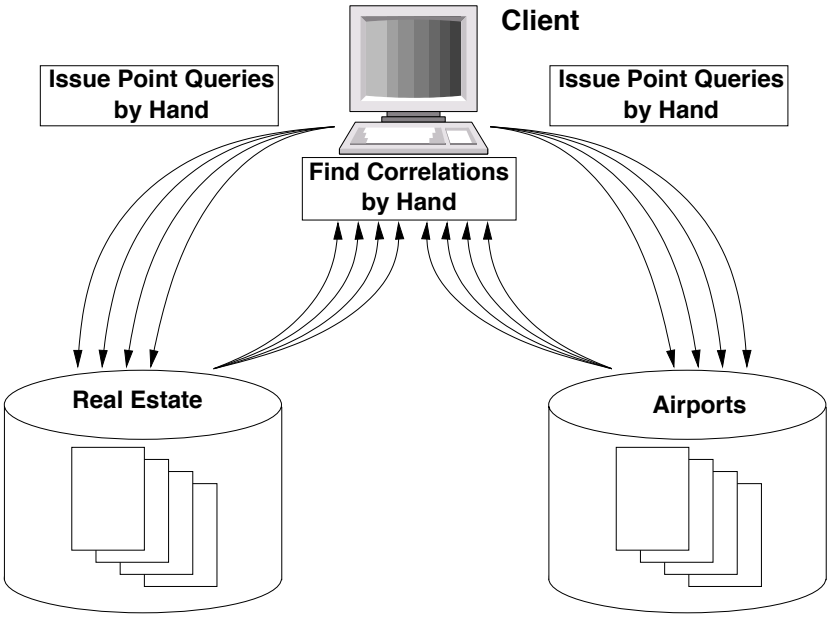


Figure 2.1: Tedious Query Processing by conventional Means on the Internet

followed by a selection) in database literature. In summary we can see that the present techniques to access data which is published on the Internet, are mostly restricted to one data source at a time. However, as seen in database technology, the ability to interrelate the contents of several data sources enhances the expressiveness of queries by far.

If both data sources would have been physically integrated in a (relational) database system, users could specify the requested interrelationship of the data items in a query and the database system would compute the matching pairs of real estate offers and airports automatically. Apparently, such a physical integration is not a viable solution since

- for most of the data providers their data represents a huge value which they would not give away without any copy protection. However, the development of such copy protection techniques is difficult and many solutions in the field of multimedia data have proved to be not effective.

- it is not manageable to incorporate the data of every data provider on the Internet. Therefore, such an integration approach will always be restricted to a rather small subset of the data providers on the Internet.

- the size of the data provided by some data providers is just too large for a practicable data integration. For example, see [BSG00].

Data integration systems, which are introduced in this chapter, can provide nearly the same query processing capabilities as the database solution in such an environment. Data integration systems are often also called middleware systems and we will use both terms to name these

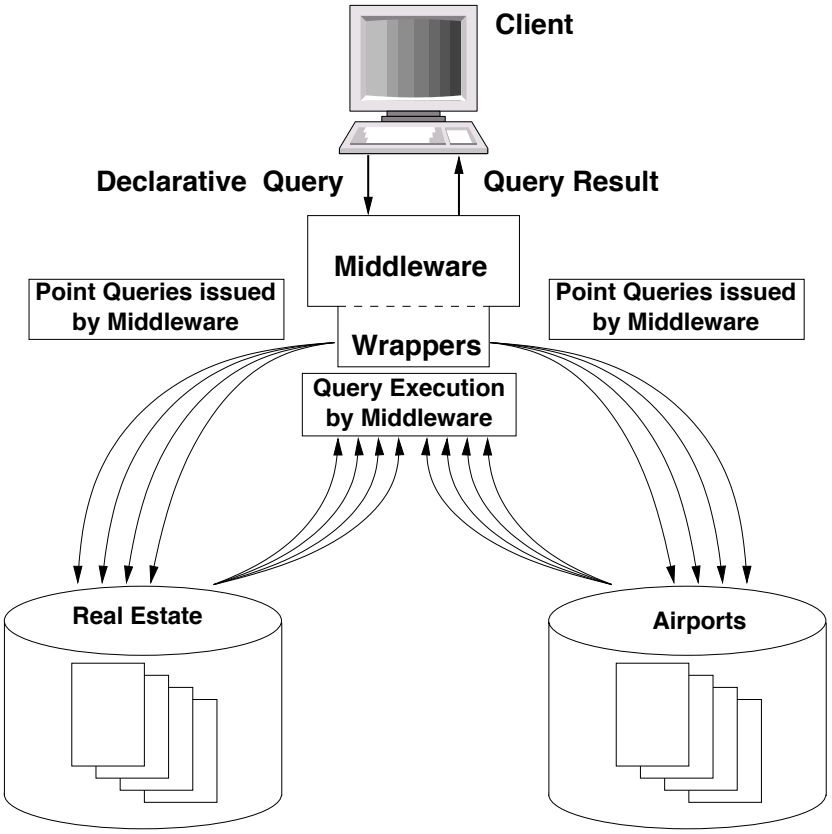


Figure 2.2: Query Execution performed by Middleware Layer

systems[1]. The term middleware system results from the usage of an intermediate layer, called mediator or middleware layer, which mediates between the information demands of a user and the integrated data sources.

2.2 The Middleware Layer

A data integration system tries to unify the access to the integrated data sources by the use of a middleware layer. The middleware layer translates a user-defined query into specific sub-queries for the affected data sources and performs the remaining tasks to complete the execution of the query itself. The basic architecture for middleware systems was proposed in [Wie93]. Figure 2.2 shows the interactions during a query execution performed by a middleware system. The client machine just has to send the specification of a query to the middleware system and gets the corresponding answer after the middleware server has executed the query. As we can see in the figure, queries are executed by the middleware itself and by so-called wrappers which are used

[1]We do not deal with other middleware techniques like CORBA or J2EE here, so there should not emerge misunderstandings about this term in the remainder of this work.

to integrate external data sources into the middleware system.

2.2.1 Query Processing

One of the major ideas in the development of data integration systems was, to add the query processing power which is missing at the data sources and which is necessary to perform the integration itself, in the middleware layer. Several alternatives for the architecture of such middleware systems has been proposed and these approaches mostly differ in the expressiveness of the queries which can be processed and in the kind of data which can be integrated. Two major types of architectures are those based on information retrieval systems and database systems. In this work we will concentrate on data integration systems which are based on database technology. In the following, the basic architectural components of these data integration systems are introduced.

Data Model

Obviously, the integrated data from external sources need a representation which can be used by the query processing operations in the middleware layer. The data model which defines these representations, has an impact on the applicability of specific query processing operations and on the ability of the middleware system to integrate specific external data sources.

The data model determines a basic set of meaningful operations which can be applied in query processing. Therefore, the query language for specifying declarative queries and the corresponding basic set of operations which can be used for query processing within the middleware layer depend on the data model. For example, if the middleware layer uses a nested-relational data model, SQL-92 as query language and the query processing operations in the relational algebra are not sufficient, since non-atomic attributes could not be processed[2].

Naturally the data model of the middleware layer also restricts the set of data sources which can be integrated in a reasonable manner. For example, an XML encoded text of Shakespeare's 'Romeo and Juliet' with annotated stage directions can hardly be integrated reasonably in a middleware system with a relational data model. However, it is also clear that a pure relational data model is not sufficient since many data sources use more complex, especially nested structures to represent their data. Therefore, nested relational or object-oriented data models are more common in data integration systems. In this work, we concentrate on middleware systems with a nested-relational data model. ObjectGlobe has a nested-relational data model with the following properties:

- The basic components of the data model are sets of records. In accordance with database terminology, we also use the term relation for such a set and the term tuple for a record. All the records of a set are structured according to a specific record type. Thus, this record type is a property of the corresponding set. Curly brackets are used in type expressions to denote a set type.

[2]See [KM94] for a discussion of relational, nested-relational and object-oriented data models.

- Analogously to programming languages, record types consist of a fixed and ordered set of attribute types. For each attribute type a data field exists in each record of the corresponding record type. Square brackets are used in type expressions to denote a record type.
- An attribute type can be one of the following:
  - An atomic type like `Boolean`, `Integer`, `Float` or `String`.
  - A set of an atomic type like {`Integer`} or {`String`}.
  - A record type like
    `[AttrName1: Integer, AttrName2: String]` or
    `[AttrName1: {String}, AttrName2: Integer]`.
  - A set of a record type like
    {`[AttrName1: Integer, AttrName2: String]`} or
    {`[AttrName1: {String}, AttrName2: Integer]`}.

An example type for a relation `rel` is then given by the following expression:

```
rel: {[AttrName1: {[nestedAttrName1: Integer,
                    nestedAttrName2: String]},
       AttrName2: String, AttrName3: {Integer}]}
```

The presented data model is expressive enough to allow the integration of a multitude of data sources with structured or semi-structured data. Structured data is normally provided by server applications like relational or object-oriented database systems. Semi-structured data like `HTML` or `XML` documents may be provided, for example, by applications which are backed by a database system, like a great deal of WWW applications today, or by desktop applications with XML-based data formats.

Query Optimization

Most database systems provide a logical view expressed in the corresponding data model on the physical representation of stored data. The internal, physical data representation can be varied without breaking the logical view. Normal users and database applications work on the logical view and thus, will not be affected by changes in the physical representation of the data. For query processing, these database systems leverage query languages which allow to specify the result of a query in a *declarative* manner based on this logical view. As shown at the left side of Figure 2.3, a declarative query is translated in a plan generation step into a *query evaluation plan* (also called query execution plan and short QEP) which contains a procedural description of the query execution. This means, that the declarative query determines the properties which should be fulfilled by the result of the query, and the query evaluation plan determines how this result can be computed. This procedural description is still based on the logical view on the data and thus we call it a logical QEP. The fundamental work for this architectural aspects of query processing in database systems has been reported in [SAC+79].

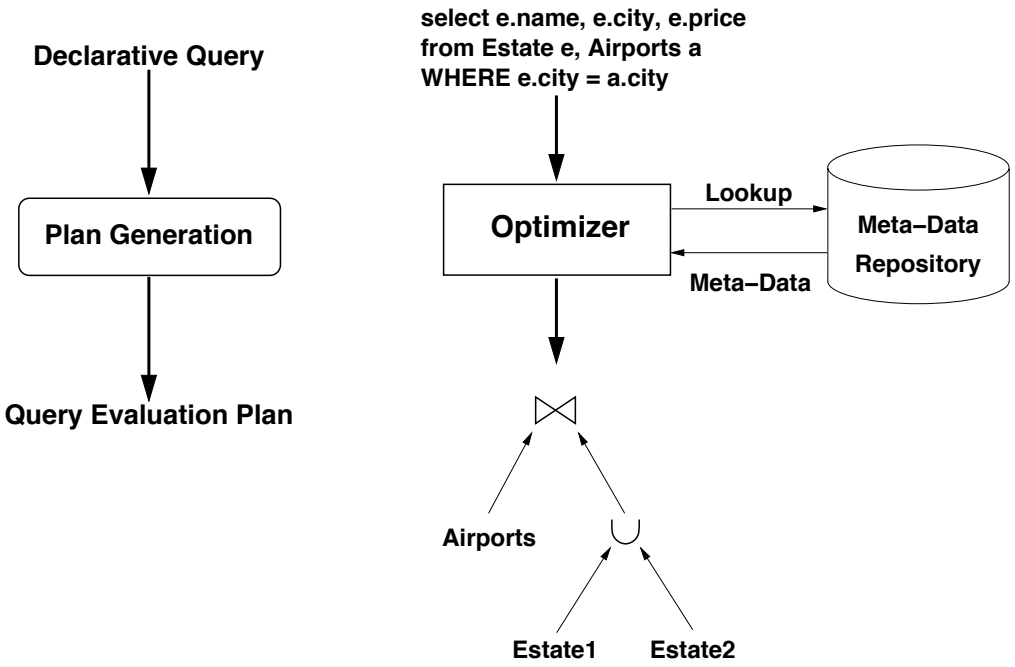


Figure 2.3: Compiling a Query Evaluation Plan from a declarative Query

Query Evaluation Plans As shown at the right side of Figure 2.3, the component which is used to generate a QEP is normally called *optimizer*. The input in the figure is a declarative query expressed in a specific *query language* which is in this case SQL [ANS92] and the result is a term expressed in a *query algebra* which is in this case the relational algebra [Cod70]. The term in relational algebra as shown in the figure is based on the logical view on the data. The physical QEP can be obtained by replacing the logical operators in a logical QEP by appropriate operator implementations which work on the physical data representation.

The effort for executing a QEP is estimated during optimization by a so called *cost model*. Depending on the application area this effort can be the work induced by the execution or the response time of the execution or some other, application-specific measure. In general, we talk abstractly of the costs of a query execution when we refer to the measure produced by a cost model. A cost model is used during optimization to select a minimum cost QEP from several equivalent alternatives.

Table 2.1 shows the operators of the query algebra used in our ObjectGlobe system. The first seven operators are just the operators of the relational algebra. The remaining ones are needed to support nested structures in tuples. Big characters denote relations or intermediate results produced by terms of the algebra. Small characters without an index denote tuples of sets and those with indexes represent attributes of tuples. An exception from this rule appears in the description of the *map* operator where the f_i variables denote functions which are applied by the map operator on the tuples[3].

[3]External parameters for the functions are not shown in the description of the map operator.

Name	Symbol	Description
set minus	$-$	$A - B := \{a \mid a \in A \wedge \nexists b \in B : b = a\}$
union	$\cup$	$A \cup B := \{c \mid c \in A \vee c \in B\}$
intersection	$\cap$	$A \cap B := \{c \mid c \in A \wedge c \in B\}$
cross product	$\times$	$\{[a_1, \ldots, a_n, b_1, \ldots, b_m] \mid [a_1, \ldots, a_n] \in A \wedge [b_1, \ldots, b_m] \in B\}$
selection	σ	$\sigma_p(A) := \{a \mid a \in A \wedge a$ fulfills predicate $p\}$
projection	Π	$\Pi_{i_1,\ldots,i_n}(A) := \{[a_{i_1}, \ldots, a_{i_n}] \mid [a_1, \ldots, a_n] \in A\}$, constraint: $\{i_1, \ldots, i_n\} \subseteq \{1, \ldots, n\}$
rename	ρ	$\rho_{a \to b}(A)$ renames attribute a in the schema of A into b
join	$\bowtie$	$A \bowtie_p B := \sigma_p(A \times B)$
map	χ	$\chi_{f_1,\ldots,f_n}(A) := \{[f_1([a_1, \ldots, a_m]), \ldots, f_n([a_1, \ldots, a_m])] \mid [a_1, \ldots, a_m] \in A\}$
unnest	μ	$\mu_{a_k}(A) := \{[a_1, \ldots, a_{k-1}, a_{k_1}, \ldots, a_{k_n}, a_{k+1}, \ldots, a_m] \mid [a_1, \ldots, a_m] \in A \wedge [a_{k_1}, \ldots, a_{k_n}] \in a_k\}$, constraint: attribute a_k is set-valued.
nest	ν	$\nu_{(a_{j_1},\ldots,a_{j_l}):a_r}(A) := \{[a_{i_1}, \ldots, a_{i_n}, a_r] \mid (\exists [b_1, \ldots, b_m] \in A : (\forall t \in \{i_1, \ldots, i_n\} : b_t = a_t)) \wedge (\forall [b_1, \ldots, b_m] \in A : (\forall t \in \{i_1, \ldots, i_n\} : b_t = a_t) \Rightarrow [b_{j_1}, \ldots, b_{j_l}] \in a_r) \wedge \{j_1, \ldots, j_l\} = \{1, \ldots, m\} \backslash \{i_1, \ldots, i_n\}\}$

Table 2.1: The Operators of the Query Algebra.

Some algebraic properties of operators like associativity and commutativity lead to different logically equivalent QEPs for a single query. For example, since the *join* operator is associative the expressions $A \bowtie (B \bowtie C)$ and $(A \bowtie B) \bowtie C$ are logically equivalent. However, depending on the data size of the relations A, B, and C and the size of the intermediate result of the first executed join operation the join order can influence the computational costs of the query execution significantly. An overview of such algebraic transformations on the relational algebra can be found in [KE99].

Furthermore, some logical operators can be implemented in different ways and depending on the situation these implementations can also differ in their computational costs vastly. For example, the nested-block join implementation is effective if one of its operands fits into main memory, but if the operands are rather large and both are of equal size, the sort-merge join is a better choice. The selection of an access path for base relations is strongly related with the appropriate selection of a physical operator. In database systems base relations can also be accessed by persistent index structures if such structures were created by the database administrator. Therefore, a QEP could use a a simple *TableScan* operator to access the tuples of the relation one after the other or an *IndexScan* could be used which provides an associative access to the tuples. Depending on the number of necessary accesses to the index structure, a *TableScan* can also be better suited than an *IndexScan*.

The query execution costs can differ by orders of magnitudes for different configurations of a QEP [IK91]. Thus, alternative physical QEPs should be regarded during the translation of a declarative query. Discrete optimization algorithms are normally used to search for an acceptable QEP. Many optimization algorithms has been studied in the literature [Ste95]. These algorithms differ in the set of alternative QEPs which are regarded during optimization (this set is also called search space) and the way they enumerate alternative QEPs. Two optimization algorithms which are very common in this context are introduced in the following.

Dynamic Programming Optimization Dynamic programming based query optimization has been introduced in a foundational work on query optimization [SAC$^+$79]. Originally, join order optimization and access path selection were the main tasks of these optimizers. Today, several sophisticated optimizer frameworks have extended the original techniques to handle external functions, new operators and external data [Loh88, HFLP89, PH92, HKWY97]. In general, dynamic programming based optimization has proved to be a versatile and stable optimization technique in database systems. Thus, such an optimizer can be found in nearly every commercial database system.

The main idea of dynmanic programming is to solve each sub-problem of the given optimization problem and to use the resulting solutions to construct a solution for the next, more complex problem. This procedure is iterated until the solution for the overall problem has been constructed. An example optimization run of dynamic programming for join order optimization is depicted in Figure 2.4. Here, a QEP for a three-way join between the relations *A, B* and *C* should be constructed and the necessary steps are explained below.

1. The optimizer considers all access paths for the base relations and selects for each relation the access path with the least costs.

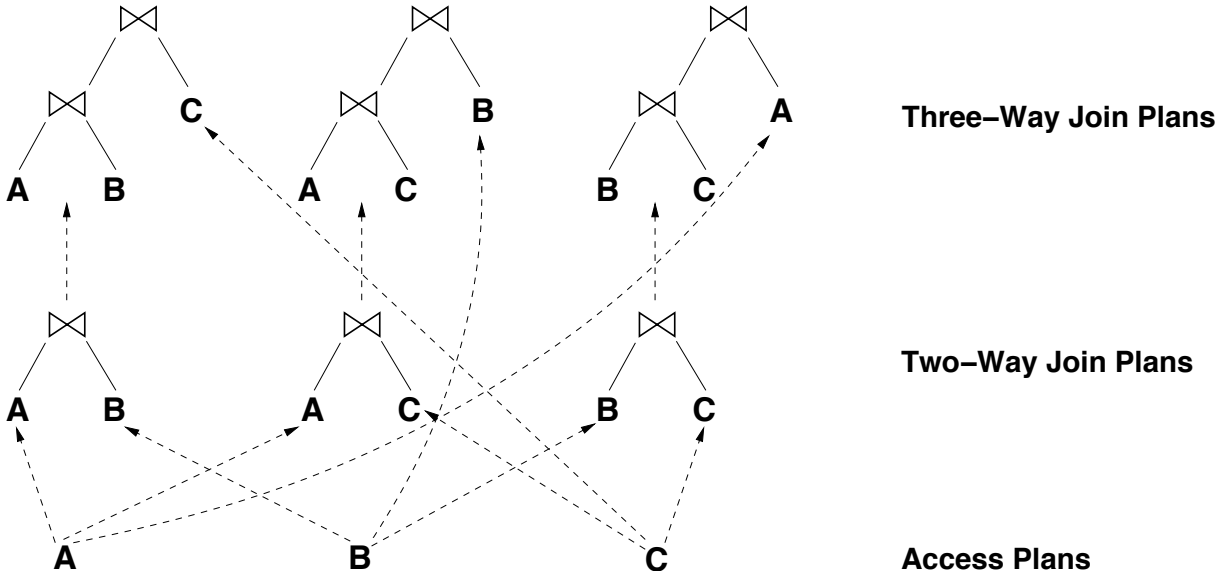


Figure 2.4: Join Order Optimization with Dynamic Programming.

2. Based on these access paths, all possible two-way join plans between the relations which are mentioned in the query are constructed.

3. Based on these two-way join plans, all three-way join plans for the relations *A, B* and *C* are constructed.

The arrows in Figure 2.4 show where the solutions for sub-problems of the given join order problem are used in the construction of a more complex problem. For example, the solution for the problem $A \bowtie B$ is used for the solution $(A \bowtie B) \bowtie C$ of the overall problem. All the three-way join plans in the figure are logically equivalent and represent solutions for the overall problem. The next step in the algorithm would be to prune those solutions which are inferior regarding the costs which are calculated by the cost model for each solution. The remaining three-way join plan is then used as the solution for the join between three relations *A, B* and *C*. This pruning step also occurs for every sub-problem which is tackled during the optimization. Thus, if we consider different join implementations for the $A \bowtie B$ problem, several solutions exist for that problem and the one with the least costs is chosen and will be considered in the solution of a more complex problem, where $A \bowtie B$ represents a sub-problem.

A dynamic programming optimization algorithm computes the optimum solution if Bellman's principle of optimality is fulfilled by the underlying problem [Bel52]. In essence, this principle says that an optimum solution can only be constructed out of the optimum solutions of its sub-problems. For example, the optimum solution for $(A \bowtie B) \bowtie C$ can only be constructed with the optimum solutions for $A \bowtie B$ and C. Unfortunately, this principle is not fulfilled in the query optimization context. Thus, dynamic programming is just used as heuristics and will in general not compute the optimum solution. Therefore, an often used enhancement is the usage of the $n(n \geq 2)$ best solutions of every considered sub-problem. Another enhancement is the

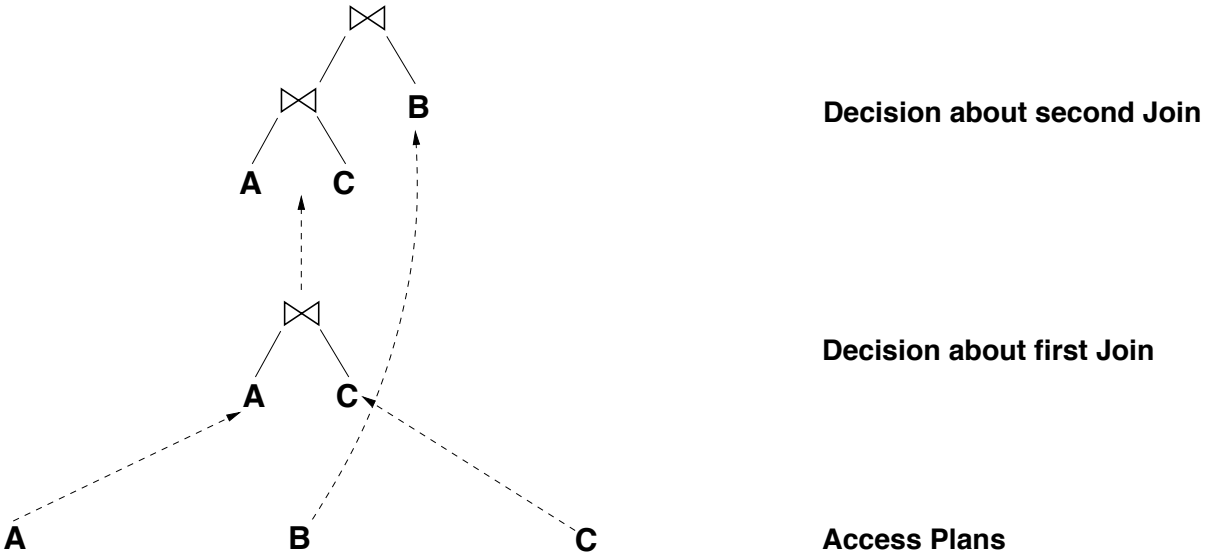


Figure 2.5: Join Order Optimization with a Greedy Algorithm.

introduction of plan properties (such as 'attribute a_1 is sorted in ascending order in the result of this plan') additional to the cost property. This results in a multi-dimensional comparison between plans and consequently in incomparable plans. All incomparable plans for a sub-problem will be regarded as a solution in the construction of a more complex problem. The mentioned enhancements increase the space and time complexity of the optimizaton algorithm. But these effects are normally accepted since these enhancements normally help to produce much better solutions.

Since dynamic programming is used in many variations for query optimization, we will not go into depth here and just leave it at the informal description from above. Detailed descriptions of dynamic programming based query optimization can be found in [VM96, KS00, GLSW94].

Greedy Optimization The complexity of dynamic programming based optimization is exponential in the number of relations which are used in the query. Therefore, the optimization time for queries which use a large number of relations is not acceptable when dynamic programming is used. The greedy algorithms which were proposed for query optimization (for example [OL90]) show a polynomial complexity in the number of relations. Therefore, they are much better suited for queries with a large number of relations. Of course, the reduced complexity results in QEPs which are often inferior to those which are found by a dynamic programming optimizer. Hence, some systems use a two stage configuration where dynamic programming is used for queries which use less relations than a given bound, and a greedy algorithm for all other queries.

Figure 2.5 shows the steps of a greedy algorithm for the optimization of the query which was also used in the dynamic programming case. Obviously, the greedy algorithm generates a much lower number of partial solutions than the dynamic programming algorithm. Just as the name of the algorithm suggests, the greedy algorithm makes a sequence of local decisions which are fixed

until the algorithm finishes. The steps of the greedy algorithm considered here, are summarized in the following:

1. Analogously to the dynamic programming case, the access paths for base relations are selected.

2. Those two relations are selected which can be joined at the cheapest costs. The resulting plan will be augmented step by step in the next phase until a complete plan has been produced.

3. In each step of this phase, that relation is selected which can be joined with the currently constructed plan at the cheapest costs. The resulting plan is further augmented in the next step until all relations have been used in the plan.

2.2.2 The Role of Wrappers

The heterogeneity of access techniques and formats which are inevitable for external data sources are a great problem for data integration systems. Complex data and query processing tasks on a set of such diverse data sources would be extremely tedious if these tasks could not be restricted to the access technique and the data format of the middleware layer. In this way, providers of new functions or operators would just have to deal with the requirements of the middleware layer and their code could be applied no matter which data sources are used. Therefore, in [Wie93] the concept of a wrapper is introduced. A wrapper transforms the specific access technique and data format of a data source in the corresponding, general concepts of the middleware layer. Of course, specific data source configurations regarding access technique and format also need specific wrappers which are specialized on this transformation step. An architectural overview of such a transformation process performed by a wrapper is depicted in Figure 2.6. In the data source layer of the figure some examples for access techniques and formats of data sources are given. A very common configuration at the moment would be that a data source is accessed through the `HTTP` protocol and the data is embedded in `HTML` text, for example in the form of a table. However, it is generally preferable to access the data in a format which more closely resembles its representation in the underlying management system. Access to data in this way is normally provided with application specific proctols. These protocols based on, e.g., `RMI` or `SOAP` and standardized access protocols like `JDBC` or `Z39.50` offer a much faster and more reliable access to data sources than WWW-based applications built on top of `http` and `HTML` techniques.

In summary, the separation of physical and logical properties of data and the mechanisms for supporting declarative query languages are prominent features of database systems and also help in the construction of data integration systems. External data sources can be seamlessly integrated in the logical view on data regardless of the way they are accessed physically. In a QEP the logical reference to a relation which represents an external data source is just replaced by the wrapper operator which is used to transform its external data representation into the format of the middleware[4].

[4]We do not consider here data sources which require external bindings for attributes. For details on this topic see

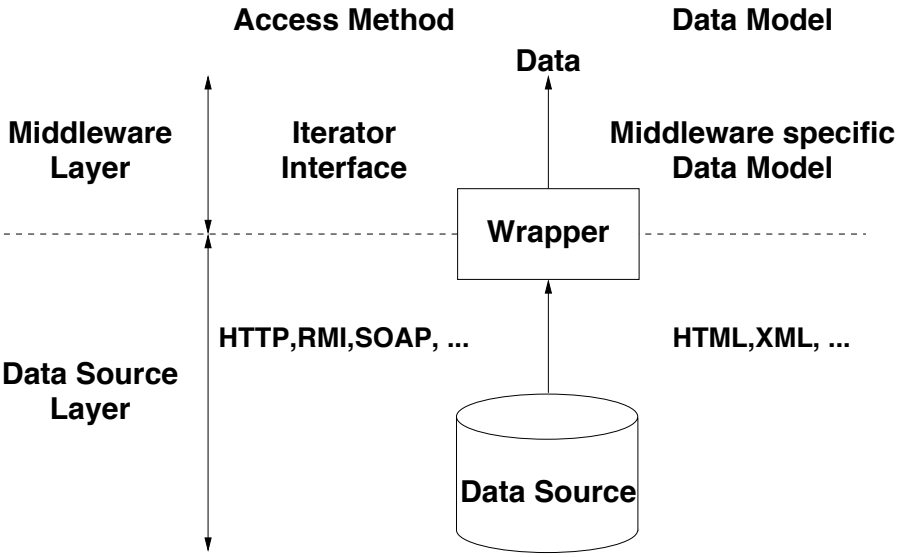


Figure 2.6: The Integration of Data Source in a Middleware Layer by a Wrapper.

[FLMS99]. The solution proposed in [FLMS99] has also been adopted in our system.

Chapter 3

The Basic Ideas of the ObjectGlobe System

Similar to the ubiquitous WWW infrastructure, a unique data integration system which can potentially cover all the appropriate data sources on the Internet would increase the usefulness and usability of such a system enormously. At the beginning of this chapter we identify the requirements for such a globally operating data integration system and from these we infer some basic system properties. Afterwards, an overview of the ObjectGlobe architecture is given and the differences to existing architectures for data integration systems are explained.

3.1 A new Architecture

Our objective is to allow the use of a data integration system in an Internet-wide scale. This entails several challenges with respect to its architecture. For instance, in a data integration system which is run by a single company, software distribution and security concerns could be solved by a central or coordinated administration of the information technology infrastructure. In an Internet scenario, such a form of administration is just not conceivable due to scalability problems. Furthermore, the providers found on the Internet are mostly independent from each other, and also the users normally do not have tight relations to providers. This means, that different requirements of users and providers have to be considered with a higher precedence than in a tighter coupled system. The different requirements of the respective parties are presented in the following section.

3.1.1 The Requirements

The differing demands of *data providers* and users with respect to a global data integration system show why current architectures for distributed databases ([CDF+94]) and mediator systems ([HKWY97, PGGMU95, JKR99]) are not sufficient. Data providers are interested in

- the **security** of their computers. Thus, some data providers with higher security demands would not be willing to execute mobile code in order to avoid the danger of a hostile system intrusion.

- the **privacy** of their data. Data providers could be interested in restricting and controlling access to their data by the use of *authorization* and *authentication* techniques. Furthermore, they may demand the use of cryptography to avoid that somebody can steal their data during network transmissions.

- the **scalability** of the system. The number of users in a global system could cause overload situations on data providers. Therefore, data providers may allow no other operation to be performed on their machines than a simple scan/index-scan on the data.

Naturally, users have completely different requirements which also seem to partly contradict the requirements of data providers. Users are interested in

- an **open** system, where service providers can be integrated and spontaneously be used in queries. As a consequence, there is no need to build several special-purpose data integration systems and a user just has to work with *one* dynamically extensible system.

- an automatic **service composition**. Users want to state a declarative query and the composition of appropriate services in the form of a *query evaluation plan* (QEP) should be performed by a *query optimizer*.

- an **extensible** system in which user-defined code can be integrated in a seamless and rather effortless manner. Especially in distributed and heterogeneous systems this is an important issue. In such systems, it is essential to be able to apply data transformations or user-defined predicates early (i.e., close to the data providers) in order to unify data representations or to reduce the data volume.

- a **quality-of-service (QoS) aware** system. Query execution in a widely distributed system can hardly be monitored by users. Therefore, they should be able to specify quality constraints on the result and the properties of the query execution itself (e.g., time and cost consumption) and the system should fulfill these constraints if possible or abort the execution of the query as early as possible. [Wei98] gives a comprehensive motivation for the need to integrate the handling of QoS guarantees in information systems.

3.1.2 A Possible Solution

In our ObjectGlobe project we have developed a distributed data integration system which works along the lines stated above. In order to help both the data providers and the users, we introduced the new services of *cycle* and *function providers*.

- Function providers offer Java byte-code in different standardized forms (query operators, predicate functions, data transformers, etc.) which are suited for the execution by a cycle provider. For example, a function provider can offer wrappers for accessing data providers, predicate functions specialized on business areas like real estate data or new query operators like join methods for spatial data.

- A cycle provider runs our Java-based query processing engine. It represents a node in our distributed data integration system which can execute plan fragments of a distributed query evaluation plan if the data providers are not willing or not suited due to their hardware capacities or their position in the network to do so. They provide a core functionality for processing queries but can also load new functionality from function providers, for example, a wrapper for accessing a data provider. A specialized Java 'sandbox' is used to secure the cycle provider's machine against malicious effects of external code.

A distributed lookup service is used for registering and querying meta-data about all known instances of services described above. Providers which want to offer one or a mix of the mentioned services can register the description of their services with the lookup service. This description is sufficient to integrate these services in the global system which can therefore be called a truly *open* system.

The service descriptions can also contain authorization data for providers. In that way, data providers can express *privacy* policies for the usage of their services. These policies are enforced either by the ObjectGlobe system or by the providers themselves. In the latter case the providers have to use the authentication services of the ObjectGlobe system.

The ObjectGlobe query planner (later on also called query optimizer) performs a *service composition* on a logical level. Based on the formulation of a specific query, the query planner uses the lookup service to retrieve meta-data about relevant services. The query planner uses this meta-data to produce a query evaluation plan, which takes into account the requirements of the selected services (authorization, authentication, ...) and the query itself (specification of the query answer, QoS requirements). The generated query evaluation plan determines what services have to be used for answering the query and how these services have to be used.

The query evaluation plan is some sort of programming code for the ObjectGlobe server process which acts as an interpreter in this respect. Such a server has to be used by every cycle provider, and in some cases will also be used by data providers. It includes a query processor which is able to integrate mobile code from function providers into the processing steps of a query. This feature provides the *extensibility* which is required by users of such a system. The security problems which are introduced by the execution of mobile code from third parties are met by an adapted *security* system of the Java runtime system. The programming language Java is used as the implementation basis of the ObjectGlobe system and the mobile code of function providers. In this way, the *portability* of the system and a smooth exchange of mobile code also across different system architectures is guaranteed.

The idea to create an *open* market place for data providers, function providers, and cycle providers where appropriate services can be selected for query processing, helps also to ensure the *scalability* of this approach. For example, data providers which do not want to also act as cycle providers perhaps due to load or security concerns, can offload processing tasks to cycle providers. Cycle providers in turn do not want to administer all the code of special functionality which should be executable on their machines. This task is performed by function providers.

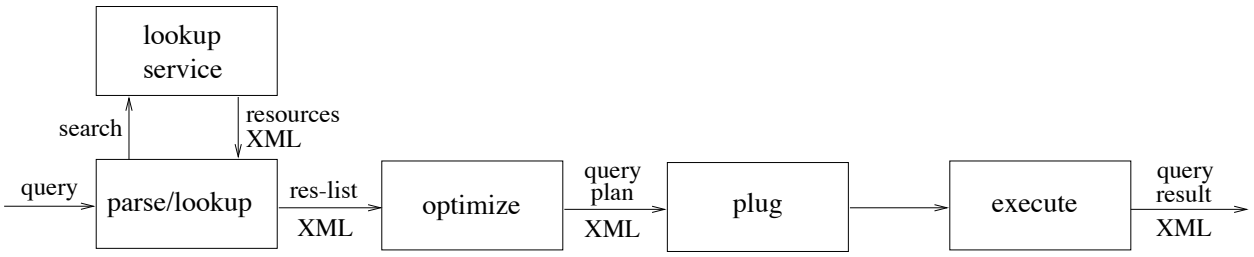


Figure 3.1: Processing a Query in ObjectGlobe

3.2 Overview of the ObjectGlobe System

The goal of the ObjectGlobe project is to distribute powerful query processing capabilities (including those found in traditional database systems) across the Internet. The idea is to create an open market place for three kinds of suppliers: *data providers* supply data, *function providers* offer query operators to process the data, and *cycle providers* are contracted to execute query operators. Of course, a single site (even a single machine) can comprise all three services, i.e., act as data-, function-, and cycle-provider. In fact, we expect that most data and function providers will also act as cycle providers. ObjectGlobe enables applications to execute complex queries which involve the execution of operators from multiple function providers at different sites (cycle providers) and the retrieval of data and documents from multiple data sources. In this section, we will outline how such queries are processed, give an example, and discuss the security requirements of the system.

3.2.1 Query Processing in ObjectGlobe

Processing a query in ObjectGlobe involves four major steps (Figure 3.1):

1. Lookup: In this phase, the ObjectGlobe lookup service is queried to find relevant data sources, cycle providers, and query operators that might be useful to execute the query. In addition, the lookup service provides the authorization data—mirrored and integrated from the individual providers—to determine what resources may be accessed by the user who initiates the query and what other restrictions apply for processing the query.

2. Optimize: The information obtained from the lookup service, is used by a quality-aware query optimizer to compile a valid (as far as user privileges are concerned) query execution plan, which is believed to fulfill the users' quality constraints. This plan is annotated with site information indicating on which cycle provider each operator is executed and from which function provider the external query operators involved in the plan are loaded.

3. Plug: The generated plan is distributed to the cycle providers and the external query operators are loaded and instantiated at each cycle provider. Furthermore, the communication paths (i.e., sockets) are established.

4. Execute: The plan is executed following an iterator model [Gra93]. In addition to the *external* query operators provided by function providers, ObjectGlobe has *built-in* query operators for selection, projection, join, union, nesting, unnesting, and sending and receiving data. If necessary, communication is encrypted and authenticated. Furthermore, the execution of the plan is monitored in order to detect failures, look for alternatives, and possibly halt the execution of a plan.

The whole system is written in Java for two reasons. First, Java is *portable* so that ObjectGlobe can be installed with very little effort; in particular, cycle providers which need to install the ObjectGlobe core functionality can very easily *join* an ObjectGlobe system. The only requirement is that a site runs the ObjectGlobe server on a Java virtual machine. Second, Java provides secure extensibility. Although many people complain about the execution speed of Java programs, we noticed that by avoiding some pitfalls in the Java I/O library the execution speed of the Java virtual machine is no bottleneck in wide area distributed systems. Like ObjectGlobe itself, external query operators are written in Java: they are loaded on demand (from function providers), and they are executed at cycle providers in their own Java "sandbox" (more details in Section 4). Just like data and cycle providers, function providers and their external query operators must be registered in the lookup service before they can be used.

ObjectGlobe supports a nested relational data model; this way, relational, object-relational, and `XML` data sources can easily be integrated. Other data formats (e.g., `HTML`), however, can be integrated by the use of wrappers that transform the data into the required nested relational format; wrappers are treated by the system as external query operators. As shown in Figure 3.1, `XML` is used as a data exchange format between the individual ObjectGlobe components. Part of the ObjectGlobe philosophy is that the individual ObjectGlobe components can be used separately; `XML` is used so that the output of every component can be easily visualized and modified. For example, users can browse through the lookup service in order to find interesting functions which they might want to use in the query. Furthermore, a user can look at and change the plan generated by the optimizer.

3.2.2 Example Plans

To illustrate query processing in ObjectGlobe, let us consider the example shown in Figure 3.2—the corresponding query plan is sketched in Figure 3.3. The real `XML` plan is given in Appendix A. In this example, there are two data providers, A and B, and one function provider. We assume that the data providers also operate as cycle providers so that the ObjectGlobe system is installed on the machines of A and B. Furthermore, the client can act as a cycle provider in this example. Data provider A supplies two data collections, a relational table R and some other collection S which needs to be transformed (i.e., wrapped) for query processing. Data provider B has a (nested) relational table T. The function provider supplies two relevant query operators: a wrapper (*wrap_S*) to transform S into nested relational format and a compression algorithm (*thumbnail*) to apply on an image attribute of T.

Figure 3.3 shows the most important annotations—in particular, the *cycle-provider*, *partition*, and *codebase* annotations—of the query plan. The *cycle-provider* annotation of an operator

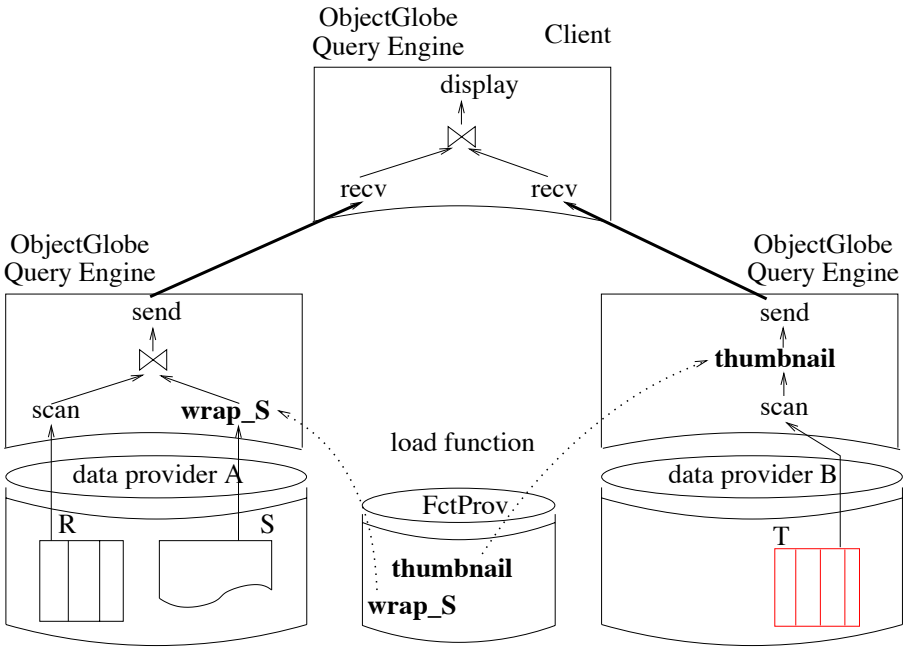


Figure 3.2: Distributed Query Processing with ObjectGlobe

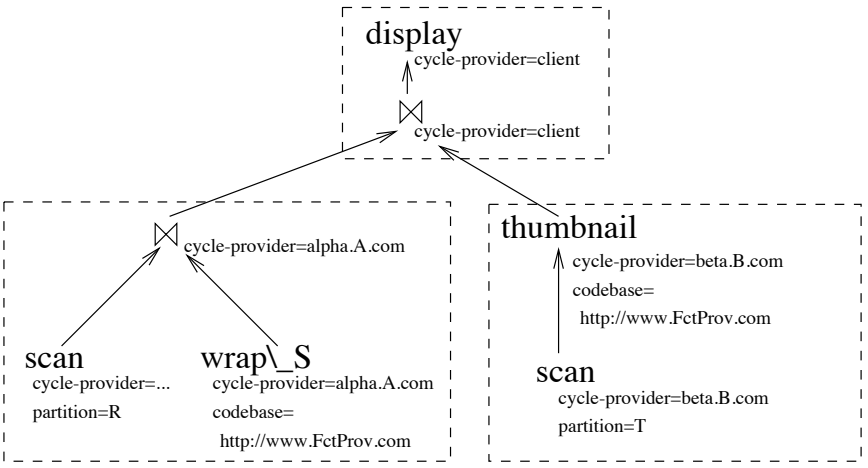


Figure 3.3: Annotated Query Execution Plan

indicates at which machine the operator is executed; e.g., the final join and the *display* operators are executed at the client. The *partition* annotation of a *scan* iterator indicates which collection is to be read. The *codebase* annotation indicates from which function provider an external query operator is loaded. *scan*, *display*, and the *joins* are built-in operators so that they do not have a *codebase* annotation.

Although the example above is rather small (in order to be illustrative) we expect ObjectGlobe systems to comprise a large number of cycle providers and far more data providers, with several of them contributing data to a specific theme. Figure 3.4 shows the structure of an example query which extracts information from a number of online databases that belong to different real estate brokers. The query uses a user-defined nearest neighbor operator (called *nn_10* in the figure) loaded from a function provider that is specialized on real estate data. The nearest neighbor logical operator is transitive and reflexive and hence allows us to perform the search for the ten nearest neighbors of a user-defined feature vector by first computing the ten nearest neighbors at every data provider and then combining these results for computing the ten nearest neighbors of the whole real estate data set. The Union operator could be carried out by one of the cycle providers that carry out the low-level *nn_10* operations or by a dedicated cycle provider in order to increase (pipelined) parallelism. Pure, dedicated cycle providers are also necessary in this example if one of the real estate data providers is not capable (e.g., not enough main memory) or not willing (e.g., for security reasons) to serve as a cycle provider.

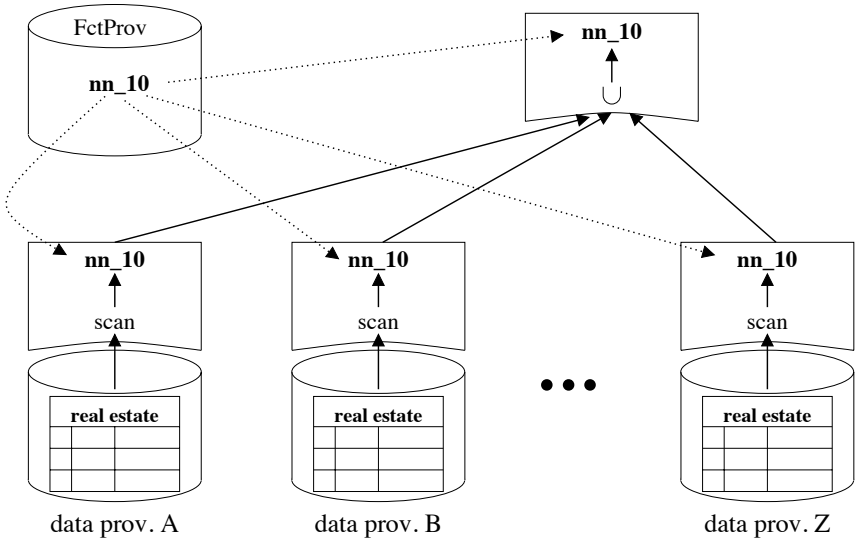


Figure 3.4: Parallel Execution in ObjectGlobe

3.2.3 Quality of Service (QoS)

As seen in the real estate example query, query execution in ObjectGlobe can involve a large number of different function, cycle and data providers. A traditional optimizer produces a plan

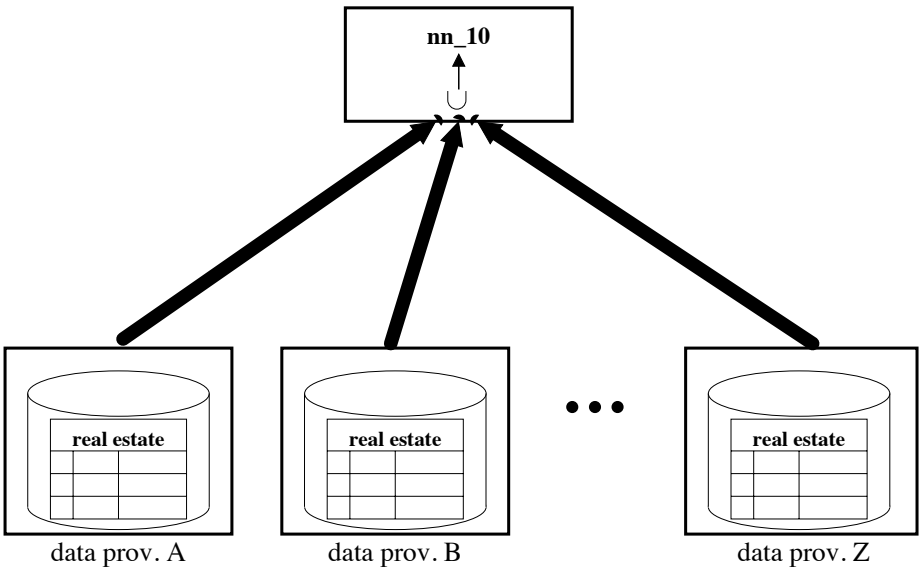


Figure 3.5: Execution in a Middleware System

that reads all the relevant data (i.e., considers all real-estate data providers). Therefore, the plan produced by a traditional optimizer will consume much more time and cost than an ObjectGlobe user is willing to spend. In such an open data integration system it is essential that a user can specify quality constraints on the execution itself. These constraints can be separated in dimensions for the query result, the execution costs and the response times.

The techniques to support the enforcement of such quality constraints need to be integrated in all steps of query processing. First, the optimizer generates a query evaluation plan whose estimated quality parameters are believed to fulfill the user-specified quality constraints of the query. For every sub-plan the optimizer states the minimum quality constraints it must obey in order to fulfill the overall quality estimations of the chosen plan and the resource requirements which are believed to be necessary to produce these quality constraints. If, during the plug phase, the resource requirements cannot be satisfied with the available resources, the plan is adapted or aborted. The QoS management reacts in the same way, if during query execution the monitoring component forecasts an eventual violation of the QoS constraints.

3.2.4 Privacy and Security Requirements in ObjectGlobe

Safety is one of the crucial issues in an open and distributed system like ObjectGlobe. ObjectGlobe provides the infrastructure to deal with the following privacy and security issues:

Protection of Cycle and Data Providers:

It has to be ensured that the resources of the cycle and data providers are protected from (possibly malicious) external operators loaded from unknown function providers. Based on the Java security model, all external query operators are therefore executed in a protected area, a so-called *sandbox* (Section 4.2.4).

Privacy and Confidentiality:

Data and function code that is processed in the ObjectGlobe system is protected against unauthorized access and manipulation. The communication streams between ObjectGlobe servers are protected using the well-established secure communication standards SSL (Secure Sockets Layer) [FKK96] and/or TLS (Transport Layer Security) [DA99, TLS] for encrypting and authenticating (digitally signing) messages. Both protocols can carry out the authentication of ObjectGlobe communication partners via X.509 certificates [HFPS99, PKI]. Furthermore, confidential information or function code is protected from being transferred to untrusted cycle providers by enforcing an authorization scheme on the flow of data and operator code specified in the site annotations of the query plan.

User Authentication/Anonymity:

ObjectGlobe supports a flexible authentication policy. Users and applications that only access free and publicly available resources can be anonymous and no authentication is required. If a user accesses a resource that charges and accepts electronic money, then the user can again stay anonymous and the electronic money is shipped as part of the "plug" step. Authentication is only required for authorization or accounting purposes of providers. Cycle providers can also require authenticated external operators to restrict the function providers; e.g., to execute only code originating from trusted sources within the same company or Intranet.

Authorization:

Some providers constrain the access or use of their resources to particular user groups. As already mentioned, providers can also constrain the information (function code) flow to ensure that only trusted cycle providers are used in the query execution plan. In general, providers apply their own autonomous authorization policy and control the execution of, say, query operators at their site themselves. In order to generate valid query execution plans and avoid failures at execution time, ObjectGlobe must know about these authorization constraints, which means, that they must be incorporated in its lookup service.

3.2.5 Comparison to Other System Architectures

Distributed database systems have been studied since the late seventies in projects like System R*, SDD-1, or Distributed Ingres. A survey of existing distributed query processing techniques studied in these projects is given in [Kos01]. ObjectGlobe shares with all these projects

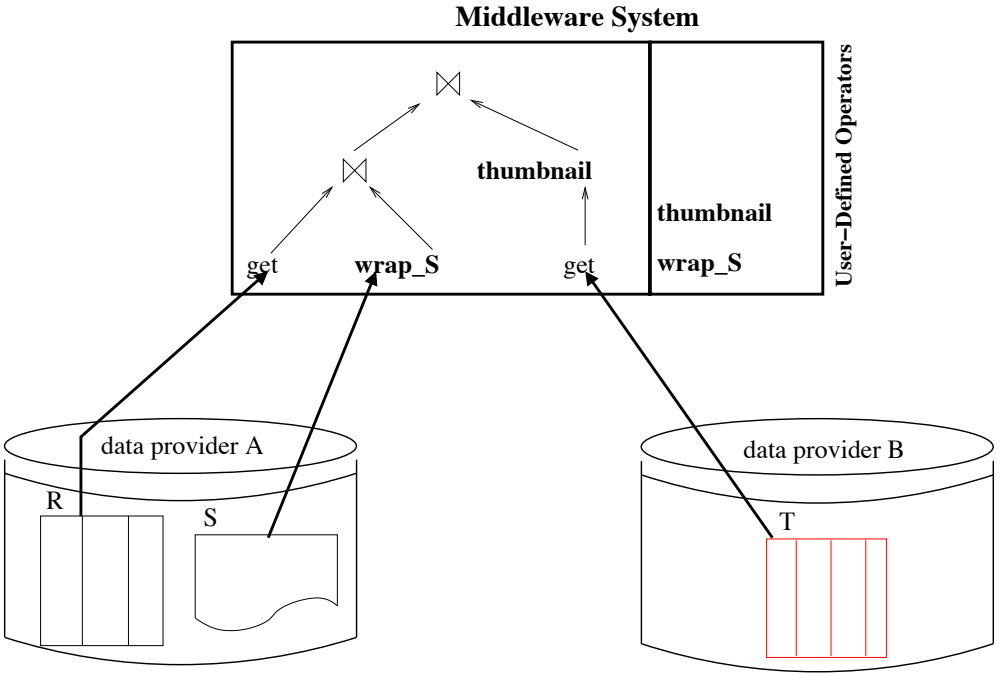


Figure 3.6: Query Evaluation in a centralized Middleware System

the vision that a distributed system can be used as easily as a centralized system (i.e., transparency) and that good performance can be achieved by sophisticated query optimization. The architecture of ObjectGlobe is more general than that of a traditional system like System R*. In a traditional system, every site acts as a data and cycle provider which executes built-in query operators; obviously, ObjectGlobe supports such a scenario as well. In addition, ObjectGlobe provides the flexibility to integrate external operators and a large number of non-database (legacy) data sources.

Today, external operators and/or legacy data sources are typically integrated using a middleware architecture; examples are Garlic [C+95] from IBM, Information Manifold [LRO96], TSIMMIS [PGGMU95], DISCO [TRV98] or Tukwila [IFF+99]. Again, ObjectGlobe's architecture is more flexible, resulting in better performance. Let us see how our example query shown in Figure 3.2 would be processed in a middleware system. As shown in Figure 3.6, middleware systems can only exploit the (limited) query processing capabilities that are hard-wired into the (legacy) data sources. If new operators are needed, such as *wrap_S* and *thumbnail*, these operators are executed at a central middleware site. This is also true for distributed middleware systems like Amos*II* [JKR99], because the corresponding server processes are restricted to the mediator's capabilities and cannot be extended by dynamically loaded mobile code. This means, that only specific servers, which can be prepared by a user in advance, can execute his/her application specific code. In Figure 3.4 the ObjectGlobe version of the nearest neighbor example plan is depicted. In contrast to the traditional execution plan of middleware systems as shown in Figure 3.5 the ObjectGlobe plan, which uses dynamic operator loading, can exploit parallel

execution of several nearest neighbor operators and causes much less network traffic. As a result, a middleware system incurs high communication costs for shipping the data to the middleware; i.e., for data shipping [FJK96]. ObjectGlobe helps reduce such communication costs by allowing to execute new and external query operators at or near the data providers.

Various aspects of the ObjectGlobe project have already been studied in other projects. The notion of an open market place in which different providers compete for queries is borrowed from Mariposa [SAL+96]—even though, ObjectGlobe does not enforce a particular business model like Mariposa. Mariposa also has some notion of QoS, but we consider user-defined quality constraints during all phases of query execution, whereas Mariposa tries to obey these constraints only during its plan fragmentation step, which takes place after optimization. We believe, that this is not sufficient in such an Internet-wide open query processor.

Extensibility has been studied in a number of database projects; e.g., Postgres [SR86], Starburst [HCL+90], or more recently in Predator [SLR97]. The safe execution of external functions has been studied in [GMSvE98], but the scope of that work is too limited for our context.

There has also been a large body of related work on the integration of services in open distributed object systems. The most prominent examples are Jini [Wal99] and CORBA [MZ95]. A related lookup service is HP's Chai (Plug & Play) system [HPI99]. The UDDI standard [UDD00] defines a framework for the management of meta data about electronic commerce Web services. Architectures for distributed object systems have been devised in the SHORE [CDF+94], Ninja [GWBC99], and AutO [Kri98] projects. The AutO project was also conducted at the University of Passau and we adopted many results such as the AutO security model and infrastructure for ObjectGlobe. As part of the Ninja project, a secure distributed directory service has been developed [CZH+99]. ObjectGlobe's lookup service also bears some similarity with X.500 [CCI88] and LDAP directory services [WHK97]. What makes ObjectGlobe different from all these works is that ObjectGlobe is capable of complex query processing; that is, a single ObjectGlobe query can involve the lookup and execution of many different services and it requires optimization because of the large amounts of data that need to be processed. In this respect, ObjectGlobe's lookup service is similar to [MRT98]'s WebSemantics project which uses Web documents to publish the location of components (wrappers and data sources) and a uniform query language to locate data sources based on this meta-data and to access the sources.

In other lines of work, researchers have tried to "query the Web" using languages like WebSQL [MMM97, KS98]; these efforts, however, only support a navigational style of access of Web pages. Junglee [GHR97] follows a data warehousing approach in order to integrate Internet data for query processing. Furthermore, Web site management has been studied in a few recent projects; e.g., Strudel [FFK+98]. The goal of systems like Strudel, however, is to improve the services (and manageability) of a single site, rather than integrating services from multiple sites.

Chapter 4

The Architecture of the ObjectGlobe System

In this chapter details of the ObjectGlobe architecture are shown. We concentrate on the techniques used during query optimization and query execution. Special emphasis during query optimization must be laid on the design and provision of meta-data and the handling of the huge search space. Another important item is security which has to be regarded during optimization and execution. Additionally, we describe the internal structures of the query processor which enable the distributed execution of a QEP.

4.1 Generating Query Plans

In this section, we show how ObjectGlobe produces a plan for a query. In particular, we describe the ObjectGlobe *lookup service* that finds relevant resources for a query and the parser and the optimizer that try to find a good plan to execute a query. Currently, ObjectGlobe supports a subset of SQL; ObjectGlobe, however, does support the use of external functions as part of a query.

4.1.1 Lookup Service

The lookup service plays the same role in ObjectGlobe as the *catalog* or *meta-data management* of a traditional query processor. Providers are registered before they can participate in ObjectGlobe. In this way, the information about available services is incrementally extended as necessary. A similar approach for integrating various business services in B2B e-commerce has been proposed recently in the UDDI standardization effort [UDD00].

We expect the registration of providers' services to become a similar market as the market for the providers themselves. So, someone interested in using a service will register this service; service providers themselves need not necessarily do this on their own. For example, wrapper developers are of course interested in registering data sources for which they have written the corresponding wrappers. Such an incremental schema enhancement by an authorized user is

possible in the ObjectGlobe lookup service just as in any other database system. This means, that an ObjectGlobe system is normally not tailored for a specific data integration problem, but can dynamically be extended with new data, cycle, and function providers by augmenting the meta-data of its lookup service.

The ObjectGlobe parser and optimizer consult the lookup service in order to find relevant resources to execute a query and obtain statistics. Furthermore, end users can use the lookup service to browse through the meta-data and search for available query capabilities and data sources for their applications.

ObjectGlobe's Meta-data

The ObjectGlobe lookup service records the following information:

data provider: Each collection of objects stored by a data provider and the *attributes* of each collection are recorded by the lookup service. A collection is either a materialized partition conforming to ObjectGlobe's internal nested relational format or a virtual collection, i.e., an Internet data source transformed into the collection's recorded schema by a wrapper. Collections are associated to a specific *theme*. A theme describes a special concept with a set of terms, called *attributes*. A theme's attributes can be viewed as the union of all attributes meaningful for the theme. Queries are formulated over the themes and their attributes. Integration of a new data source is achieved by registering it as a new collection and associating it to a theme. So collections can be seen as horizontal (possibly overlapping) partitions. The attributes provided by the new collection must be a subset of the attributes defined by the associated theme. Currently ObjectGlobe uses a non-hierarchical set of themes, but more complex ontologies [BCV99] could be added on top of our flat theme structure. As an example, `www.HotelBook.com` and `www.HotelGuide.com` provide different collections which are associated to the theme *hotel*.

Furthermore, the lookup service stores binding patterns of a collection, statistics about a collection like histograms for estimating the selectivity of simple (i.e., non-external) predicates, and information about replicas (i.e., mirrors) of a collection, which could be provided by some other data provider.

cycle provider: The CPU power, size of main memory, and temporary disk space of each cycle provider is recorded. The load on the cycle provider regarding CPU power and available main memory is stored as a function of time and likewise we store the latency and bandwidth information for the network links between cycle providers.

function provider: The name and signature of each query operator is recorded. Furthermore, formulas to estimate the consumption of CPU cycles, main memory, disk space, and the selectivity for each query operator are kept by the lookup service. These formulas use a set of parameters which describe the characteristics of the executing cycle provider (e.g., the available CPU power/main memory) and the input data for a specific application of this operator.

ObjectGlobe differentiates between *iterators* like join or display and *transformers* such as *thumbnail*. (In addition, ObjectGlobe has also special categories for *predicates* and *aggregate functions*.) Any kind of function, however, will automatically be wrapped by ObjectGlobe into an iterator so that we ignore these distinctions in this work and use the words *function* and *query operator* interchangeably for the general concept.

authorization information: the lookup service maintains authorization information which is obtained from the providers and indicates which data may be processed at which cycle provider and by which query operator. To guarantee privacy and confidentiality, the providers can also restrict the flow of information (and code) in order to prevent data (and functions) from being processed on untrusted cycle providers. Following the ObjectGlobe authorization model, it is possible to specify positive and negative authorizations [RBKW91, BJS99]. Also, it is possible to group collections, functions, and cycle providers into "authorization classes"—using role-based authorization [SCFY96]—in order to reduce the overhead of maintaining and processing this information in the lookup service.

Appendix B shows an example `RDF` document that can be used by a data provider to register a *hotel* collection. The meta-data kept in the lookup service can be outdated or incomplete. It is possible, for instance, that a data provider revokes the privilege of some cycle providers to process its data without notifying the lookup service; as a result, the execution of a query might fail due to an authorization violation which is detected at execution time. ObjectGlobe relies on data, function, and cycle providers to notify the lookup service if important meta-data changes. If a plan fails due to stale meta-data in the lookup service, all the relevant meta-data is invalidated so that providers that do not update their meta-data are eventually excluded from the ObjectGlobe federation. As an alternative, [CZH+99] proposes to use a *time-to-live* scheme; in that scheme, providers must periodically contact the lookup service if they want to continue to remain in the federation.

Using the ObjectGlobe Lookup Service

As mentioned before, data, function, and cycle providers are registered by generating `RDF` documents describing their services. We use `RDF` because it is very flexible and a WWW standard for describing resources [BG99]. Typical collections, such as relational or `XML` data sources, can very easily be described using `RDF`; it is also possible to automatically produce large fractions of an `RDF` description from, say, an `XML DTD` or a relational schema. An `RDF` document is also used to update the meta-data if a provider changes or extends its services and the ID of an `RDF` object is used to unregister (i.e., delete) services.

To find relevant resources and retrieve statistics and authorization information, the lookup service provides a declarative query language. As an example, Figure 4.2 shows how to ask the lookup service for all collections that supply data for the *hotel* theme. More specifically, the query of Figure 4.2 asks for *hotel* collections which have *city*, *address*, and *price* attributes and the query asks for the `uniqueId` of the collection (used to identify replicas) and information about all *attributes*. (The "?" in the query is an *any* operator.) The result of this query is shown

select price, address
from hotel
where city='New York'

Figure 4.1: An Example SQL Query

search Partition d
select d.uniqueId, d.attributes.∗
where d.theme.name="hotel"
and d.attritutes.?.topic="city"
and d.attritutes.?.topic="address"
and d.attritutes.?.topic="price"

Figure 4.2: Example Lookup Search Query

```
<collection>
  <uniqueId>4711</uniqueId>
  <attribute topic="city" domain="String"/>
  <attribute topic="price" domain="Integer"/>
  <attribute topic="address" domain="String"/>
</collection>
```

Figure 4.3: Example Search Result

in Figure 4.3; here, we show the results for the *hotel* collection specified in the RDF document of Appendix B.

The lookup service also allows the definition of views. These views can be materialized. Such materialized views are very helpful to support *sessions* in which search results are iteratively refined. For example, it is possible to first ask for all cycle providers which are allowed to process objects of a specific collection and then, in a separate search request, ask which of *these* cycle providers are capable to execute a specific query operator.[1] This feature is important for parsing and optimization and for users who interactively browse the meta-data. An advanced implementation of the ObjectGlobe lookup service with a distributed and layered architecture was devised in [KKKK02]. Additionally, this implementation uses a publish and subscribe mechanism to allow an efficient distribution of meta-data.

4.1.2 Parser and Optimizer

Plans for a query are generated by the ObjectGlobe query parser and optimizer. As shown in Figure 3.1, the parser looks up the relevant resources for a query and the optimizer produces a plan based on (a subset of) these resources.

Parser

The main effort carried out during parsing is to issue search requests to the lookup service in order to discover all relevant resources (i.e., collections, functions, and cycle providers). The parser

[1] Of course, these cycle providers could also be found in a single search request.

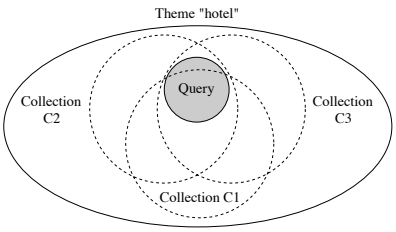


Figure 4.4: Relationship of Theme, Collections, and Query Attributes

aborts the processing of a query if for some part of the query, no resources can be found. Relevant collections are found using the *themes* and *attributes* specified in a query. All themes used in the query's `FROM` clause and their corresponding attributes used in the `SELECT` and `WHERE` clauses define the query's schema. The relationship between the attributes used in a query, the attributes recorded for collections in the lookup service, and the attributes of a theme are depicted in Figure 4.4. For every *theme* referred to in the query, the parser queries all *matching* collections from the lookup service; a collection matches if it is associated to the requested theme and provides a superset of all attributes used in the query. For example, assume the SQL query given in Figure 4.1. From this query the parser determines a schema consisting of the attributes *hotel.city*, *hotel.address*, *hotel.price*, represented by the gray filled circle in Figure 4.4. To find all relevant collections the parser queries from the lookup service all collections associated to the *hotel* theme (collections C1, C2, and C3) and providing at least the attributes *city*, *address*, and *price* (only collections C2 and C3). The resulting search request to find relevant collections for the query of Figure 4.1 is given in Figure 4.2.

As Figure 4.4 shows, collections may provide more attributes than are actually used in a query. In the execution phase, the schema of a collection is projected to the schema required by the query execution plan. So, in Figure 4.4, the operator used to access collection C2 will not return all attributes represented by the dashed circle C2, but only the attributes of the intersection of the sets C2 and Query (the attributes *city*, *address*, and *price*).

Likewise, the parser looks for function providers for each external function used in a query; again, external functions such as *thumbnail* can have several implementations from different function providers; all implementations that match the right name and signature are considered. Query operators such as *join, union,* or *display* are typically implicit in a query; for *join* and *union* the parser will consider built-in variants and all variants provided by function providers. For *display*, the parser will always consider ObjectGlobe's built-in variant which produces `XML` to represent query results; the parser will only consider a different *display* operator if this is explicitly requested.

In addition to discovering the relevant resources, the parser consults the lookup service in order to retrieve all available statistics and authorization information. As a result, the parser produces a (quite complex) `XML` document which is then used by the optimizer in order to generate a plan. Figure 4.5 shows how the authorization and applicability information is represented as a *compatibility matrix* for the collections, functions, and cycle providers of the example of Section 3.2.2. For each relevant data collection such a compatibility matrix is generated by the

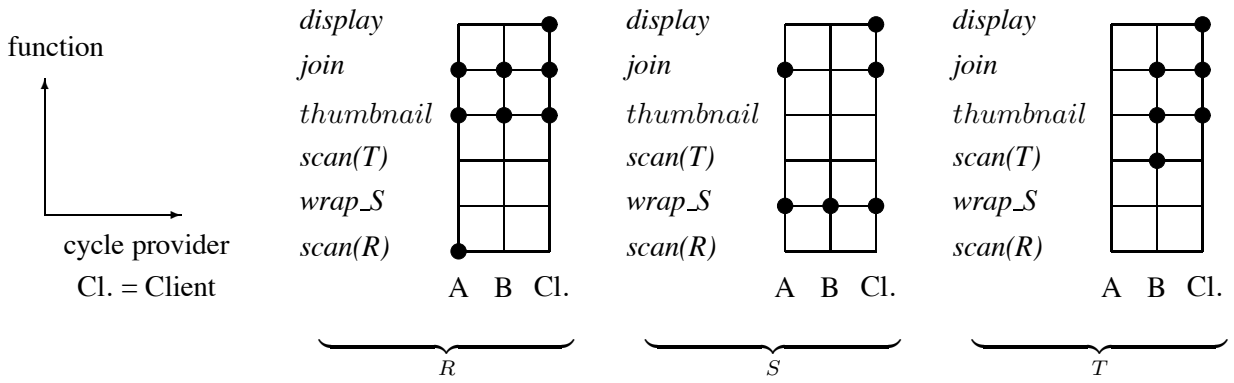


Figure 4.5: Compatibility Matrices for the Example of Section 3.2.2

parser. A point at (c, f) in a matrix of a collection is set if cycle provider c is authorized to see the collection, function f is authorized to process objects of the collection, c is authorized to execute f, and c is capable of executing f (i.e., has enough memory and disk space). For instance, *wrap_S* may be executed at all cycle providers in order to read collection S, but it may obviously not be used anywhere to read collection R or T. In the matrix, built-in query operators such as *display, scan*, and *join* are treated in the same way as external functions (e.g., *thumbnail* and *wrap_S*); it would be possible, for instance, that a cycle provider only allows its own join methods to be executed on its machines.

Up to now we did not explain how the set of cycle providers which are considered for a specific query, is initially fixed by the parser. Naturally, the selection of cycle providers for a specific query depends on their abilities (CPU speed, network connection, memory sizes) and their authorizations in respect of the used collections and functions. The abilities of a cycle provider must be considered with respect to its assigned operators and the way these operators are embedded in the QEP. Therefore, a meaningful decision about the cycle provider which should be used for a specific operation in the QEP can only be made during optimization. The parser just has to pre-select cycle providers which are allowed to work on relevant collections. During optimization the compatibility matrices as explained above have to be consulted in order to select potential cycle providers for specific operations.

Optimizer

The goal of the optimizer is to find a good plan to execute a query, if a plan exists. The "if a plan exists" part is important because the ObjectGlobe optimizer, unlike a traditional optimizer, might sometimes fail to find a plan, even if the parser was able to discover relevant resources. First of all, limitations due to authorizations can make it impossible to find a valid plan; for instance, it might happen that two collections cannot be joined because there is no cycle provider that has permission to see both collections. Furthermore, ObjectGlobe users and applications can specify quality parameters for the query execution itself as described later in this work. For example, if the user's upper bound for the costs of a query is 10 € and the optimizer does not find a matching

plan for this constraint, the user is informed about this fact and no query execution takes place.

The optimizer enumerates alternative query evaluation plans using a System-R style dynamic programming algorithm [SAC+79] or (optionally) a greedy algorithm. Both algorithms have in common that the optimizer builds plans in a bottom-up way: first so-called *access plans* are constructed that specify how each collection is read (i.e., at which cycle provider and with which *scan* or *wrapper* operator). After that, *join plans* are constructed from these *access plans* and (later) from simpler *join plans*. The basic ideas of both optimization algorithms were already presented in Section 2.2.1. To deal with unary external functions and predicates, the algorithms are extended as described in [CS96].

In the following, we would like to highlight the peculiarities that make the ObjectGlobe optimizer special:

Compatibility Matrix During query optimization every plan is annotated (among others) with a compatibility matrix. The compatibility matrix of an access plan is identical with the compatibility matrix generated by the parser for the corresponding collection (4.5). The matrix of a join plan which is composed of two sub-plans is generated by ANDing the two compatibility matrices of the two sub-plans, resulting in a more restrictive matrix.

Sanity Checks Some sub-plans can be immediately discarded during plan enumeration based on the sub-plan's compatibility matrix. As an example, consider the following situation: collections R_1 and R_2 belong to the same theme $\mathcal{R}$ and a query is interested in $f(\mathcal{R})$ for some external function f. For collection R_1, f may only be executed by cycle provider x; for collection R_2, f may only be executed by cycle provider y. Now a sub-plan $R_1 \cup R_2$ can immediately be discarded because there is no way to execute f on top of $R_1 \cup R_2$ (neither x nor y work); in other words, the $R_1 \cup R_2$ plan has no points set in the f row of its compatibility matrix. (Note, however, that an $f(R_1) \cup f(R_2)$ plan is valid, if it is equivalent.) If several variants of f exist, then the $R_1 \cup R_2$ plan can be discarded if there is no point set in the *shelf* of f rows. (A shelf is a set of rows in the matrix for different variants of the same function.) Obviously, a plan can also be immediately discarded if an estimated value for one of its quality parameters exceeds the specified limit.

We also carry out more sophisticated sanity checks at the beginning of query optimization. For example, there must be at least one cycle provider which has permission and is capable to execute the *display* operator for each collection. Typically, this must be the client machine at which the query was issued. If such a cycle provider does not exist, then no plan exists and the optimizer can stop without enumerating any plans. In theory, such sanity checks that span several compatibility matrices could be applied in order to discard certain sub-plans during the plan enumeration process; since these sanity checks are quite costly, however, they are only carried out once, at the beginning before plan enumeration starts.

Compact Query Evaluation Plans As we have already seen in Section 4.1.1, several collections can contribute data for a specific theme. A similar situation can be found in distributed databases, where the data for a relation can be splitted in several horizontal partitions. If in a distributed database such a relation is referenced in a query, all the partitions of the relation

are unioned in the resulting QEP in order to consider all the data of the relation in the query evaluation.

In a globally operating data integration system we must expect that a large number of data providers contribute to the same theme. For example, data for real estate offers, hotel rooms, or flight information are offered by many data providers on the Internet. Since these data providers can be distributed over the whole Internet, the incorporation of all the corresponding collections in a query execution would be too expensive for most practical applications. This means that the response time of the query would be too large.

Therefore, the ObjectGlobe optimizer allows users to specify a completeness constraint for every theme in a query. This constraint states the minimum percentage of all the data for the corresponding theme which should be considered in the query execution. This is one aspect of the quality of service properties which are explained in more detail later in this work. Here, we are interested in the resulting possibility to select a subset of the collections of a theme in order to fulfill a completeness constraint.

Naturally, the distribution of data providers affects the execution speed of the corresponding query. Here, we assume that the predominant factors which affect the response time of a query, are the bandwidths of the used network connections. Hence, if the completeness constraints allow the selection of a reduced number of data providers, this selection should induce a rather compact QEP in order to achieve a reduced response time. This means that the data providers should be located as near as possible to one another, i.e., they should be connected by rather fast network connections.

In the worst case, a brute force method would have to check all combinations of different subsets (except for the empty set) of data providers for each theme. For each such combination the respective optimization algorithm would have to be applied. Thus, the running time of the optimizer is increased by the factor

$$\prod_{i=1}^{n}(2^{s_i}-1)$$

where n is the number of themes referenced in the query and s_i is the number of collections registered for the i-th theme. Obviously, this factor has in almost all practical scenarios a value which forbids the application of this brute force method. Therefore, we use heuristics which are based on the bandwidth of the network connections between the data providers. We regard all data providers and the corresponding network connections between them as a fully connected, weighted graph. The data providers are the nodes, the network connections the edges, and the bandwidth of a connection represents its weight. We recursively apply a clustering algorithm [Ger96] on this graph. Each recursive application delivers a disjunctive partitioning of all data providers and through the information of how the clusters evolved from former applications of the clustering algorithm a so-called cluster tree can be constructed. An example for such a tree is depicted in Figure 4.6.

Now, as a prerequisite for finding access paths for collections of a theme we are searching for special clusters in the cluster tree. These clusters must contain a set of collections for the respective theme which fulfills the completeness constraint for that theme. Obviously, the root of the cluster tree is always such a cluster. Since a greater depth of the cluster in the tree means

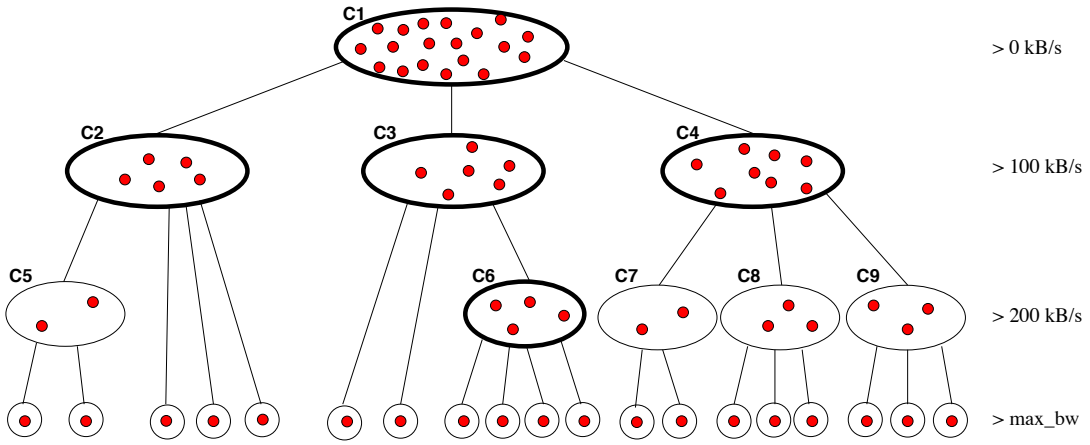


Figure 4.6: A Cluster Tree informs about the Quality of the Network Connections between Data Providers.

that the data providers in that cluster have better network connections to one another than in any cluster above, we are searching for the deepest clusters which fulfill the completeness constraint. For example, in Figure 4.6 we assume that all clusters with a thick border fulfill such a constraint for a theme. The candidate clusters are then *C2, C4* and *C6*[2]. The search for such candidate clusters for themes can be seen as a search for theme-level access plans. Analogously, we handle these theme-level access plans as alternatives in the optimization process. This means that each candidate cluster at least delivers one sub-plan for accessing the corresponding theme and this sub-plan is considered during optimization. For example, assume that we join the themes $t1$ with candidate clusters $C6$ and $C2$ and $t2$ with candidate clusters $C3$ and $C9$. If we use our dynamic programming optimizer in a configuration which keeps two alternative physical plans for each logical plan, the plan which uses $C6$ for accessing $t1$ and $C3$ for accessing $t2$ will most likely be chosen, since within $C3$ a fast access to the collections of $t1$ *and* $t2$ is available.

Selecting Cycle Providers During optimization, for every operator which has to appear in the QEP a cycle provider has to be determined which executes this operator. Again, testing all registered cycle providers for their ability to execute a specific operator results in a prohibitively high running time for the optimization process.

Naturally, the selection of cycle providers in a query has to consider the placement of used collections. The clusters which were chosen for accessing data providers can also guide the selection of appropriate cycle providers which work on the corresponding data collections. Therefore, additionally to data providers also cycle providers are considered in the construction process of the cluster tree. In this way, every cycle provider (analogously to a data provider) is associated with a path in the cluster tree which starts at the root cluster and ends in the leaf node which represents the corresponding cycle provider.

During the bottom-up construction of a QEP, each of its sub-plans is associated with the

[2]Note that more than one such cluster can exist if the completeness constraint is below 50%.

lowest cluster in the tree which covers all the data and cycle providers which appear in this sub-plan. This cluster represents the tightest environment of this sub-plan which can be found in the cluster tree. Initially every access plan for a data collection is associated with the leaf node of the cluster tree which represents the respective data provider. This means that access plans are associated with a cluster *and* a specific host. Every other kind of sub-plan has in addition to the association with a cluster also a representative host which is the cycle provider executing the top most operation in the sub-plan.

We assume now that the sub-plans s_1 and s_2 with associated clusters c_1 and c_2 and associated hosts h_1 and h_2 have to be combined by a binary operator. The cycle provider for executing this operator is chosen from the set of cycle providers which appear in the cluster which is the lowest common ancestor of c_1 and c_2. For example, if the clusters are $C2$ and $C6$ of Figure 4.6, the cycle provider is searched in cluster $C1$. Although, the number of cycle providers in such a cluster is normally less than the overall number of cycle providers (unless the cluster is the root cluster) it may be too expensive to test all the cycle providers during optimization time. Therefore, for every h_1 and h_2 we determine a set of candidate cycle providers which could be useful to execute the operator which combines plans associated with these hosts. This selection does not depend on a specific query but just on the bandwidths of the affected network connections and need not be recomputed for every query.

Depending on the sizes of the intermediate results and the way a QEP can be completed in further steps of the optimization process several strategies for choosing a cycle provider are conceivable. The following cycle providers can be useful:

1. h_1 and h_2 themselves.
2. a cycle provider cp with a high value for *bandwidth(cp,*h_1*)* or[3] *bandwidth(cp,*h_2*)*.
3. a cycle provider cp with a high value for *bandwidth*(cp, h_1) + *bandwidth*(cp, h_2).
4. a cycle provider cp with a high value for *bandwidth*$(cp, h_1) * $ *bandwidth*(cp, h_2).

The function *bandwidth* determines the bandwidth between the hosts given as parameters. To select the most appropriate cycle providers for the last three criteria, priority queues are used. A user-defined bound b determines how many cycle providers from each such priority queue should be selected. The optimization algorithm then has to check at most $3 * b + 2$ cycle providers for every binary operation. In Figure 4.7 some candidate cycle providers for the hosts h_1 and h_2 are shown. The cycle providers connected with a solid line to h_1 or h_2 are candidates due to the second criterion. Analogously, the cycle providers connected with a dashed and dotted line are candidates due to the third and fourth criteria, respectively.

Unary operators are not placed at separate cycle providers by our optimizer. If a unary operator must be inserted in a QEP, it is executed at the cycle provider which is associated with the current sub-plan. If this is not possible (e.g., if the sub-plan is associated with a pure data provider) the insertion of this operator is delayed until the cycle provider for the next binary operation has been selected.

[3]The *or* is an exclusive or here.

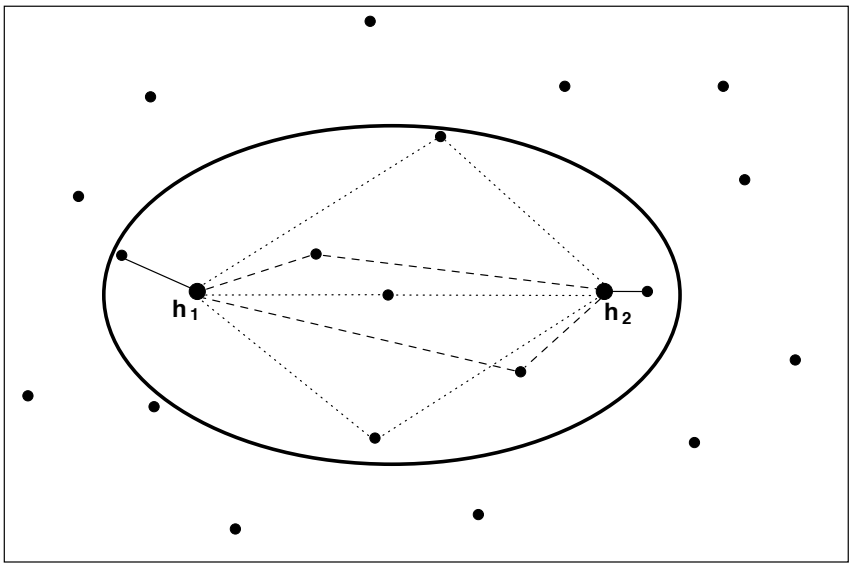


Figure 4.7: The Criteria for Selecting Cycle Providers for Binary Operations.

UNION Queries As shown earlier, several data providers can contribute data for the same theme. The simplest way for incorporating the data for such a theme in a QEP would be to perform *union* operations on all the affected and selected (see above) data collections before any other operation is considered for that theme. For example, if the themes R and S with collections $\{s_1, s_2\}$ and $\{r_1, r_2, r_3\}$ must be joined, the resulting QEP would then be

$$(s_1 \cup s_2) \bowtie (r_1 \cup r_2 \cup r_3).$$

In the following we say that this kind of QEP is in standard form. Since the union operator is associative and a join operator distributes over a union operator, an equivalent expression to the previous one would, for example, be

$$(s_1 \bowtie (r_1 \cup r_2 \cup r_3)) \cup (s_2 \bowtie (r_1 \cup r_2 \cup r_3)).$$

The latter expression will normally result in a query execution with a better response time than the execution of the standard form QEP. This is due to the join operations which can be executed in parallel by different cycle providers and the reduced data volume each of the join operations has to process. The savings in response time then simply result from the fact that a join operator is in general much more expensive than a union operator.

Therefore, plans which are deduced from a standard form QEP by the application of such equivalence transformations are normally good candidates during query optimization. Unfortunately, the number of plans which have to be considered during optimization when all possible equivalence transformations should be taken into account, is rather large. If we have to join the themes A_1 and A_2 with corresponding numbers of collections a_1 and a_2, the number of QEPs for

this query can be computed by the function *UJ*:

$$\mathrm{UJ}(a_1, a_2) = \sum_{j=1}^{a_1} \left(\left\{ \begin{matrix} a_1 \\ j \end{matrix} \right\} \mathrm{bell}(a_2)^j \right) + \sum_{j=1}^{a_2} \left(\left\{ \begin{matrix} a_2 \\ j \end{matrix} \right\} \mathrm{bell}(a_1)^j \right) - \mathrm{bell}(a_1)\mathrm{bell}(a_2)$$

In this definition $\left\{ \begin{matrix} n \\ k \end{matrix} \right\}$ denotes the Stirling number of the second kind which represents the number of ways a set with n elements can be partitioned into k disjoint, non-empty subsets. The term $\mathrm{bell}(n)$ denotes the Bell number which represents the number of ways a set with n elements can be partitioned into disjoint, non-empty subsets. The definition of *UJ* follows the construction of a QEP starting from its standard form. First we have to select a theme the collections of which are partitioned first. Each such partition has to be joined with an expression which is equivalent to the other theme. All these expressions are counted by the call to the function for the Bell numbers. At the end we have to consider duplicate QEPs which are generated when for every appearance of a theme in a QEP the same partitioning is selected. If the same partitionings are selected, the order in which the themes are used in the construction of a QEP does not matter anymore. Therefore, the last term of the definition of *UJ* includes the number of QEPs with that property.

For a join with $a_1 = 2$ and $a_2 = 2$ there are 8 different QEPs as counted by *UJ*[4]. The latter number rises up to 913749304 for $a_1 = 5$ and $a_2 = 5$. The number of such plans for n-way join plans can be computed by recursively applying the function *UJ* on intermediate results of the join plan.

The cluster tree from above also helps here to implement heuristics which allow us to consider a promising subset of all QEPs which can be constructed from the standard form QEP by the transformations shown above.

Suppose that we have to create a plan which joins the sub-plans *R* and *S*. *R* and *S* represent joins of complete themes; that is, there exists two set of themes $\{A_1, \ldots, A_n\}$ and $\{B_1, \ldots, B_m\}$ with $R \equiv A_1 \bowtie \ldots \bowtie A_n$ and $S \equiv B_1 \bowtie \ldots \bowtie B_m$. *R* and *S* are in the form $R = R_1 \cup \ldots \cup R_k$ and $S = S_1 \cup \ldots \cup S_l$. Initially, every theme itself represented by the plan which unions all selected collections of the theme fulfills these properties.

As shown above, a cluster C_j for the join operation is selected. The direct sub-clusters of C_j induce a partitioning on $R_1, \ldots, R_k$ and $S_1, \ldots, S_l$. The corresponding partitions are named $RP_1, \ldots, RP_p$ and $SP_1, \ldots, SP_q$. The QEP which is then chosen for the join adopts the structure of these partitionings:

$$\left(\bigcup_{r \in RP_1} r \right) \bowtie \left(\bigcup_{s \in SP_1} s \right) \cup \ldots \cup \left(\bigcup_{r \in RP_1} r \right) \bowtie \left(\bigcup_{s \in SP_q} s \right) \cup \ldots \cup \left(\bigcup_{r \in RP_p} r \right) \bowtie \left(\bigcup_{s \in SP_q} s \right).$$

The resulting QEP again fulfills the properties which were required for R and S. This means, that this kind of partitioning can continue recursively during the QEP is completed.

Now, C_j is not necessarily used for the execution of a join because for every join operation in the resulting QEP a cycle provider is searched anew. In this way, a cycle provider which best

[4]We do not regard QEPs which can be deduced by the associative law for the join operator.

fits the positions of the corresponding partitions in the cluster tree, can be selected. In order to demonstrate the construction of the QEP for our example cluster tree in Figure 4.6, assume that R_1 is associated with cluster C_5, R_2 with C_8, S_1 with C_7 and S_2 with cluster C_9. Then, C_j is determined as C_1 and the resulting QEP is

$$(R_1 \bowtie (S_1 \cup S_2)) \cup (R_2 \bowtie (S_1 \cup S_2)).$$

Note that the last join can be executed locally to cluster C_4 which is better than an execution in $C_j = C_1$.

4.2 Query Plan Distribution and Execution

As mentioned before, ObjectGlobe was implemented in Java for two reasons: portability and security. In this section we will describe how we utilized Java's features to achieve extensibility and query operator mobility without compromising security. We will also describe ObjectGlobe's monitor concept for controlling the progress of distributed query plans.

4.2.1 Distributing Query Evaluation Plans

Query evaluation plans, represented in XML, constitute trees, where each node contains various annotations: the query operator; if it is an external operator, the code base from which it is loaded; the cycle provider on which the operator is executed; possibly some constants to substitute for the operator‘s parameters; etc. The distribution of such a query plan starts at the client with a depth-first traversal of the query evaluation plan. For each node, we check its annotations in order to obtain the data needed for instantiating the operator. First, the *cycle-provider* annotation indicates at what site the respective operator should be executed. If its value is not equal to the current site, a Send-/Receive iterator pair is inserted into the query plan. The Receive iterator is instantiated at the current site and the Send iterator at the remote site. A network connection is created between the Send- and the Receive iterator, which will be used during query execution for the transfer of the intermediate result stream.

The query plan's annotation could require to use a secure communication channel, in which case an SSL (Secure Sockets Layer) connection is established. SSL [FKK96][5] is a de-facto standard for providing privacy and reliability of network communication by encrypting network traffic and checking the data integrity using Message Authentication Codes (MAC). Also, the SSL protocol can carry out the authentication of both ObjectGlobe communication partners via certificates.

After the Send- and Receive iterators have been established, the instantiation of the query sub-plan rooted at the current node is delegated to the remote host. The instantiation at the local site continues as if the traversal of the sent-away sub-tree was finished.

If a node constitutes an external operator, the *codeBase* annotation contains the reference (an URL) to the appropriate function provider. Every cycle provider loads the code of external

[5]There is also the standardized TLS (Transport Layer Security) protocol [DA99] of the Internet Engineering Task Force (IETF) which is quite similar to the current SSL 3.0 protocol.

operators with a specialized ObjectGlobe class loader (OGClassLoader). If a cycle provider requires that the code is signed (authenticated), then the OGClassLoader will check the signature of the code. The loaded code is used to create an instance of the specified external operator. The security issues concerning such dynamically loaded code will be discussed in Section 4.2.4.

4.2.2 Authentication and Authorization

If a provider restricts the use of its resources and therefore requires some kind of authentication of users the authentication information will be part of the query plan (again, as part of an annotation). Two possible authentication schemes are supported. (1) A user can provide a password. The password is used to generate a secret key (using the PKCS#5 Password-Based Encryption Standard [RSA99]) which is afterwards used to calculate a MAC (Message Authentication Code) of the query plan and some additional data. (2) The user possesses a valid X.509 certificate [HFPS99, PKI]. The certificate is used to calculate a digital signature of the query plan and some additional data.

One problem remains. What if a data provider does not support one of these schemes, i.e., requires the password in plain text? The password is included (as authentication information) in the query plan. The wrapper accessing the data provider extracts the password and passes it to the data provider. To keep the password secure it is encrypted with the public key of the cycle provider that executes the wrapper. So no other cycle provider is able to access the plain password.

Authorization is carried out by the individual providers when a query is instantiated. Each provider autonomously decides if it allows the local execution of the query plan depending on the local policy. Most providers will delegate this decision to a local security provider which is included in the ObjectGlobe system. Data providers may also have their own security system (as most DBMSs have) that they can use instead of the ObjectGlobe security provider.

The security provider uses a role-based access control (RBAC) model [SCFY96] to specify authorization rules. RBAC distinguishes between users, roles which are assigned to users and permissions which are assigned to roles. ObjectGlobe provides permissions for allowing or denying access to a relation (i.e., executing a wrapper), loading and executing an operator and using a cycle provider (i.e., execute a query plan at the cycle provider).

4.2.3 Extensibility

The ability to load external operators into ObjectGlobe's query engine is a key feature for flexibility and performance (e.g., by moving code to the data). Arbitrary tasks can be implemented by an external operator as long as the rules of the security system are obeyed and the prescribed interface is observed. The iterator interface, which is used in our system is nearly the same as the one described in [Gra93]. The methods of this interface are shown below:

```
TypeSpec          open();
ElementDescriptor next();
void              close();
void              reopen();
```

A sample call pattern is given in Figure 4.8. In the following we briefly describe the `open` method for iterators, since it has a special meaning for the incorporation of external operators. In the `open` method the operator has to make preparations in order to be able to produce its result tuples afterwards in the calls to its `next` method. So what an operator has to do first in its `open` method is, to ensure that *its* input operators (if it is not a leaf in the operator tree) can produce their result tuples. Therefore, the operator calls the `open` methods of its input operators. The `open` method returns an object of a class named `TypeSpec`. Such an object describes the type of the tuples which will be produced in every call of the `next` method. Type specifications are also recorded in the lookup service; just like authorization information, however, the type specifications recorded in the lookup service might be outdated or incomplete.

It is important that the `TypeSpec` is given by every iterator individually because only the iterator itself knows about its own structural modifications. Based on these (runtime) `TypeSpecs` polymorphic functions can be constructed. Furthermore, it is possible to compute the *outer union* of two collections that have different attributes; for example, two *hotel* data sources on the Internet (e.g., `www.HotelBook.com` and `www.HotelGuide.com`) might have slightly different attributes and it is nevertheless possible in ObjectGlobe to ask a `SELECT *` query that retrieves all attributes from both sources. For simple functions, such as aggregate- or transformer functions (e.g., thumbnail), ObjectGlobe provides a simpler mechanism by "plugging" such functions into generic (built-in) operators.

4.2.4 Secure Query Engine Extensibility

We have utilized Java's security model [Oak98] to guarantee security of ObjectGlobe servers while executing external operators from possibly unknown function providers. Java's five-layer security model is illustrated in Figure 4.9. Java is a strongly typed object-oriented programming language with information hiding. The adherence to typing and information hiding rules are verified by the compiler and again by the class/bytecode-verifier before a Class object is generated from the bytecode because code could be generated by an evil compiler. The class loader's task is to load the bytecode of a class into memory, monitor the loaded code's origin (i.e., its `URL`) and to verify the signature of the authenticated code. The security manager controls the access to safety critical system resources such as the file system, network sockets, peripherals, etc. The security manager is used to create a so-called *sandbox* in which untrusted code is executed. A special, particularly restrictive sandbox is used, for example, by Web browsers to execute Applets. The ObjectGlobe system is based on the latest Java Release 2, in which the Security Manager interfaces with the Access Controller. The Access Controller verifies whether an access to a safety-critical resource is legitimate based on a configurable policy, which is stored in the `PolicyFile`. Privileges can be granted based on the origin of the code and whether or not it is digitally signed (i.e., authenticated) code. In addition, the Access Controller allows to temporarily give classes the ability to perform an action on behalf of a class that might not normally have that ability by marking code as *privileged*. This feature is essential, e.g., for granting access to temporary files as explained below. Finally, the Java program is executed by the interpreter (the

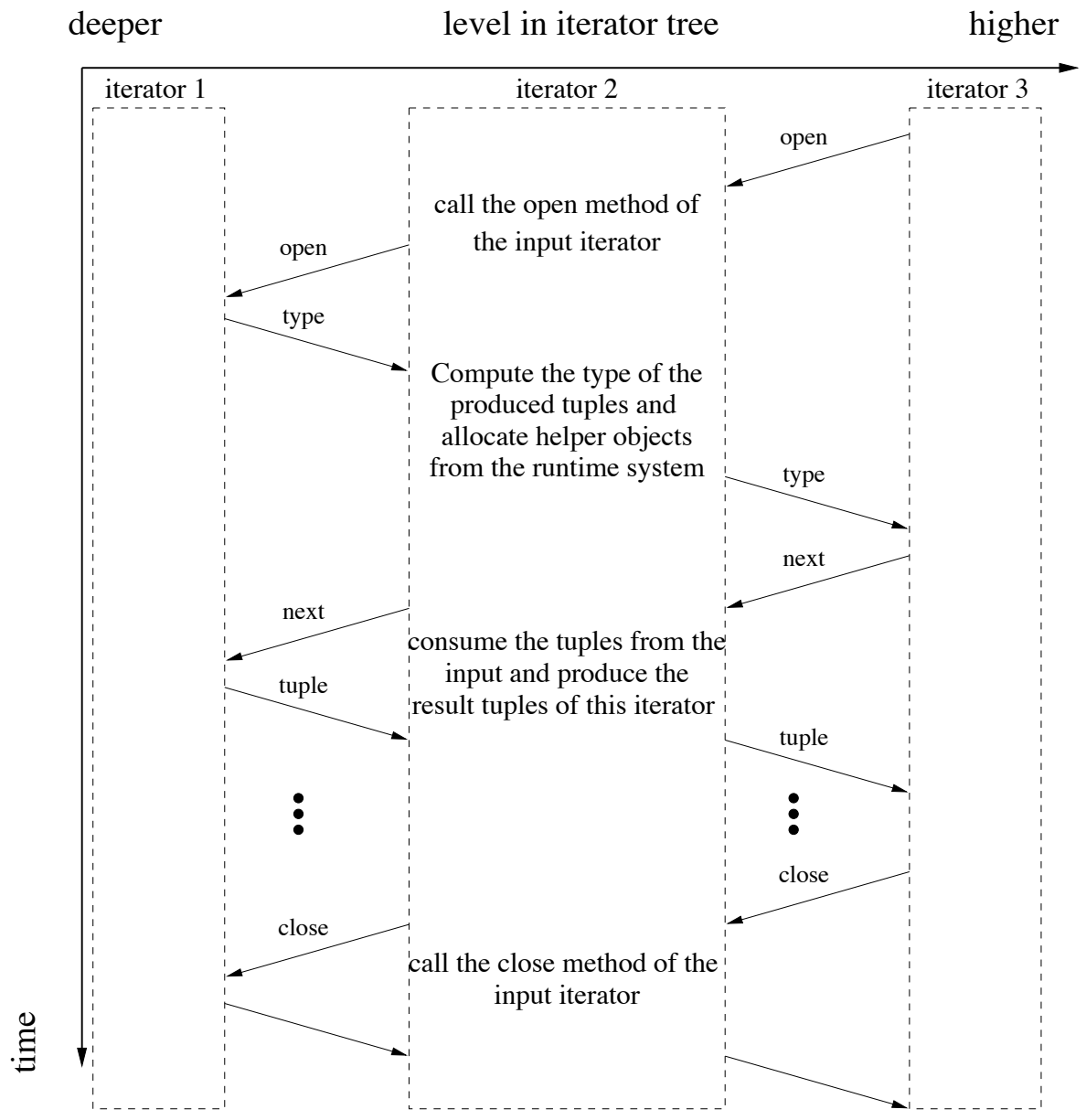


Figure 4.8: A Sample Call-Pattern for the Iterator Interface.

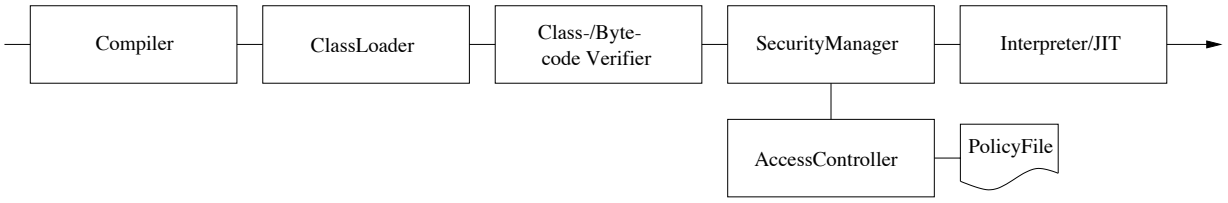


Figure 4.9: Java's Five-Layer Security Model

JVM) which is responsible for runtime enforcement of security by checking array bounds and object casts, among others. From a security perspective, it is irrelevant whether or not parts of the code are compiled by a just-in-time (JIT) compiler to increase performance.

Of course, it would be unreasonable to grant unprotected access to system resources—such as the file system, the network sockets, etc—to unknown code. Therefore, all external operators are executed in a "tight" sandbox. Furthermore, the name spaces of concurrent queries are separated from each other (to be accurate every external operator runs within its own namespace to avoid problems with name clashes and version mismatches). This way it is guaranteed, that they cannot illegitimately exchange information via covert channels ("hidden communication paths"), e.g., via static class variables of external operators. The name space separation is achieved by using a new, dedicated class loader (called *OGClassLoader*) for each query. This class loader is responsible for loading any additional functions beyond the built-in ObjectGlobe classes. The code bases (i.e., the function providers) from which these operators can be loaded are annotated in the query execution plan. Since an external operator could abuse the connection to a function provider as bidirectional communication channel, all (non built-in) classes required by an external operator must be combined into a JAR[6] file. This archive file is loaded and cached by a class loader and the connection to the function provider is closed. All requests to non built-in classes must point to classes in the cached JAR file otherwise they are rejected as illegal. Schematically, the name space separation and the class loaders are illustrated in Figure 4.10.

Some user-defined query operators may require access to the cycle provider's secondary memory in order to store temporary results. Obviously, we cannot generally grant access to the file system to any external operator. Instead, a particular built-in class, called *TmpFile* has to be used. This built-in class provides a safe interface to create a temporary file, to write into and read from the temporary file and to delete the temporary file. Furthermore, a *TmpFile* object ensures the automatic deletion of the corresponding file when it is garbage collected. This way it is guaranteed that external operators can only operate on temporary files that they created themselves (within the same query execution plan). This scenario is illustrated in Figure 4.11.

Access to network sockets is normally prohibited to external operators to prevent them from sending any information about the data they process (to unknown locations). This restriction needs to be relaxed when a cycle provider wants to execute a wrapper which accesses data that is published by, e.g., a Web server. Therefore the policy of the Access Controller can be configured

[6]JAR (java archive) is a platform-independent file format that aggregates many files (compressed) into one (like ZIP) and is supported by the Java Runtime Environment.

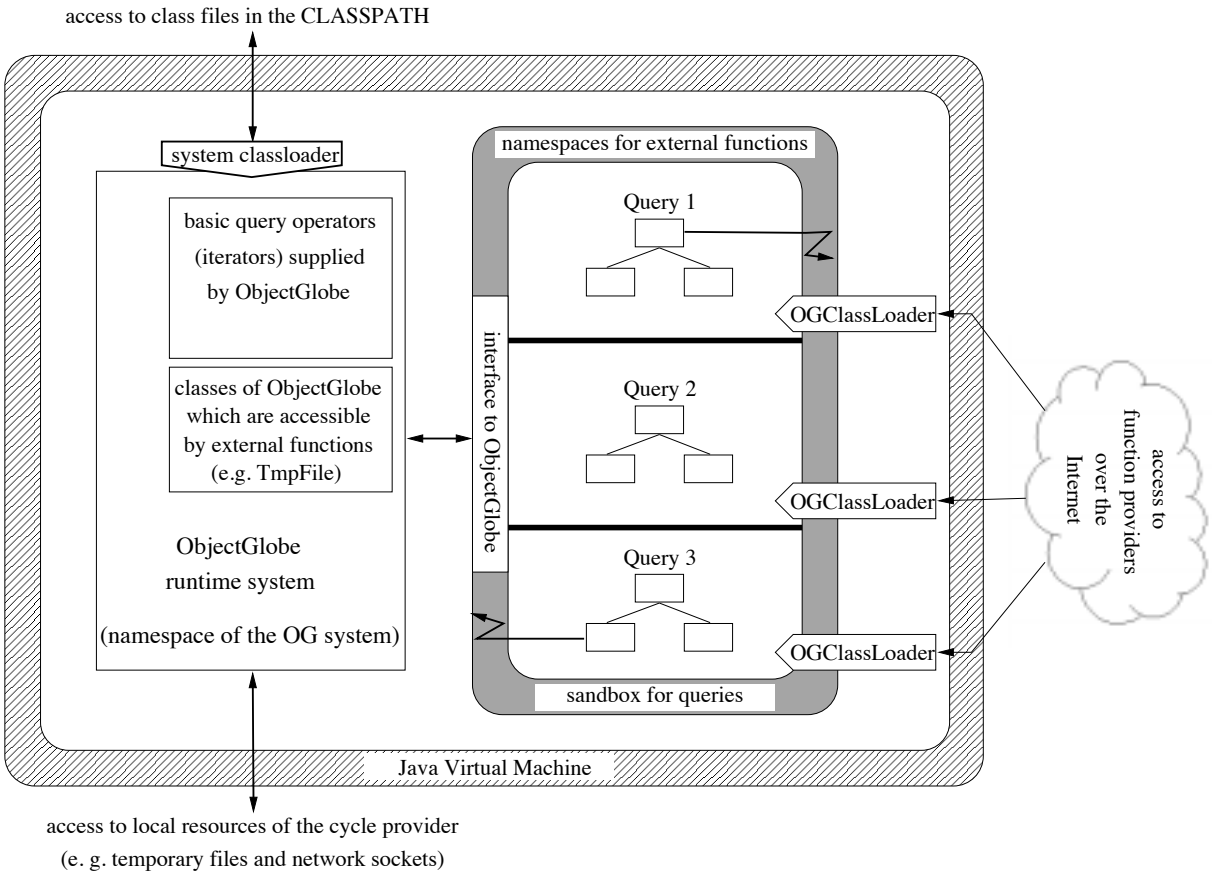


Figure 4.10: Security of Dynamically Loaded Code

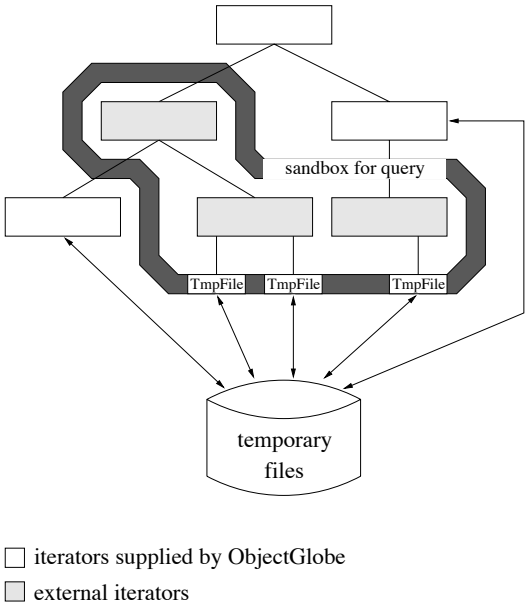


Figure 4.11: Extending Privileged Access Rights to User-Defined Operators

to allow a trusted and authenticated wrapper to establish a connection to a particular host on a given port. It is also possible to configure a relaxed policy that gives this privilege to arbitrary wrappers. The more restrictive policy situation is, for example, suitable for a wrapper accessing an `FTP` server to fetch a file. Granting the right to connect to this server to any external operator would allow operators to store any kind of information at this server, which is certainly not desirable. The more relaxed policy is applicable if granting access to a server is harmless; e.g., access to a server which only sends up-to-date exchange rates for given currencies.

The sandbox security model cannot protect providers from so-called denial of service attacks where malicious code overconsumes CPU cycles or other resources. To protect cycle or data providers from this kind of attack, accounting and authentication can help for identifying intruders. In a work [SBK01] which extends the security system described here, a (system dependent) java library based on the Java Virtual Machine Profiler Interface (JVMPI) [Sun99] is developed. This library keeps account of memory and CPU usage of external operators, other resources like the number of bytes written to secondary memory can be determined using pure Java.

As a part of a general accounting mechanism we will describe our monitor component which is used to control the progress of query operators. This way some simple overconsumption problems, such as operators which maliciously or accidentally consume resources without producing results, can be detected and repaired by halting the query execution.

4.2.5 Monitoring the Progress of Query Execution

Obviously, users want to know if their queries still make any progress at all. The execution of a distributed query can fail for various reasons: network failures, crashed servers, badly pro-

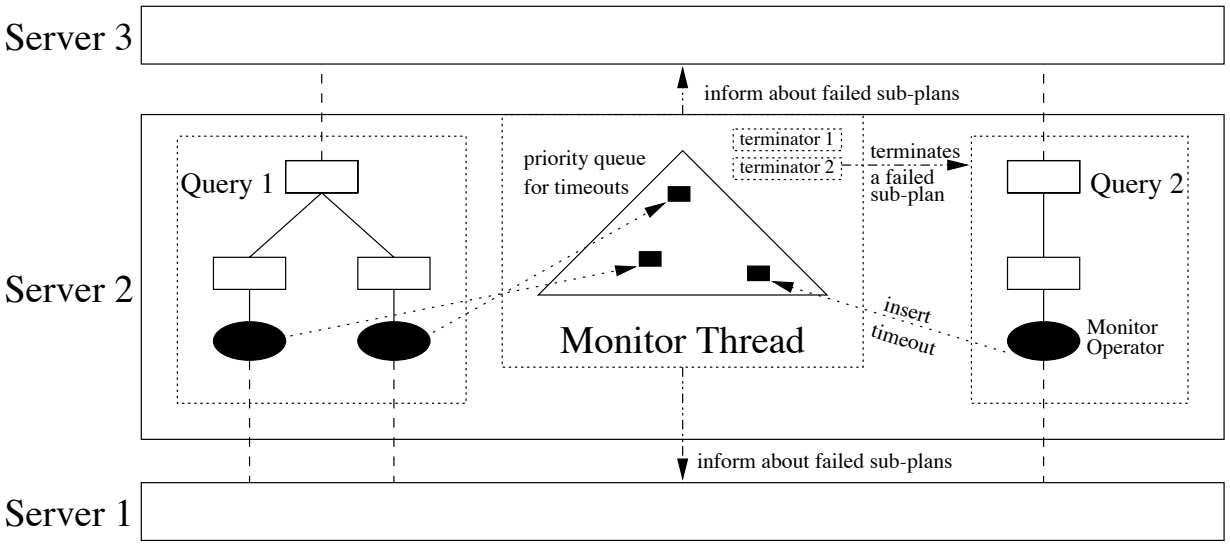


Figure 4.12: The Architecture of the Monitor Component.

grammed external operators, extremely overloaded servers, etc. Without precautions such failures can lead to live- or deadlocked query execution plans, in which upper-level query operators wait indefinitely for blocked sub-plans to deliver their results. Therefore, it is important to monitor the progress of the query execution and inform the participating ObjectGlobe servers about failures.

Each ObjectGlobe server uses a dedicated thread (we call it the *monitor thread*) for detecting timed-out queries. A monitor thread operates on a data structure, which is organized as a priority queue. The objects stored in this queue represent future points in time and the object with the closest point in time has the highest priority. Such an object (we call it a *timeout object*) specifies an event inside a query, which has to occur in that query until the specified point in time has been reached. If its time has expired, the monitor thread removes the timeout object from the queue and checks if the associated event has occurred. If this is the case, the object is discarded and nothing else happens. Otherwise the affected sub-plan of the query is assumed to be blocked and it is terminated by a special "terminator" thread. When a sub-plan is stopped due to an error condition in an operator, the ObjectGlobe servers, executing the operators beneath and above the failed one in the plan hierarchy will be informed about this fact. The sub-plans of the operators below the blocked node will normally fail. The operators above it could react to the failure in special ways (also fail, rearrange the plan, execute an alternative sub-plan, etc. [CD99]). The propagation of an error up the hierarchy is performed by the standard exception handling mechanism of Java "with a little help" from our send-/receive operator pair for crossing network connections. The servers of child operators cannot be informed with the exception mechanism. A special (UDP) network protocol is used for this purpose.

So far we have not mentioned where the timeout objects come from. These objects are created by a special type of operator, the *monitor operator*. A monitor operator can be inserted

at arbitrary positions in a query evaluation plan, since it does not change its input tuple stream. Positions where we will always insert monitor operators are above receive operators and above any external operator. Its task is to observe the progress of the actions performed by the sub-plan beneath. For example, at the beginning of its open method a monitor operator creates a timeout object for the event "end of open reached" and inserts this object into the priority queue of the monitor thread, while also keeping a reference to that object. After that, the open method of its child operator is called. When the method invocation returns, the timeout object is informed, that its awaited event has occurred.

The advantage of this architecture is that the decisions about where to monitor in a query evaluation plan and with what parameters the timeouts should be initialized can be made in a flexible manner. Setting timeouts is critical, just as in any other system. One option is to set the timeout based on the response time estimates of the optimizer. Another option is to use a default value. Other operators and especially external operators need not implement anything for the monitor component. An overview of this architecture is given in Figure 4.12. Monitor operators are not only used for observing the liveliness of a query execution, but also measure the current status of the quality parameters of the execution. This is discussed in detail in Section 7.2.2.

As stated above a silent failure of a query should be made apparent by the monitor component. But what are explicit errors and what do we do with them?

The Java programming language possesses a powerful exception handling mechanism, which is consistently used to inform a program about all errors it could ever see. Naturally we also use exceptions to indicate application specific error conditions, which are propagated from the position where the error occurred up along the call stack until the exception is caught. In an iterator based query processor this means, that error indications are propagated up the iterator hierarchy of the query evaluation plan. This is quite helpful for us, since this enables an iterator to deal with errors, which have occurred during the generation of its input stream(s). For example, a union iterator could just ignore an input stream, if an error occurred during its generation and this iterator could produce its output stream with the other, error free input streams.

Since query evaluation plans in ObjectGlobe are normally distributed, we also have the problem of sending exceptions across network links in order to let an exception be propagated up the operator hierarchy of a plan. Exceptions in Java can only be used local to a thread, therefore a Send-/Receive iterator pair, which constitutes a network connection between a parent- and a child iterator in our system, transports the information about a thrown exception to the server of the parent iterator and re-throws the exception there.

Chapter 5

Performance Experiments

In the following, we describe the results of benchmarks which assess the ability of the ObjectGlobe implementation to perform the intended query processing tasks in a satisfying manner. First, the combination of our optimizer and our external lookup service has to prove that the access of meta-data during optimization does not incur too much overhead. Again, the optimizer has to show that the additional techniques which are introduced to handle environments with a large number of providers, are effective. One prominent feature of ObjectGlobe is operator mobility and its potential is shown here in a benchmark. At last, the mechanisms for securing network connections and for loading external operators are examined for their performance impacts.

5.1 Overheads of Plan Generation

To determine the overheads of plan generation, we measured the *lookup* and *optimize* steps of processing a five-way join query[1]. The optimizer ran on a Sun Ultra 10 workstation; the lookup service ran on a Sun Ultra 1 workstation. There were six relevant cycle providers and the optimizer considered three different join variants (nested-loops, hash, and sort-merge). We studied two different scenarios. In Scenario I, all joins could be executed at all cycle providers; in Scenario II, joins with two of the five collections could only be executed at one specific cycle provider. Table 5.1 summarizes the results. Even though the meta-database of the lookup service is very small, most of the time is consumed in the lookup step; the reason is that twelve search requests are required for this query and the overhead of each search request is rather high. The optimization time is acceptable in this experiment ($<$ 1 sec). The optimization time is much lower for Scenario II than for Scenario I because the search space is much smaller for Scenario II due to the authorization restrictions.

[1]The benchmarks in this section were performed by an older optimizer implemented in C++. The optimizer benchmarks in the following section were performed by our new optimizer which is implemented in Java.

	Total Lookup Time	Avg. Time per Search	Optimization Time
Scenario I	5.64 secs	0.47 secs	0.83 secs
Scenario II	5.64 secs	0.47 secs	0.07 secs

Table 5.1: Overheads of Plan Generation

5.2 Using a Cluster Tree for Optimization

The modified optimization algorithms which were introduced in Section 4.1 should produce compact and parallel-working QEPs in large environments with a great number of cycle and data providers. In order to test these algorithms we implemented a meta-data generator which can produce the meta-data for such environments. This generator program takes several parameters, for example,

- the number of cycle providers,
- the number of data providers,
- the minimum bandwidth/maximum latency between two hosts in the environment, and
- the number of pools where providers are grouped.

We use the term *pool* for collections of providers which have network connections to one another with a better bandwidth/latency than the average network connection. These pools are counterparts to local area networks, research networks, metropolitan area networks and company intra-nets which appear at the leafs in the somewhat hierarchically structured Internet and also show better performance properties than the rest of the Internet. The sizes of pools and the quality of their internal network connections are varied randomly within some limits which can be predefined. A user-defined percentage of all providers are placed randomly in such pools and the remaining providers are randomly placed outside these pools. Analogously, the relative quality of network connections between pools are also determined randomly.

Data collections for themes are also randomly generated and placed on data providers. The user can steer this process by providing values for

- the number of themes,
- the number of tuples for a specific theme,
- the size of a tuple for a theme (in bytes), and
- the number of collections for a specific theme.

In the following, we present results from benchmarks which used an environment with 1000 data providers, 100 cycle providers and 50 pools. 200 providers were placed outside a pool, the themes varied in size from 0.7 to 17 MBytes and in the number of tuples from 2000 to 400000.

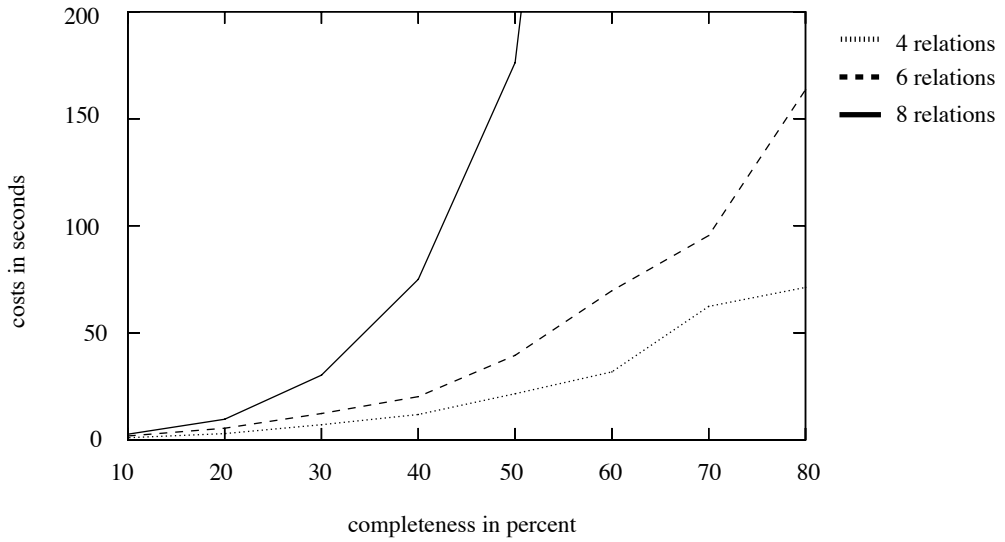


Figure 5.1: The Impact of the Completeness Constraint on Plan Quality.

The queries which are passed to the optimizer are also randomly generated. The constraints for the generated queries are that there are 100 tuples in the result of the query, 50% of the join operations are followed by a projection and for every third theme an aggregation operation is used which decreases the size to about 20% of the original data. The benchmarks were conducted on a Sun Enterprise 450 with 4 GBytes of main memory and four 400 MHz processors. The optimizer internally does not use more than one thread which means that just one of the processors is used. Every point in the following graphs represents the mean value of 20 experiments, each conducted in a newly created scenario.

First, it is interesting to see, if our optimizer can take advantage of a lower demand for the completeness of themes in a query. The cluster tree is utilized during optimization to find a more local and hence faster QEP for a query. This task should be easier with more relaxed completeness constraints since the optimizer then can choose between a greater number of data collection subsets. In Figure 5.1 exactly this effect can be seen for the dynamic programming optimizer with cluster tree support. The response time for queries with a more relaxed completeness constraint is much lower. Note that this effect cannot only be caused by the reduced amount of data which has to be processed, since we use a response time cost model [GHK92] and in our generated environments, just as in the Internet today, the network transfer times are the dominant cost factors. Therefore, the distances of the used cycle and data providers to one another are the critical parameters which has to be optimized. The corresponding experiments with the greedy optimizer and the dynamic programming and greedy variants with recursive partitioning of intermediate results produced similar results and are not shown here.

In the Figures 5.2 and 5.3 the dynamic programming (denoted as DP in the figures) and greedy (GR) optimizers and the corresponding variants with recursive partitioning of intermediate results (RP-DP and RP-GR) are compared regarding the quality of the plans they produce

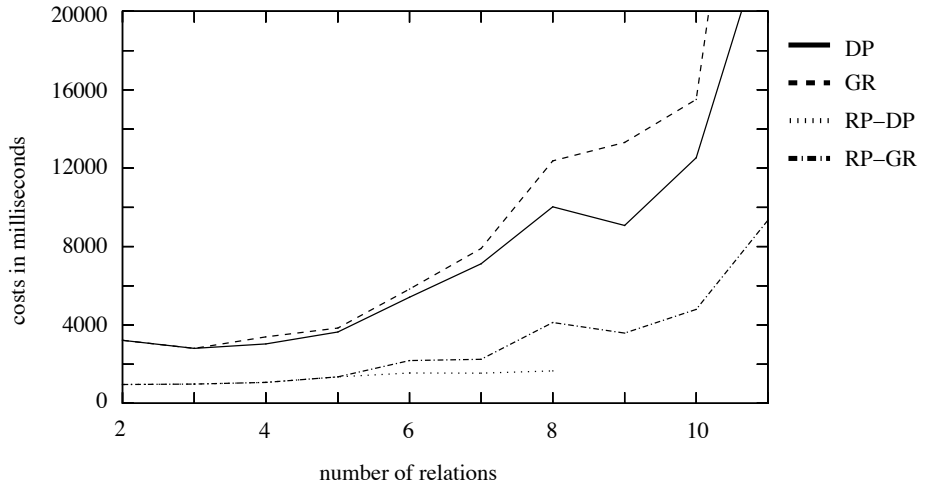


Figure 5.2: Quality of different Optimizer Configurations.

and the time the optimizers are consuming. Naturally, greedy optimizers consume less time than the corresponding dynamic programming counterparts, but they also produce plans with a lower quality. This is a well known result and it is more interesting to study the behavior of the recursive partitioning versions. The dynamic programming algorithm with recursive partitioning was only tested for up to 8 themes in this benchmark because for more themes its running time gets rather high. We see that additionally to the selection of providers with the cluster tree, when we also use the structure of this tree to guide the construction of a QEP, the resulting dynamic programming version produces better results than the version without recursive partitioning. The same holds for the greedy algorithms. On the other hand, the recursive partitioning technique results in increased optimization times since more cycle providers have to be selected for the additional join operations on the partitions. Furthermore, it is interesting here that the recursive partitioning greedy algorithm can compete with the standard dynamic programming algorithm with respect to the running time of the optimizer and especially with respect to the quality of the plans. But we must note that the greedy algorithms more likely produce outliers in the form of very bad plans than the dynamic programming algorithms which show a more stable behavior in this respect. At the end, we can say that none of the examined optimization algorithm is a clear winner. Similar to standard database systems, the choice of the optimization algorithm depends on the situation. For example, when a large number of themes is involved in an ad-hoc query the standard greedy algorithm should be used in order to obtain acceptable optimization times. If a query will run several times the recursive partitioning dynamic programming algorithm can justify its own running time by producing a high quality QEP. For the cases in between these two extremes the other two optimization algorithms can represent a good compromise between plan quality and optimization effort.

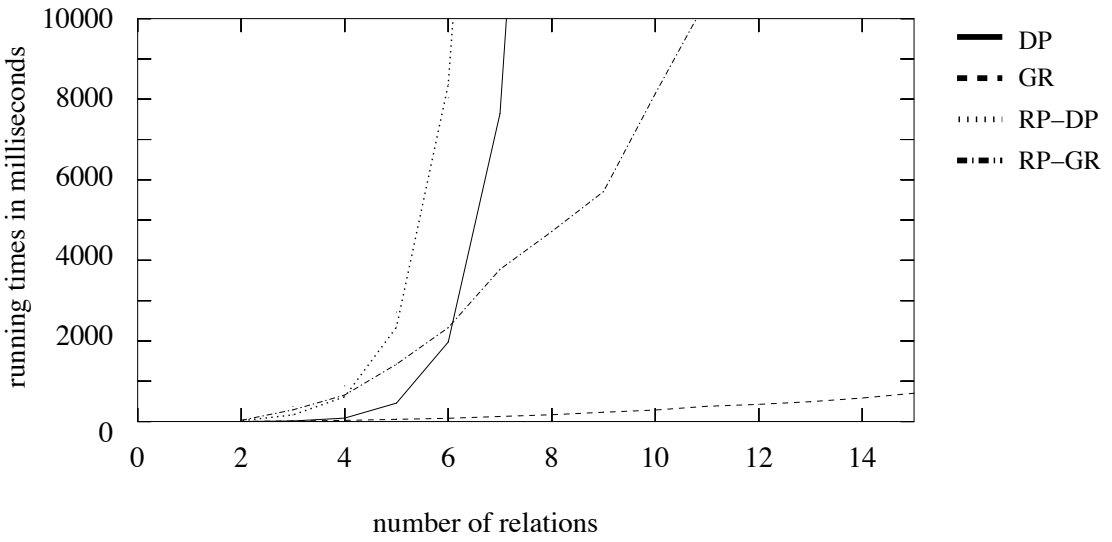


Figure 5.3: Running Times of different Optimizer Configurations.

5.3 Query Execution Times

5.3.1 Benefits of Operator Mobility

The following experiment shows the benefits of ObjectGlobe's ability to execute query operators near data sources. We measured the execution time of a query which determines the hotel in Berlin with the greatest number of hotel rooms. The information about hotels is gathered from two Internet sites namely HotelBook (`www.hotelbook.com`) and HotelGuide (`www.hotelguide.com`). To perform this task wrappers were used which first query a list of all hotels in a given city and afterwards query detailed information for every single hotel in this list; according to the query capabilities of the data sources. We measured two different plans for this query, which structurally correspond to the plans shown in Figure 3.4 and Figure 3.5, except that we use a group operator instead of a nearest neighbor operator. The traditional one is to execute the wrappers at the client in Passau, the other one which is made available by ObjectGlobe is to execute the wrappers and intermediate group operators at a cycle provider near the data sources. Because it is impossible to execute the wrapper at the hosts serving HotelBook or HotelGuide, we used a host in Maryland for this experiment.

We executed these two plans every two hours in a 24 hour range and as the results in Figure 5.4 show that there is a clear benefit if the wrappers are executed near the data sources, i.e., at a cycle provider with a good network connection to the data sources. Therefore the latency time is reduced when the wrapper iteratively accesses the HotelBook or the HotelGuide database. This experiment does not demonstrate how parallelism can be used to speed up query execution, because the network costs dominated the CPU costs by far, but performance gains from parallelism can also be achieved with ObjectGlobe.

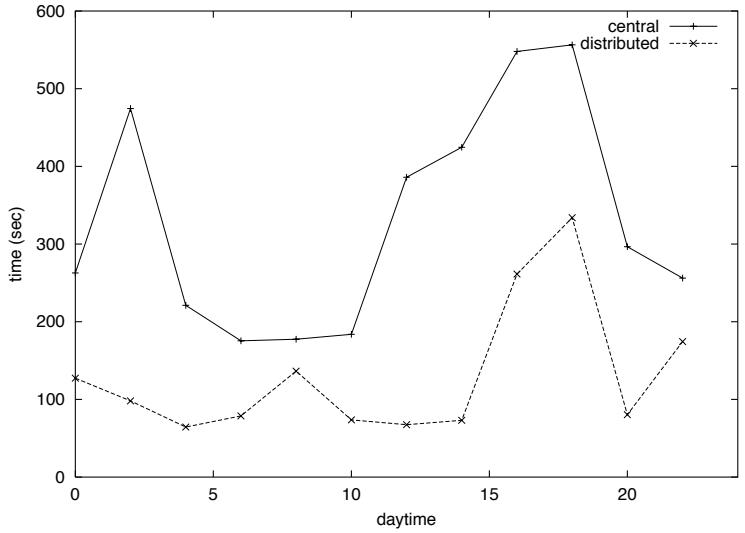


Figure 5.4: Centralized versus Distributed Execution of Plans

5.3.2 Costs of Secure Communication

The use of SSL sockets [FKK96] and therewith encryption and Message Authentication Codes (MACs) is an effective way to integrate secure communication into a distributed system. But cryptographic algorithms have additional costs when transmitting data across a network. To demonstrate this effect we executed a simple scan-display plan and varied sites of the scan operator and the usage of SSL. In all cases the scan operator had to process 10 MB of data. As Table 5.2 illustrates, costs for encryption and MAC calculation can be neglected in a WAN environment. The first column contains information about where the scan and the display operators were executed[2] and across what kind of network the data was sent. The remaining three columns list the times of query executions where the data was not encrypted and no MAC was calculated (plain), where only a MAC was calculated (SHA) and where both, encryption and MAC calculation, were done (IDEA + SHA). The first row shows that secure communication increases the query execution time in LAN environments (but the overall execution time is even with fully secured communication much faster than query executions in a WAN environment with unsecure communication). The second row shows that in a WAN environment there is no significant time difference between secure and insecure query execution because costs for cryptographic algorithms are CPU costs and are superimposed by communication costs.

5.3.3 Costs of Dynamic Extensibility

One of the prominent features of ObjectGlobe is its dynamic extensibility by external operators. There are of course additional costs caused by loading classes from the network and the separation of name spaces of different queries compared to loading locally available built-in op-

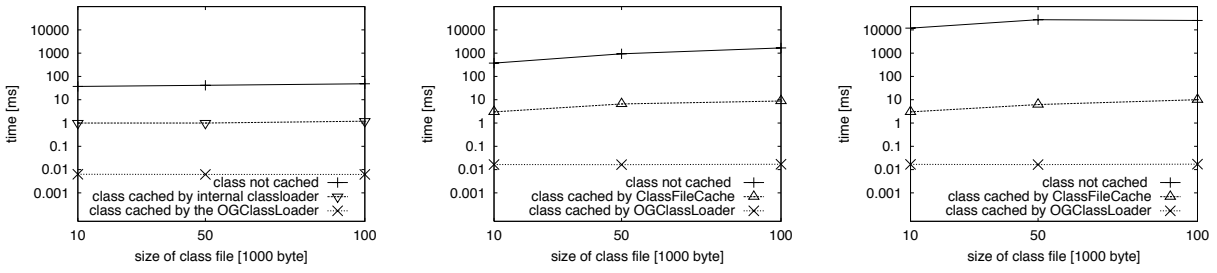
[2] X $\rightarrow$ Y means that the scan operator was executed at host X and the display operator was executed on host Y.

	plain	SHA	IDEA + SHA
scan [Passau → Passau], 100 MBit LAN	3.54 secs	5.31 secs	11.86 secs
scan [Mannheim → Passau], WAN	81.93 secs	81.86 secs	82.04 secs

Table 5.2: Costs of Secure Communication in Different Network Environments

erators. This separation of name spaces is achieved by using an individual OGClassLoader for every query and it forbids the caching of Class objects for external operators. Instead, only the bytecode (rather than the instantiated class object) of an external operator can be cached and this bytecode is cached in a separate ClassFileCache. To measure the overheads of loading an operator from a remote site and from the ClassFileCache, we loaded built-in and external operators of different size stored at different locations using our OGClassLoader: built-in operators from disk and external operators from a local function provider in Passau and a remote function provider in Maryland. For external operators, we measure three scenarios: (a) the bytecode is not cached at all; (b) the bytecode is cached in the ClassFileCache; (c) the operator is cached as a class object internally in the OGClassLoader. Scenario (c) is used as a baseline and simulates the behavior of a system without security measures. Figure 5.5 shows the following effects:

- The costs for the initial loading of a class from disk or network are very high (the +-lines in Figure 5.5) but can be heavily reduced by caching the class object of built-in operators or caching the bytecode of external operators (the triangle lines).
- Comparing the ×-lines (Scenario (c)) and triangle lines (Scenario (b)), we see that the overheads to ensure security are relatively high; compared to the overall costs of query processing on the Internet, however, the overheads for security can usually be neglected (less than a second in all cases).



a) Built-In Operator b) Ext. Op./Local Prov. c) Ext. Op./Remote Prov.

Figure 5.5: Costs of Loading an Operator by the ObjectGlobe Class Loader

Chapter 6

QoS in Data Integration Systems

In this chapter, we motivate, why the support for QoS mechanisms in a data integration system like ObjectGlobe is necessary in order to increase its user acceptance. After that, we list related work in the area of QoS management. In the last section, we describe the QoS parameters which can be used in our system to specify QoS constraints and explain the basic aims and strategies of our QoS management system.

6.1 The Relevance of QoS for Data Integration Systems

Accessing and processing distributed data sources have become important factors for businesses today. This is especially true for the emerging virtual enterprises with their data and processing capabilities spread across the Internet. Unfortunately, however, query processing on the Internet is not predictable and robust enough to meet the requirements of many business applications. For instance, the response time of a query can be unexpectedly high; or the monetary cost might be too high if the partners charge for the usage of their data or processing capabilities; or the result of the query might be useless because it is based on outdated data or only on parts (rather than all) the available data. Here, we show how our ObjectGlobe system can be extended in order to support quality of service (QoS) guarantees. We propose ways to integrate QoS management in the various phases of query processing: (1) Query optimization uses a multi-dimensional assessment (cost, time and result quality) of query plans, (2) query plan instantiation comprises an admission control for sub-plans and (3) during query plan execution the QoS of the query is monitored and a fuzzy controller initiates repairing actions if needed. The goal of our work is to provide an initial step towards QoS management in distributed query processing systems and do significantly better than current distributed database systems which are based on a best-effort policy.

Our ObjectGlobe system with its distinction of cycle, data and function providers, enables an open and distributed query processing services market. This market can also be regarded as an information economy. In order to become commercially relevant, however, it is necessary to give guarantees on the services provided. Today, almost all open systems on the Internet are based on the "best-effort-principle" and nobody is willing to construct mission-critical applications in such

an environment because such applications would simply not be reliable enough. Specifically, users would like to constrain the cost of running applications, the running times of applications, and the quality of the results obtained by running the applications using external data sources. To demonstrate these needs we give the following example:

> A realtor in the USA has an appointment with a customer, who wants to buy a villa in the Mediterranean area. The customer has mentioned his requirements regarding the maximum distance to the next airport and the amount of building area in advance. The realtor poses a query against a distributed query processing system which covers some European realtors as data providers. There exists also a data provider which has information about airports in that area. Since the requirements of the customer seem to be very selective the realtor requests that at least 70% of the registered data about estate offers should be considered in the query and at least 20 result tuples should be produced. The appointment is in 20 minutes; therefore the data should be available in no more than 10 minutes since the realtor wants to check the offers before. Considering the budget for IT-services and the prospects of a successful deal, the realtor sets the upper bound for the query execution cost to 10 Euros.

The SQL-query for this application computes a join between the real estate information and the data about airports. The join predicate uses a user-defined, external function which computes the length of the bee-line between two locations. The query uses another external function for scaling the images of the estate offers into a handy size. We used the real estate DTD [Pet99] as a template for our real estate relation; the meaning of the attributes should be obvious.

```
select e.Price, e.Location.City, scale(e.Image,0.3), a.Name
from Estate e, Airports a
where e.building-area > 200 and
      geoDistance(e.Location,a.Location) < 10 and
      e.Location.region = 'Mediterranean';
```

The data of each European realtor represents a partition of a relation named *Estate*. For the *Airports* relation we assume that there is only one data provider; therefore it does not make sense to specify a completeness constraint for this relation. This means, that implicitly 100% completeness is requested for the *Airports* relation. The quality constraints given in the textual description are:

- completeness of the used real estate information $\geq 70\%$.
- cardinality of query result ≥ 20.
- total cost ≤ 10 Euros.
- total response time ≤ 10 minutes.

A possible query evaluation plan for the given query is depicted in Figure 6.1. The leaves of the operator tree represent autonomous data providers which contribute their information to the query. The plan fragment consisting of the `union` and `scale` operation may be executed

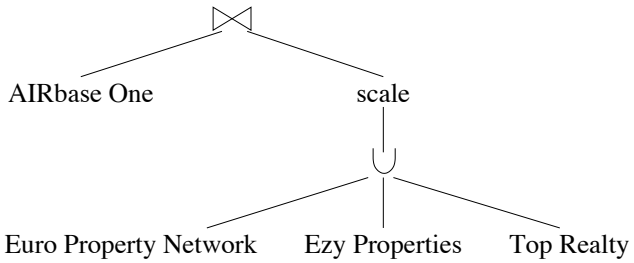


Figure 6.1: Query Evaluation Plan for the Example Query

at a cycle provider which has a good network connection to the data providers beneath it. The `scale` operator itself could be loaded from a function provider, specialized on functions for image processing.

This example demonstrates the need to give guarantees at the "application level" and that this involves coordinating the services of several providers. Obviously, it is not possible to fulfill all requests in such an environment since servers might be down, interconnects might be congested, or simply because the right data providers might not be found. However, the goal should be to fulfill as many requests as possible and to abort and inform the user as early as possible if a request cannot be fulfilled. This is the challenge we would like to address in the remainder of this work. Specifically, we will pick up this challenge for query processing as required in this example.

6.2 Related Work

The term "Quality of Service" is mainly used in the area of networked multimedia applications. These applications need more restrictive constraints for the service quality they get offered by the underlying devices in order to, for example, provide for smooth video playing. For multimedia data streams, precautions have to be taken in the server and the client machines as well as in the network infrastructure. Frameworks for managing QoS in a distributed multimedia application are presented, for example, in [LN99] or [ACH98]. In respect to our work, it should be noted that the nature of QoS parameters in multimedia applications and distributed query processing differ substantially. For multimedia applications the QoS parameters mainly constrain the current execution of the application (Do we provide smooth playing at the moment?), whereas for query processing most QoS parameters refer to the end of the execution (Is the result set sufficiently large?).

[Wei99] gives a good motivation for the need to integrate the handling of service quality guarantees in information systems. Of the many aspects discussed in [Wei99] we concentrate on the quality of service in distributed query processing over autonomous data- and cycle providers on the Internet. Work on service quality in the area of query processing is rare and the existing work particularly concentrates on constraints for the response time. This can be seen in papers about query scheduling like [BLR98], [GI97] or [BMCL94] and also in the area of real-time databases [PCL95]. In many cases constraints cannot be defined for a single query but a query has

to be assigned to a query class with predefined constraints on response time. Query processing in an open data integration system requires to also consider result quality and execution costs and the definition of QoS constraints should be possible for single queries. In addition, a novel aspect of our work is that the execution of a query utilizes the services of several autonomous providers. The difficulty here is to find a plan for the query which uses these services in a way so that the user-defined QoS constraints are fulfilled.

One aspect of QoS in a data integration system is certainly data quality. This topic has already been tackled in the literature, for example, in [FKL97] and [NLF99]. The reliability of a data provider, the accuracy of the corresponding data and its relevancy are example data quality aspects. Data quality constraints for a query can be considered in a pre-selection phase for data providers. In our work we also consider data quality constraints but we concentrate on the interference of these constraints with those for costs and performance and how these interferences can be treated in query optimization, distribution and execution. This means that we deal with data quality constraints in another context than the existing work which concentrates on the selection of data providers.

Our work is related to the Mariposa project [SAL+96]. In Mariposa a user can constrain the ratio between the running time of a query and its execution cost by the means of a bidding curve. Mariposa tries to obey these constraints during its plan fragmentation step which takes place after optimization. In our work, we consider the user-defined quality constraints during all phases of query processing. We think that this is necessary for achieving high rates of quality conforming query executions in a distributed and open environment.

During query execution we use adaptations of the query in order to react to violations in its QoS constraints. There is a great deal of work on adaptations of queries during execution time. [GW89] and [INSS92] propose the generation of query execution plans with embedded alternative sub-plans. The decision for a specific plan configuration is made at execution time, depending on the current load situation. In [KD98] and [IFF+99] query execution steps are mixed with re-optimization steps. Query scrambling [UFA98] tries to hide delays in data delivery of remote data sources by re-scheduling executable sub-plans of the query. The authors of [AZ96] suggest to start competing sub-plans in parallel and after a winner has emerged, the “losers” are stopped. A more recent work on runtime adaptation can be found in [AH00]. These authors propose to decide for each tuple and for every of its processing steps by which operator in a pipeline it should be processed next. This decision is based on the back pressure observed at the queues which are associated with every operator.

Our adaptations supplement those above because we mainly focus on resource allocation. But this does not mean, that we only adjust local settings like CPU priority or memory allocation. In our environment, cycle, data, and function providers are also resources. For example, if a cycle provider becomes a bottleneck and, as a consequence a query seems to miss its guarantees on response time, we could move the corresponding plan fragment to another, more suitable cycle provider. The logical structure of the plan is not changed in this way.

6.3 The Quality of Service Model

Analogously to cost models in traditional database systems, we need a model for our quality constraints in order to describe and assess the quality of queries, query evaluation plans and query executions. The basics of this QoS model and its role for query processing are presented in this section.

6.3.1 The Quality of Service Dimensions

As we have seen in Section 6.1, in an information economy a user should be able to constrain the relationship between the qualities of the result and the execution itself and the costs for the execution. Therefore, we need a set of QoS parameters which can be used for declaratively specifying QoS constraints for a query execution. In the query processing context QoS parameters can belong to three different dimensions: the result quality of the query, the duration/timeliness of the query execution and its monetary cost.

Query Result Quality We assume that a relation[1] S is divided into several partitions $S_1, \ldots, S_n$ which may be managed by independent data providers. In our example, every realtor participating in the data integration system, represents a data provider which contributes a partition of the relation *Estate*. Usually only a subset of these partitions is used in a specific query. For each relation S in a query, we can constrain the following QoS parameters:

- the oldest time stamp of the last update for a partition p of S or its maximum staleness factor as introduced in Mariposa. We call this parameter QR^p_{age}.
- the share of the used partitions in respect to the whole data of the relation S (QR^S_{comp}). This parameter is called completeness later on.

The result cardinality can be characterized by

- a lower bound for the result cardinality ($QR_{min\#}$). This parameter can be used to express that the user expects a minimum number of result tuples. If a query uses just a share of the available data for a relation, the system should incorporate as much data into query processing as is necessary to accomplish at least this result cardinality. If all the data for all relations in a query is used, this parameter need not be regarded anymore.
- an upper bound for the result cardinality ($QR_{max\#}$). Such a parameter corresponds to a *stop after* clause, whose support in query optimization and execution has already been studied in the literature [CK98]. A *stop after* clause will normally be used only in conjunction with an *order by* on the result.

[1] The QoS techniques developed here are not restricted to data integration systems. Therefore, we use standard database terminology and use the term relation instead of theme and partition instead of data collection in the following.

Query Execution Time The execution of the query is characterized by the time spent in different phases of the execution of a plan:

- the time spent in the `open`-phase of a plan (QT_{first}). In an iterator based [Gra93] query engine like ours, this is roughly the time from the start of the query execution until the first tuple can be delivered.
- the time for producing all the result tuples of the plan—called QT_{last}—starting at the point, when the open-phase has finished and ending, when the last tuple has been produced by the query. This phase is called the `next`-phase.

A small QT_{first} value for the resulting query execution is important in a distributed environment because the user can already look at the first tuples whereas the remainder of the (maybe long lasting) query is still executed in the background.

Query Evaluation Cost Since providers can charge for their services, a user should be able to specify an upper bound for the respective consumption by a query. Therefore, the quality parameters regarding the cost of a query take into account:

- the cost for services of function providers ($QC_{function}$), i.e., the cost for leasing a function for the duration of the query.
- the cost for services of data providers (QC_{data}), i.e., the cost for reading the data at the respective data providers.
- the cost for services of cycle providers (QC_{cycle}), i.e., the cost for executing parts of the query at foreign cycle providers.

In many cases not all quality parameters will be interesting for a user or perhaps just the sum of some of them, like the total cost or the total response time. The quality constraints for a specific quality parameter could also be expressed in the form of a continuous function over the space given by some or all other quality parameters. For instance, the user is willing to pay more money for the query execution, if more data was incorporated into the query processing. In our example, the realtor posed constraints on the total cost, total response time and the completeness of the *Estate* relation. Similar to real-time systems (hard real-time, soft real-time) some constraints on quality parameters could be strict and others could be handled in a relaxed way, by not aborting the query, if the constraints are not fulfilled any longer during execution. For example, users could limit the total response time for their queries, but they may still be interested in the result even if it takes longer. For batch queries users might only be interested in the total cost and the quality of the result because they do not wait for the completion of the query.

6.3.2 The Integration of QoS Management in Query Processing

It is currently not possible to construct a distributed and open query processing system on the Internet which can enforce the quality constraints of an accepted query under all circumstances.

The corresponding obstacles are either caused by the environment or by system immanent inaccuracies. Environmental factors, like network failures and unforeseeable load fluctuations on network links and cycle providers can obviously inhibit a QoS conform execution of a query. Likewise, difficulties in producing exact formulas for quality estimations and the limited accuracy of statistics for attribute value distributions of partitions or load distributions of resources can lead to overestimations of quality parameters. In this section, we first describe and motivate the overall goal of the QoS management component in our query processing systems. After that, we give an overview of the responsibilities and tasks of our QoS management component.

The Goal of QoS Management

Naturally, the ability to guarantee QoS constraints depends on the service quality guarantees one can get from the underlying shared resources. Among the common scheduling disciplines *best effort*, *priority-based scheduling*, and *reservation*, only the latter allows to construct a QoS management which can absolutely guarantee the QoS constraints for an accepted query. But reservation normally entails over-booking and inefficient resource utilization and is therefore rarely used for scheduling computer or network devices. Due to this fact, there remain two obvious goals for QoS management in our context:

- The percentage of queries, whose quality constraints could be fulfilled, should be maximized. This percentage is calculated based on the overall number of queries which are issued and not only on the number of those for which a constraint compliant query plan could be determined.
- The execution of queries which cannot fulfill their QoS constraints, should be stopped as early as possible. In this way the query does not waste the time an money of the user anymore. Of course, if the missed quality constraint is a soft one, the query should not be stopped but executed in a best-effort manner.

A query can only meet its quality constraints, if it gets a sufficiently good service from all involved providers. The difficulty in achieving a high percentage of QoS compliant queries is to find at optimization time a query plan which uses providers which can provide for a sufficiently good service at execution time. The optimizer uses estimates about the providers to construct such a query plan and the question is now, whether these estimates also hold during execution. There are two possible solutions for obtaining these estimates:

- The optimizer could initiate a resource availability test for each of the resources during optimization. Since we are expecting such a data integration system to consist of several thousands of providers and a lot of these would have to be regarded during plan generation, this approach is obviously not scalable enough.
- The remaining approach is then, to gather statistics about resource availability which is used during plan generation to estimate the current values for the resources in question.

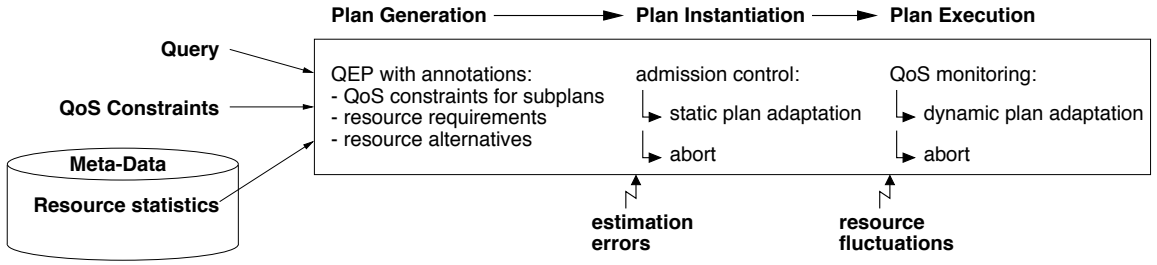


Figure 6.2: The Interaction of Query Processing and QoS Management

In the latter approach imprecise estimations for the resource availability together with all the other problems mentioned at the beginning of Section 6.3.2 could lead to failed quality constraints at run-time. In order to avoid a restrictive admission control which could reduce the number of failed executions, we exploit that different queries often have different demands at specific providers (e.g., batch queries in contrast to interactive queries). For every involved provider we explicitly state these demands in the form of resource requirements and quality constraints in the query plan. During query instantiation and execution we can use this information in order to check, if the demands of a query can be satisfied, and we can react appropriately. For example, if a query seems to miss its quality constraints, it could try to get a greater share on the resources of the provider at the expense of queries which can work sufficiently with a smaller share. If such an adaptation is not possible and the query will not fulfill its QoS constraints our second goal rules to stop the query in order to save the time and money of the user.

The Phases of Query Processing

An overview of processing a query in the context of our QoS management is depicted in Figure 6.2. The starting point for query processing in our system is given by the description of the query itself, the QoS constraints for it and statistics about the resources which can be used for processing the query. In our scenario cycle providers, the partitions from data providers and the functions of function providers belong to these resources together with all the network links connecting them.

Figure 6.2 also shows the activities of our QoS management during the query processing phases of plan generation, instantiation and execution.

Plan Generation: The optimizer generates a query evaluation plan (QEP) which contains information about the used data, cycle and function providers and about the way their services are combined to compute a specific query. The optimizer itself is in essential an intelligent search routine which is in many cases and also in ours dynamic programming based. It assesses a huge number of different plans with different providers by a quality model which provides formulas for estimating the quality parameters based on the structure of the plan and statistics about the providers. Hereby, every considered plan is constructed piecewise in a bottom-up manner and for every sub-plan which appears in such a process its quality parameters are computed with the quality model. Only a plan which fulfills all the user

constraints is considered for the later phases and its plan description will be annotated with the quality estimations and resource requirements for every sub-plan. Additionally, if the optimizer can find approximately equivalent alternatives for resources used in the query evaluation plan, these are also annotated in the plan.

Plan Instantiation: During plan instantiation, sub-plans are distributed to cycle providers, functions are loaded from function providers and connections to data providers are established. When a sub-plan of a query uses the service of a specific provider, it is checked, if the resource requirements resulting from the quality constraints for that sub-plan can be satisfied by this provider. For a cycle provider this would mean that if the optimizer underestimated the load on the respective cycle provider the newly arrived sub-plan will probably not be able to meet its constraints. Furthermore, all the other sub-plans on that cycle provider would be in danger of missing their quality constraints, if we execute the new sub-plan there. As a result of this admission control, the execution of the new sub-plan would be rejected or, if possible, the sub-plan will be adapted.

Plan Execution: During query execution, fluctuations regarding resource availability for, e.g., CPU time or network bandwidth and again estimation errors by the optimizer might violate the constraints on the quality parameters. In order to detect these violations, a monitoring component traces the current status of these parameters for every relevant sub-plan of the query. If this component detects a potential violation of the quality constraints for a sub-plan, it first tries to adapt the sub-plan so that it will meet its constraints again, or if this is not possible, it will abort the execution of this sub-plan. The plan adaptations during the instantiation phase can be performed rather easily because the plan is not instantiated yet. Here we need adaptations which can be applied also after a sub-plan has started to execute.

By estimating the necessary quality constraints for sub-plans of a QoS compliant query plan the QoS management can monitor the development of the quality parameters on a fine granularity. This helps in detecting potential quality misses quite early. For example, if there are pipeline-breaking operators in the plan, most of the work for the query could have already been done, when the first tuples arrive at the top of the plan. In our case, the plan beneath the pipeline breaker has its own quality constraints which must be monitored and enforced by the QoS management.

Chapter 7

Enforcement of QoS Constraints

The enforcement of QoS constraints needs to be considered during query optimization, distribution and execution. Each of these steps with respect to the QoS enforcement builds upon the achievements of the former one or relys to some extent on the abilities of the next one. The corresponding techniques for QoS enforcement in each of these steps are shown in the following.

7.1 Quality of Service Enhanced Plan Generation

In this section we discuss the necessary modifications of a classic, dynamic programming based query optimizer for supporting QoS constraints during plan generation. We concentrate on the description of those parts of the optimization process which play an important role for QoS management and thus need modifications compared to their standard form.

- In many cases, it will not be feasible to consider all possible data providers for a given data set because the resulting amount of data processed in a widely distributed environment would result in unacceptable running times. Thus, an additional task for QoS management is to select the most relevant data providers and find appropriate cycle providers which are able to efficiently process the data from the selected data providers. This aspect of query optimization in ObjectGlobe has already been presented in Section 4.1 and we concentrate here on the additional information which has to be produced in order to support admission control and runtime adaptations.

- Query evaluation plans are constructed in a modular way using basic operators such as join, union, or user-defined operators. Analogously, the cost (and other properties) of a plans are computed in a modular way using cost functions for the individual operators. For QoS management, an extended framework is needed in order to construct plans and estimate the properties of plans in such a modular way.

- In order to estimate the running time of a plan, it is necessary to predict the load of resources (machines and network interconnects). Like traditional optimizers, we use statistics to estimate such loads. For QoS management, however, these statistics need to be

evaluated in a different way because the probabilities for specific load situations need to be explicitly computed.

- The query optimizer enumerates alternative plans and prunes inferior plans based on their properties (e.g., estimated cost). A QoS-enhanced optimizer requires a special pruning metric in order to take all QoS parameters of a query into account.

In the following we will examine the problems which are listed above. Finally, some post-optimization actions are explained which are needed to adjust the optimizer estimations so that they are usable for admission control and monitoring.

7.1.1 Selecting Providers

Relevant data partitions from data providers and functions from function providers can be searched by a lookup service query. The corresponding parameters for the query like the requested type or freshness of the data and the class and signature of the function can be directly retrieved from the query itself or its quality constraints. As a result we get sets of matching functions together with the cost formulas registered by the respective developers and sets of matching partitions together with information about cardinality, relevance, attribute value distributions (histograms) and average tuple size. This level of detail in the meta-data for the relations can be reached even in a heterogeneous environment by histograms constructed by query feedback [AC99], frameworks which estimate data source relevance and overlap [LRO96, FKL97] and mediator costing frameworks [ROH99]. Furthermore, we also need statistics about the load characteristics of cycle providers and network links. Again, work on retrieving these statistics in a distributed environment has already been done [GRZZ00, WSP97].

As described in Section 4.1, we use a cluster tree for selecting cycle and data providers for a query. This cluster tree is based on the interconnection graph of data and cycle providers with the bandwidth of a network connection as the weight of the corresponding edge. We use this structure to search for the hot spot clusters of a query. In the context of QoS we also must consider the constraints on staleness and cardinality in addition to the completeness constraints when we search for these clusters. Figure 7.1 shows such a cluster tree.

The information about useful clusters is also utilized, when we group the sets of partitions, cycle providers and functions in classes which we call similarity classes. The similarity classes of resources will be annotated in the corresponding plan. These classes define a kind of environment for the plan which can be utilized during plan distribution and execution to break up the resource binding of the optimizer by replacing the initial resource with a better suited one out of its respective environment. An informal description of these similarity classes follows:

- The similarity class $Sim_{DP}^{R,dp}$ of a relation *R* and a data provider *dp* is the set of replicas of *p* or other partitions, currently not used in the query evaluation plan whose data providers appear in the same cluster as *dp*.

- The similarity class Sim_{CP}^{cp} of a cycle provider *cp*, is the set of cycle providers which, analogously, appear in the same cluster as *cp*.

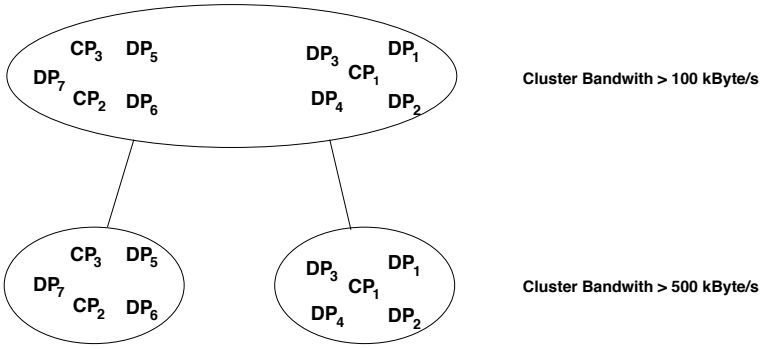


Figure 7.1: A simple Cluster Tree with Levels defined by Minimum Bandwidth Interconnections.

- The similarity class Sim_{FP}^{f} of a function f, is the set of all functions in the same class as found in the lookup service.

7.1.2 Estimating QoS Parameters

The estimations of cost and time consumption at a cycle provider have in common that they are based on the amount of work W which is induced by a specific plan. This dependency is obvious for the time consumption. However, it is artificially introduced for the cost consumption. Hereby we assume that the cost which is charged by cycle providers for the execution of plans depends on the work these plans induce.

The foundation for estimating the response time of query evaluation plans was made in the work done in [GHK92] and [GI97]. In the following, we refine this work by also considering parts of query evaluation plans which are executed in parallel with the rest of the plan but within these parts operators are executed in an interleaved way. Furthermore, we concentrate on the definition of a clearly structured quality model which should be obvious to implement.

Estimating the performed Work of an Operator

The amount of work done by an operator depends on the respective processing information *PI* (e.g., definitions for selection or join predicates or other parameters which affect the processing by the respective operator), the available main memory M, and the properties *DProps* (e.g., cardinalities, attribute value distributions, mean attribute sizes) of the operator's input streams. The implementor of an operator has to provide a *work function WF* which can compute the work performed by the operator based on that information. This function must adhere to the following signature:

$$WF : PI \times M \times DProps \rightarrow W \tag{7.1}$$

Additionally, the implementor of an operator must also provide a function *DPropsF* which computes the properties of the operator's output stream in a specific context:

$$DPropsF : PI \times DProps \rightarrow DProps \tag{7.2}$$

For each operator the function *WF* is itself constructed out of two functions:

$$WF(pi, M, dprops) = WOF(pi, M, dprops) + WNF(pi, M, dprops) \quad (7.3)$$

The function *WOF* estimates the work performed in the open phase of the operator and *WNF* the work in the next phase, respectively. Furthermore, the return values for all three functions *WF*, *WOF*, and *WNF* are equally dimensioned vectors (we call them *work vectors*), which means that the $+$-operation in Equation 7.3 corresponds to the usual summation on vectors. Each dimension in such a vector is associated with a resource, like CPU or disk. The unit of work for each dimension is determined by the type of the corresponding resource. For a CPU, work is measured by the number of instructions executed on it. If a more detailed model of the CPU should be used, we could integrate several more dimensions in the work vector, one dimension for every relevant kind of CPU instruction. The work functions would need to measure the quantities for the different instructions separately for each operator. For disk drives, we introduce two dimensions in the work vector, one for the number of disk accesses and the other for the overall number of bytes transfered (read or written).

Network connections between plan fragments are modeled as virtual, unary operators which use another work vector model than other operators. This work vector does not need information about disk accesses but has an entry for the number of needed network round trips for messages and the number of bytes which are transferred. The CPU related entries in the work vector of a network operator are replicated in order to consider the work done at the sender and receiver site separately. The CPU work estimated for a network operator also reflects the encryption or compression techniques which can be applied to the transfered data. Just as for normal operators, a network operator has different work functions for the open and the next phase of query execution.

Estimating the Time and Cost Consumption of Operators

The time and cost consumption of an operator *op* can be computed easily on the basis of the corresponding work vector. For each entry of such a vector the lookup service of our system provides meta-data about the cost and time consumption per unit work for that resource (that is *CpU* and *TpU* respectively) and the load (L) on it for the relevant range of time. For example, for a specific cycle provider we can obtain the information that the execution of an instruction lasts $5\mu s$ and costs 10^{-8} \$ and the load on the CPU is 40%. Now, we can compute the time and cost consumption of *op* for a specific resource *res*. In the formulas, we use the resource name for indexing the vectors for work, cost, and time consumption:

Cost Consumption (CC): $CC_{res} = (WF(pi, mem, dprops))_{res} * CpU_{res}$

Time Consumption (TC): $TC_{res} = (WF(pi, mem, dprops))_{res} * TpU_{res} * 1/L$

The overall time and cost consumption for an operator can then be computed as the summation of the respective values over all the resources. Due to the structure of WF, we can also compute the time consumption in the open (*TOC*) and the next (*TNC*) phases separately. In Section 7.1.3 we will provide more details on the right choice of the load parameters L of resources, which is correlated with the probability for load fluctuations on those resources.

Estimating Quality Parameters of Plans

For each sub-plan considered during optimization a corresponding plan descriptor keeps the information about its estimated quality parameters, information about the properties *Dprops* of the intermediate result produced by the sub-plan, and the associated resource requirements. The plan descriptor contains not only entries for the time dimension of our quality model, but also for the cost- and result quality dimensions. An example plan descriptor is shown below. For simplicity, we omitted the resource requirements and the *Dprops* information and the result quality parameters are restricted to one involved relation R.

$$PD := [\underbrace{QT_{first}, QT_{last}}_{\text{time sub-dimensions}}, \underbrace{QR^{R}_{comp}, QR_{min\#}}_{\text{result quality sub-dimensions}}, \underbrace{QC_{cycle}, QC_{data}, QC_{function}}_{\text{cost sub-dimensions}}]$$

The computation of a plan descriptor for a sub-plan needs the information delivered by the *DPropsF* and *WF* function of the operator which appears at the top of the sub-plan. This operator is called *root* in the following. Additionally, we need the plan descriptors $\{ipd_1, \ldots, ipd_n\}$ for the n sub-plans which produce the input data streams of *root*. Together with this information the quality parameters, except for the time quality parameters and the resource requirements, are easy to compute:

- The properties of the intermediate result of the sub-plan are given by the *DPropsF* function of the *root* operator with parameters taken from $ipd_1, \ldots, ipd_n$. If *root* is a leaf node the information can be found in the meta-data.
- The computation of the QR_{comp} parameter for a specific relation depends on the kind of *root*:
  - *root* = union: Add the respective parameters from $ipd_1, \ldots, ipd_n$.
  - *root* $\neq$ union: Take the minimum over the respective parameters from $ipd_1, \ldots, ipd_n$.

  Again, if *root* is a leaf node, the corresponding information can be retrieved from the meta-data.
- The cardinality estimations for the $QR_{min\#}$ are done with histogram support in the usual way [PIHS96] and the information for leave nodes can be found in the meta-data, again.
- The cost quality parameters for data, cycle, and function providers can be computed simply by summing up the corresponding values from $ipd_1, \ldots, ipd_n$ and *root*.

The Scheduling Model of an Operator For the computation of the time quality parameters it is important to know how operators schedule the actions of their input sub-plans. For example, Figure 7.2 shows the chronology of two query evaluation plans with a join operator being fed with input data by two scan operators. In the upper plan a double-pipelined hash join (DPHJOIN)

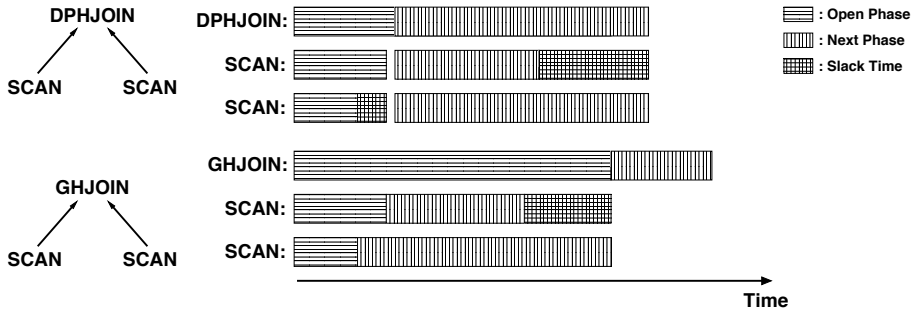


Figure 7.2: Chronology of Query Executions with different Scheduling Models

is used and in the lower plan a GRACE hash join (GHJOIN). As we can see in the picture, the GRACE hash join consumes its input streams in parallel during the open phase whereas the double-pipelined hash join does so in its next phase. Obviously, the QT_{first} parameter is mostly affected by the different scheduling behavior in this example.

We use a model, which we call *scheduling model* of an operator, to describe for each operator in a plan how it schedules its own actions together with the actions of its input plans. This model does not only depend on the implementation of an operator but also on the kind of communication between the operators. Communication between two operators can be realized by a synchronous interface which results in a pipelined-sequential execution of the corresponding operators, or an asynchronous interface which is used for communication across thread and machine boundaries and results in a pipelined-parallel execution of the corresponding operators. The information about the degree of parallelism in the execution can be expressed in the model by two operators:

- The binary sequence operator ';' indicates that the execution of the left and right operands is equivalent to a sequential execution. By this definition it is also allowed that the left and right operand intermingle their execution as this occurs in a pipelined-sequential execution of the next phases of two operators.

- The binary parallel operator '$\|$' indicates that the left and right operands are executed in a manner which is in its time consumption equivalent to a parallel execution.

The operands of these two operations can be

- *op.open* and *op.next* which stand for the actions which are performed by operator *op* in its open and next phase, respectively.

- ipd_k*.open* and ipd_k*.next* which stand for the actions performed by the sub-plan associated with the input plan descriptor ipd_k in its open and next phase respectively.

We define that the sequence operator has a higher precedence than the parallel operator and that precedence can be modified by parentheses in the usual way. Both operators are associative.

We provide two different scheduling terms for an operator in order to reflect the scheduling behavior in its open phase and in its next phase. For example, the corresponding terms for the double-pipelined hash join and the GRACE hash join are:

DPHJOIN: **open:** $op.open \| ipd_1.open \| ipd_2.open$

next: $op.next \| ipd_1.next \| ipd_2.next$

GHJOIN: **open:** $op.open \| (ipd_1.open; ipd_1.next) \| (ipd_2.open; ipd_2.next)$

next: $op.next$

In these cases all connections between operators are assumed to be pipelined-parallel[1]. A completely pipelined-sequential scheduling model of the GRACE hash join is:

open: $op.open; ipd_1.open; ipd_1.next; ipd_2.open; ipd_2.next$

next: $op.next$

Evaluating Scheduling Models of Plans For the computation of the time quality parameters of a plan we introduce a function *evalTime* which is applied recursively to the scheduling models of the open or the next phase of the operators which appear in the plan. Applied to the scheduling model of the open phase of the operator *op* it computes the QT_{first} parameter of the sub-plan rooted at *op*, applied on the scheduling model of the next phase it computes the value QT_{last}. This means, that for a given sub-plan which appears in the bottom-up optimization process *evalTime* can be used to compute the QT_{first} and QT_{last} parameters of the according plan descriptor. *evalTime* is recursively defined as follows (for a simpler presentation we omitted all the parameters of the function except for the scheduling model):

$$
\begin{aligned}
evalTime(x \| y) &= \max(evalTime(x), evalTime(y)) \\
evalTime(x; y) &= evalTime(x) + evalTime(y) \\
evalTime((x)) &= evalTime(x) \\
evalTime(ipd_k.open) &= ipd_k.QT_{first} \\
evalTime(ipd_k.next) &= ipd_k.QT_{last} \\
evalTime(op.open) &= op.TOC \\
evalTime(op.next) &= op.TNC
\end{aligned}
$$

Although the definition of *evalTime* is already quite complex, applied on a complete query evaluation plan it still cannot assess the effects of parallel execution exactly. For example, look at the query evaluation plan in Figure 7.3. The thick lines denote pipelined-parallel connections and the thin lines pipelined-sequential ones. The evaluation of *evalTime* will take into account that the actions of the *union* operator are executed in parallel with those of the operators *scan2*, *scan3*, and *scan4*. Obviously, the actions of the *scale* operator do also run in parallel with the three scan operators, but this will not be realized by *evalTime*.

[1]For the GRACE hash join, operator-internal parallelism is also needed to compute the join in the way the scheduling model suggests, but this can be captured in the scheduling model by encapsulating each input operator in a virtual operator which also performs some actions of the join operator.

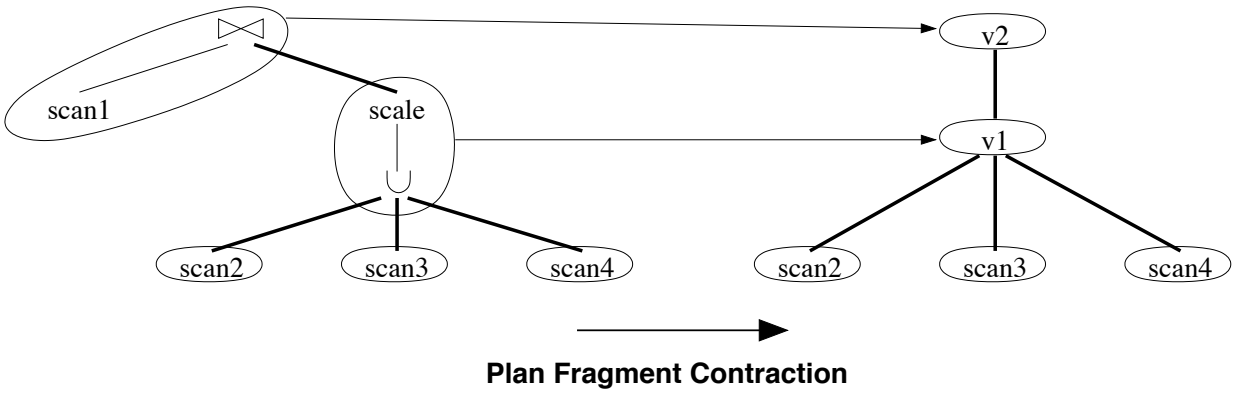


Figure 7.3: Fragmentation of QEPs in Parallel Working Plan Fragments.

Therefore, the time quality parameters are computed in two phases. First, we just consider plan fragments which represent the maximum components in the graph of the query evaluation plan, connected only by pipelined-sequential connections. These plan fragments are handled as virtual operators. The time consumption of such a virtual operator is computed by the application of *evalTime* to the plan fragments scheduling model. This scheduling model is constructed out of the fragment's operators. An example for the relationship between those plan fragments and the corresponding virtual operators is shown in Figure 7.3. In the second phase, we merge the scheduling models of those operators of the plan fragment which have a pipelined-parallel input link in order to obtain a scheduling model for the virtual operator. This merge operation is rather simple since the *op.open* and *op.next* actions of the operators' scheduling models are unified to represent the corresponding actions of the virtual operator and the actions of the input plans are scheduled in the same way (parallel/sequential) as in the operators' scheduling models. This means that in the merge process '||'-operators are merged before ';'-operators. In the virtual operator scheduling model the terms *op.open* and *op.next* refer to the time estimates of the plan fragment computed in the first phase. The time estimates of the resulting contracted query evaluation plan are then computed by the *evalTime* function.

7.1.3 Managing Uncertainty in Resource Availability

In Section 7.1.2 we stated, that the optimizer needs meta data about the properties of the data, the available main memory, and the load on resources used in a query evaluation plan. Obviously, some of these environmental parameters exhibit a somewhat random behavior in the changes of their values and can therefore be considered as random variables with an associated probability distribution. This observation has also been made for central query processing which resulted in a solution for finding a query evaluation plan with the least expected costs[CHS99]. In our scenario with globally distributed cycle and data providers, it is even more inevitable to take account of the uncertainty in the parameter estimates, especially for main memory availability and load estimates. In the following, we will for simplicity of presentation only discuss load

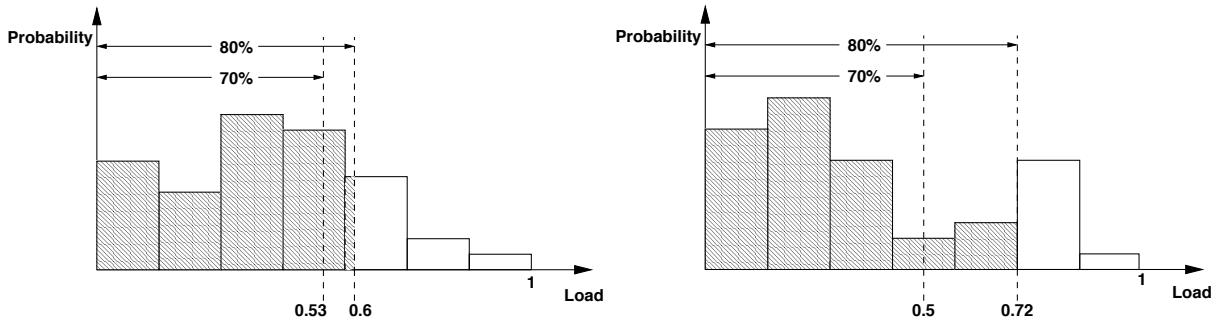


Figure 7.4: Load Distribution of different Cycle Providers.

parameters of time shared resources (disk, CPU, network) but the availability of the space shared resource main memory can be handled analogously.

For estimating the load on resources we use a statistical approach and roughly follow the work reported in [GRZZ00] and [WSP97]. Naturally, the activity of users and the resulting load on resources is based on time patterns which can be found within a day[2], a week, a month or a year. As described in [GRZZ00], a learning algorithm can be used to identify periods with a rather stable usage of a specific resource. In spite of this classification of usage periods, the load on resources will also vary in such periods. Therefore, for each such period a histogram as shown in both graphs of Figure 7.4 is constructed to approximate the probability density function for the random variable load for the corresponding resource. The load measurements for the learning algorithm and the histogram construction are retrieved from a distributed resource monitoring framework for computational and network resources. The architecture of such a framework in a globally distributed environment is described in [WSP97].

Uncertainty in the Availability of one Resource

When we have to generate an evaluation plan for a given query, we could now use the load histograms of potentially used resources for that period which belongs to the scheduled execution time of the query. But the question is, how to use these statistics. The formulas for computing the time consumption of a query use a fixed load parameter for each affected resource and cannot operate on a histogram. One solution could be to compute the expected time consumption of a query as proposed in [CHS99] for central query processing. But due to the properties of the expectation, the plan will achieve a higher time consumption than the estimated one with a probability of 50%. If the estimated time consumption exactly hits the corresponding quality constraint of the user, the plan will fail this quality constraint with a probability of 50%. This shows, that a dependability probability for the time consumption estimates of about 50% is not adequate for a system which tries to enforce user-defined QoS constraints on this parameter[3].

Therefore, we take the reverse way and allow the specification of a minimum dependability

[2]We assume that for the patterns within a day the load is logged with respect to a fixed time zone.

[3]Note, that the dependability probability itself depends on the accuracy of the resource statistics.

probability P_D for the time consumption estimates. This specification has to be translated into a concrete load estimate for every resource such that the given P_D holds. Now assume, that the whole plan just uses one resource. The optimizer then selects a maximum load value *mlv* so that the real load value *lv* falls below *mlv* according to the given histogram with probability P_D:

$$P(lv \leq mlv) = P_D$$

If we use this value for *mlv* in the computation of the time consumption, the query evaluation plan will find a resource availability during execution which is with a probability of P_D as good as assumed during optimization or even better. In the two histograms of Figure 7.4 load distributions of two different cycle providers are depicted. The height of a bar in such a histogram gives us the probability that a load value between the minimum and the maximum value of the bar's extension on the load axis will be discovered at run-time. The shaded areas in Figure 7.4 correspond to a value of 0.8 for P_D. The largest value on the load axis which belongs to a shaded area denotes the value for *mlv*. Therefore, the shaded area represents the summed up probability that the load value discovered at run-time will be less than *mlv*. We can see in the figure, that a lower given value for P_D (0.7 in this example) results in a lower value for *mlv* and consequently in a larger assumed resource availability for the estimation process. Furthermore, the histograms suggest that for the P_D value of 0.8 the resource corresponding to the left histogram is more suited than that for the right histogram, but it is vice versa for the P_D value 0.7.

Uncertainty in the Availability of n Resources

If a plan uses n resources and the optimizer proceeds for each resource as explained above, the overall dependability of the time consumption estimate will not be reached anymore if we assume that the load on the resources is independent from each other. Due to independence, the overall dependability can be computed by multiplying the individual probabilities (P_d for short) and the following holds:

$$P_D \leq (P_D)^n, \textit{ with } n > 0 \textit{ and } 0 \leq P_D \leq 1 \ ^{4}$$

To correct this divergence from P_D, we set the individual dependability probabilities for each resource to $\sqrt[n]{P_D}$. For growing n the value for P_d will approach the value 1 rather quickly, for example, $P_D = 0.9$ and $n = 10$ results in $P_d \approx 0.9895$. This means, that together with n the mlv value for a resource grows and consequently the assumed availability and the utility of the resource for enforcing the time quality constraints decreases. In this way, the number of eligible data and cycle providers for the optimization process could be reduced considerably. This effect can be mitigated and is not really a problem at all in many cases:

- Obviously, not all resources have independent load distributions, especially not those resources (CPU, disk) which belong to the same provider. Thus, the optimizer treats these resources and all the incoming network connections for a provider which are present in a specific evaluation plan, as one resource in the computation of P_d. Furthermore, providers with a small net distance to each other (this can be checked with the cluster tree introduced

[4] For example, if each of 5 resources have P_d values of 0.9, then $P_D = 0.9^5 \approx 0.6$.

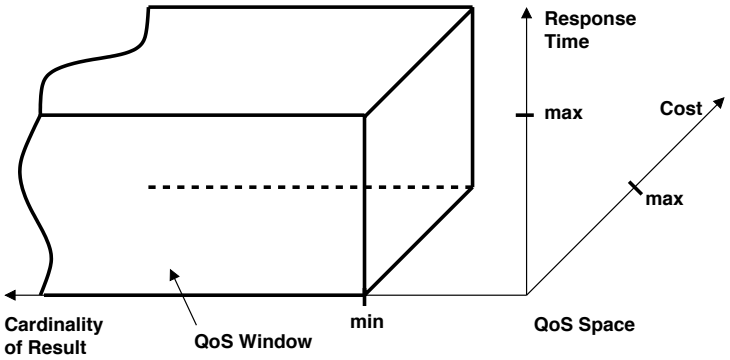


Figure 7.5: The QoS Space and the QoS Window.

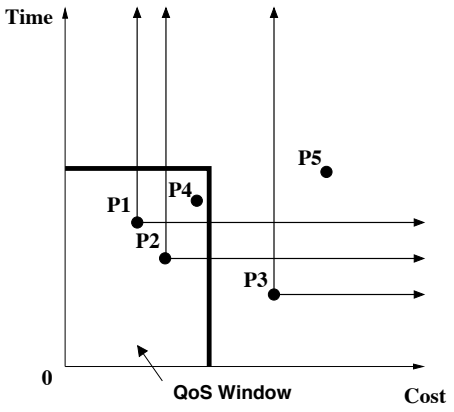


Figure 7.6: The Partial Order for Plans.

in section 7.1.1) will most likely show a dependent load. These providers will often be located in the same time zone and will therefore be exposed to the same activity patterns. Hence, providers which appear in the same cluster at a user-defined level of the cluster tree, should also be regarded as one resource in the computation of P_d.

- A rather large value for P_d does not inevitably mean that we have problems in finding suitable providers to produce a QoS compliant query evaluation plan. For example, think of a cycle provider whose load never exceeds 40%. Even if P_d would have a value of $1, 60$ % of the provider's resources would be available for the query. Depending on the resource requirements of the query and the nominal power of the provider this resource availability can be much more than enough. Of course, with increasing P_D and n the number of providers which will not pass this kind of filter in the optimization process will also increase. But only the fittest providers for a given query will pass this filter and this is how it has to work in this setting.

7.1.4 Pruning Query Evaluation Plans

The quality dimensions span a space which we call QoS space, and the user-defined constraints determine an area in that space which we call QoS window. This is shown in Figure 7.5 for the (simplified) three dimensional QoS space. During optimization every enumerated plan is mapped on a point in that QoS space by estimating the value for every quality parameter which appears in the quality model. Only plans which lie within the QoS window, fulfill the constraints of the user. For our realtor example the QoS window is given by four intervals:

- $[0\,,10\,]$ is the valid range for the total cost.
- $[0\,\mathrm{min}, 10\,\mathrm{min}]$ is the valid range for the total response time.
- $[70\%, 100\%]$ is the valid range for the QR_{comp}^{Estate} parameter.

- $[20, \infty]$ is the valid range for the $QR_{min\#}$ parameter.

In multi-objective optimization, pruning can only work with a partial order in such a setting. This is depicted in Figure 7.6, where we restricted the QoS space even further to only two dimensions in order to simplify the illustration. The figure shows, for example, that the plan *P1* is superior regarding time and cost consumption to the plans *P4* and *P5*. Although *P1* produces the query result faster, its execution is cheaper than the execution of *P4* and *P5*. *P1*, *P2* and *P3* are incomparable, but only *P1* and *P2* are candidate plans because *P3* lies outside the QoS window. The arrows emanating from these incomparable plans mark the area in the QoS space which is dominated by the respective plan. The plan *P4* lies inside the QoS window (i.e., the plan fulfills the user constraints), but it is no candidate plan, because it is dominated by the plans *P1* and *P2* both of which are superior to *P4* in *all* dimensions of the QoS space. Thus, *P1* and *P2* are the only plans "surviving" pruning.

The definition of the partial order $\leq_p$ which is used to compare alternative plans is shown below. Naturally, this order has to work on the plan descriptors which contain the quality estimates of the sub-plans generated during the optimization process.

$$PD_1 \leq_p PD_2 \quad <=> \quad \begin{cases} PD_1.QT & \leq_t & PD_2.QT \\ PD_1.QR & \leq_r & PD_2.QR \\ PD_1.QC & \leq_c & PD_2.QC \end{cases}$$

The comparison for each dimension in the definition above is itself defined with appropriately oriented comparisons for each of its sub-dimensions. For example, in the case of the result quality parameters, the following holds:

$$PD_1.QR \leq_r PD_2.QR <=> PD_1.QR_{comp} \geq PD_2.QR_{comp} \ \wedge \ PD_1.QR_{min\#} \geq PD_2.QR_{min\#}$$

When we use $\leq_p$ to prune plans from a set of semantically equivalent plans at some stage of the dynamic programming algorithm a set of plans—the Pareto curve—will remain. Then, we can further prune by just selecting these plans which lie within the intersection of the Pareto curve and the QoS window.

As mentioned in [GHK92], in the worst case, the complexity of the optimization process can increase by a factor which is exponential in the number of dimensions of the plan descriptor. This worst case occurs, if the values for the different dimensions are distributed independently. In our case, the plan descriptor has quite a lot dimensions, but it is apparently that several dimensions are highly correlated. The dimensions QT_{first}, QT_{last}, and QC_{cycle} are all based on the work performed by the corresponding plan and the dimensions QR_{comp}, $QR_{min\#}$, and QC_{data} depend on the selection of data providers for the plan. Furthermore, the possibilities for varying the value for the $QC_{function}$ dimension will be rather limited and nearly equal for every semantically equivalent plan. Therefore, the size of the Pareto curve should be much smaller than in the worst case and should not restrict the applicability of multi-objective optimization here in a significant way.

If there is more than one Pareto-optimal plan in the QoS window at the end of the optimization process as in the example above we also need heuristics to choose one of the remaining plans.

Here we must keep in mind, that possibly there will be some adaptation of the query execution necessary at instantiation or execution time of the plan. These adaptations normally result in a tradeoff between different quality parameters. Examples for such tradeoffs are

- cost versus result quality.
- cost versus response time.
- response time versus result quality.

For example, we could move a plan fragment from one cycle provider to another, in order to react on an imminent quality loss in response time; however, this could lead to a higher cost for the QC_{cycle} parameter.

This shows, that we should choose a plan which is robust against quality violations caused by resource fluctuations or side-effects of adaptations. Therefore, for every plan descriptor in the QoS window we determine the minimal distance to each of the user-defined borders of the window. Beforehand we normalize the values for the parameters with the maximal values occurring for the respective parameter in these plan descriptors. Otherwise the distances could not be compared in a fair way across dimensions. For our example we get:

$$\begin{aligned}\mathrm{Dist}(PD) = \min(&costConstraint - PD.QC,\\ &timeConstraint - PD.QT,\\ &resultConstraint - PD.QR)\end{aligned}$$

Again, the subtraction operation on each quality dimension is defined appropriately based on the corresponding sub-dimensions. We then choose that plan descriptor PD out of the set $PD_1, \ldots, PD_n$ of plan descriptors in the QoS window for which the following holds:

$$\mathrm{Dist}(PD) = \max(\mathrm{Dist}(PD_1), \ldots, \mathrm{Dist}(PD_n))$$

For the simplified example in Figure 7.7 this definition results in the selection of the plan P, because $\mathrm{Dist}(P) = ps$, $\mathrm{Dist}(Q) = qs$ and $ps > qs$.

[PY00] deals with a restricted form of our multi-objective optimization problem where a set of WWW data sources should be selected under a cost-time-quality tradeoff. The authors of that work present an optimization algorithm which generates an approximate solution for an instance of the problem and give some theoretical results on the complexity of finding such an approximate solution.

7.1.5 Relaxing some Constraints on Sub-Plans

As we have described in Section 7.1.4 the optimizer selects a plan with a maximum distance to the user-defined quality constraints. Therefore, the execution of the plan has some credits in the quality parameters which result from the differences between the user-defined quality constraints and the estimates of the optimizer. For example, in Figure 7.7 we see, that the plan P is allowed to cost pc cost units more and to use ps percent less data than estimated for the execution of P

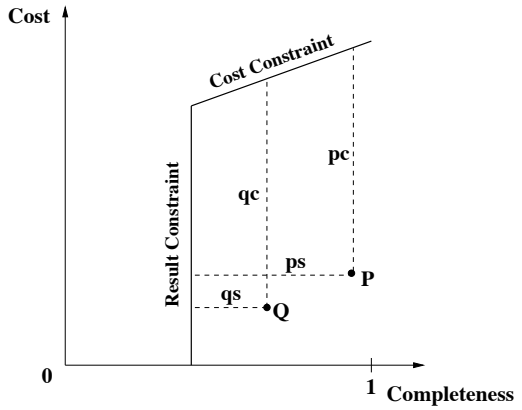


Figure 7.7: Plan Descriptors in the QoS Window

and nevertheless will still meet the user constraints. These credits are collected in a so called remainder account for the whole query execution plan. We do not only need these credits to "finance" dynamic adaptations of the query at execution time, but also for slightly missed quality constraints which are compensated with the credits of the remainder account instead of a more risky adaptation.

Again, the response time must be treated in a special way. During optimization, the response time part of the QoS model estimates the running time of a plan by detecting its critical path. Thus, a plan fragment which is not on the critical path, may consume more time than initially estimated by the optimizer because plan fragments executing in parallel consume more time than this one. For example, in Figure 7.2 two simple join plans are depicted, one with a double-pipelined hash join operator [WA91, IFF+99] and one with a grace hash join operator. Both join operators drive their input plans in parallel and due to the different time consumptions of the respective input plans, an extra time called *slack time*, can be added to the time constraints of the faster input plans.

Slack time cannot be distributed in a perfect manner during a processing step after optimization, when in the corresponding plan slack time would have to be shared among sequentially executed actions. For example, consider the Grace Hash Join plan in Figure 7.2. Available slack time for the `open` phase of this join operator can be distributed among the `open` and `next` phases of both input plans. But the `open`- and the `next` phase of an input plan are executed sequentially and we do not know in advance the needs of each of the phases for extra time. At runtime we could assign all the available slack time first to the `open` phase and after its execution, the remaining slack time can be assigned to the `next` phase.

However, estimating a plan fragment's resource requirements which are needed for admission control, demands a priori knowledge of its allowed running time. Furthermore, run time distribution of slack time in a distributed environment is a complex task. Thus, we use heuristics to distribute slack time statically after optimization. These heuristics work top-down on the scheduling model of the contracted plan (see Figure 7.3) and distribute slack time to sequentially executed actions proportionally to their estimated running time. We only have to define how the

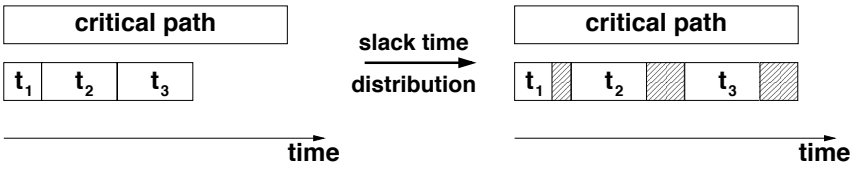


Figure 7.8: Slack Time Distribution

parallel and the sequence operator in the scheduling model of a plan have to be handled. Assume, that *t* is a term in the scheduling model. The available slack time for *t* which is passed from the operator above in the model is denoted as *t.st* and the estimated time consumption as *t.et*. The slack time distribution is then defined by the following two formulas:

$$\begin{aligned} t = t_1; \ldots; t_n &> \forall i \in \{1, \ldots, n\} : t_i.st = t.st \cdot \frac{t_i.et}{\sum_{j=1}^{n} t_j.et} \\ t = t_1 \| \ldots \| t_n &> \forall i \in \{1, \ldots, n\} : t_i.st = \max(t_1.et, \ldots, t_n.et) - t_i.et \end{aligned}$$

An example slack time distribution for the term *critical path*$\|(t_1\,;\,t_2\,;\,t_3)$ is shown in Figure 7.8. At the beginning, the extra time from the corresponding remainder account is assigned to the whole scheduling model of the plan as slack time.

7.2 QoS Enforcement during Plan Instantiation and Execution

The tasks of QoS management in these two phases are somewhat similar. The assumptions (resource or quality) determined by the optimizer are checked and in the case of an invalidated assumption the plan is adapted or, if this is not possible, rejected or aborted, respectively.

7.2.1 Plan Instantiation and Admission Control

The assumptions of the optimizer are stated in the form of the resource requirements annotated in the plan description. They will be checked by the admission control towards their validity directly before the respective service will be activated. The following resource parameters are rather easy to check, since a simple comparison of values is sufficient:

- The freshness and the size of requested data.
- The cost factors for CPU cycles, usage of functions and data consumption.
- The availability of main memory.

Admission control is more complicated for the resource requirements which more directly affect the time quality parameters of the query. The optimizer produces estimations for the work performed by plan fragments on resources like the CPU or the disk and with the help of operator

scheduling models the time consumption for the execution of plan fragments. As a result, the optimizer produces for every plan fragment and resource a vector (W, T) where W denotes the work performed by the fragment on a resource and T the maximum amount of time the plan fragment is allowed to consume. As in the work reported in [GI97] we assume that the work performed by an operator and consequently by a plan fragment is equally distributed along its execution. When a new plan fragment wants to be admitted, the admission control gathers for each of the running plan fragments the information about the remaining work W_r the fragment still has to perform and the time T_r the fragment is still allowed to run. This information is deduced by monitor operators from the original (W, T) vector of the plan fragment. The fraction W_r/T_r denotes the minimum average resource usage the corresponding fragment can tolerate. The vertical bars for *Query* 1 to 3 in Figure 7.9 represent these resource usages for all active plan fragments. Now, admission control can gather the information about available "resource packages", shown by the shaded boxes in the figure. The size of these resource packages will increase in the future development because already running plan fragments will finish their execution and consequently release the occupied resources. This gathering process is limited to the point in time when the maximum running time T of the new fragment is reached. If the size of the collected resource packages does not exceed the work information W for the new fragment, admission fails and the plan fragment is rejected. If this check succeeds for all types of resources affected by the fragment, the plan fragment can start execution on this cycle provider.

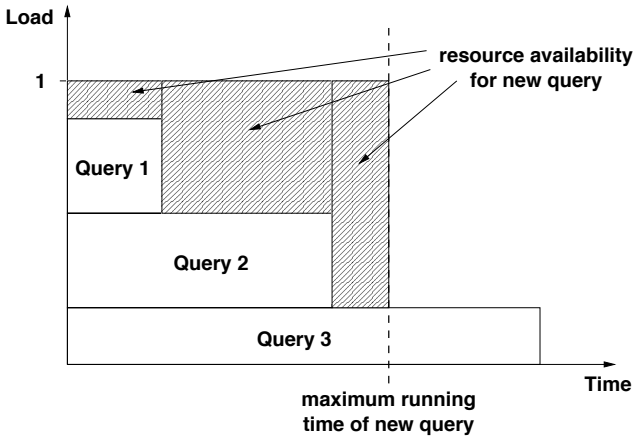


Figure 7.9: Gathering Available Shares on a Resource During Admission Control

If admission control fails, the estimates of the optimizer have been too optimistic. The decision about the application of an adaptation or an abort of this plan fragment at this stage can be made based upon information on the extent of the divergence and the kind of parameter which does not comply with the estimates. Since we use the same framework here as for the control of the adaptations during execution time which will be described later, we omit a further discussion here.

If there is a violation in the resource requirements and there are no useful alternative resources, then admission control aborts the plan fragment. As a result, the whole query needs to

be re-optimized with adjusted meta-data and re-started.

7.2.2 Plan Execution and Monitoring

Query evaluation plans in our system have a "natural" fragmentation which is given by the thread and machine boundaries which appear in the instantiated operator tree. For example, the plan fragmentation as shown in Figure 7.10 arises when the operators in the two fragments are executed in two different threads on the same machine or on two different machines. Monitoring and as we will see later also adaptation is done on the basis of plan fragments. Every plan fragment monitors its inputs—these are leaf operators of the operator tree or the inputs from other plan fragments—and its output by the use of monitor operators. For operators which represent pipeline breakers we also introduce monitor operators within a plan fragment in order to monitor the inputs of the pipeline breaker separately. A monitor operator traces the actual quality parameters of its input plan and forecasts these parameters for the end of its execution. The corresponding optimizer estimates for the respective sub-plan represent the target values for the forecasted values and a comparison of these values shows if the corresponding sub-plan is still within its quotas.

During query execution monitor operators keep track of the number of tuples produced, the time and cost consumption of the execution and some rates like cost or time consumption per produced tuple. These rates are used for projecting the past development of the quality parameters into the future. For example, the formula

$$T_N + R_{TN}(C_E - C_N)$$

uses the time consumption for the production of all the tuples so far (T_N), the time consumption per produced tuple (R_{TN}) and the estimated and current result cardinality, C_E and C_N, to compute an estimate for the overall time consumption of the sub-plan. For the calculation of the mentioned rates we use a moving average computation in order to detect changes in the execution behavior in a prompt way. The window size of the moving average computation determines how fast we can detect changes in the execution behavior. A larger window size results in a slow, but more reliable forecast; a smaller window size results in a fast, but less reliable forecast because the forecasting mechanism is more susceptible to bursty tuple production and skew in the data. Thus, we adapt the window size during the execution according to the riskiness of a sudden change to our quality parameters. This means that we start with a relatively large window size which is decreased constantly during the course of the execution.

Additionally, we gather runtime information for plan fragments which helps in the analysis of critical points in a query execution with jeopardized quality parameters. This information can also be used to reason about the effects of specific adaptations executed on the plan fragment. Some of the information we gather during runtime is listed below:

State Size: The size of the internal state of all operators in the plan fragment which would have to be transmitted, if the plan fragment was moved to another cycle provider.

Execution Progress: This parameter comes in two flavors: the remaining time until the execution has to be finished and the number of result tuples, the plan fragment still has to

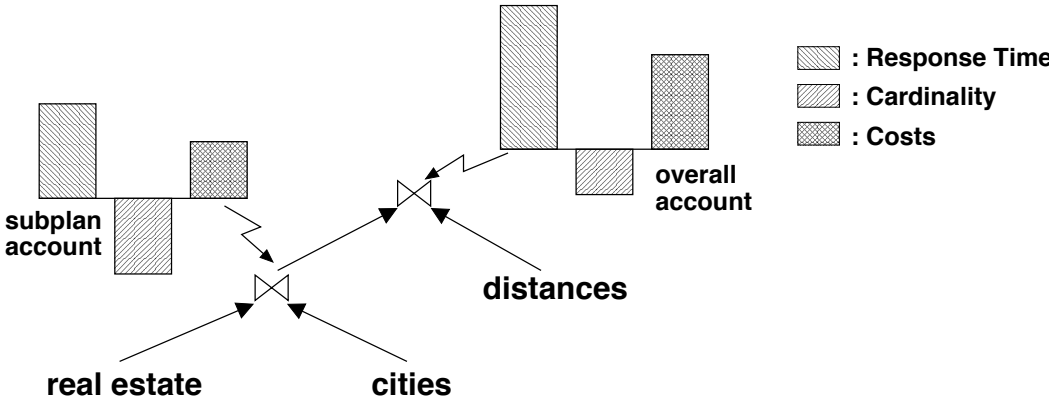


Figure 7.10: QoS Accounts of a Query Plan.

produce. This variable is important for our adaptation rule set, because expensive adaptations should not be executed, when a plan has nearly reached such a limit.

Buffer Pressure: In order to determine, if a plan fragment is itself a bottleneck or just suffers from slow plan fragments below and above it, we compare the fill rate of the respective input buffers and the output buffer. The difference between the fill rates of the input buffers and the fill rate of the output buffer determines the buffer pressure. Therefore, a high buffer pressure means, that the corresponding plan fragment is probably the bottleneck in the execution. Conversely, if the input buffers have a low fill rate and the output buffer has a high fill rate, we can deduce that the buffer pressure is low and the current plan fragment is certainly not the bottleneck and therefore adaptations to improve the response time are useless for this plan fragment.

Fragment Selectivity: The monitor measurements for the cardinality of the inputs and the output together with the respective cardinality estimates of the optimizer can be used to compute a current selectivity value for the whole plan fragment. This can be used to detect skew in the data.

In summary, the gathered statistics should help to determine if there is a problem with a quality constraint and if the current plan fragment is responsible for this problem. In the next section we show, how a plan fragment can react to such a problem.

7.3 The Adaptation of a Query Execution Plan

Adaptations are used during the query instantiation and the query execution phase. In the following we will concentrate on the query execution phase because the adaptations for the query instantiation phase are quite similar and the corresponding rule-based fuzzy controller uses only a different set of input variables and therefore, also a different set of rules. When a violation of the QoS constraints is expected during the runtime of a query, we can try to counteract this

violation by adapting the query execution plan. In this section we will first show some possible adaptations for the prevention of quality losses. After that we will describe the control system which decides when and how to adapt a plan.

7.3.1 Adaptations

The adaptations which we employ in our system to react to predicted QoS violations, operate on the resource allocation of the query evaluation plan. The corresponding resources are, for example, cycle providers, partitions from data providers and functions from function providers.

Prevention of Response Time Violation If a plan fragment seems to miss its constraints on the response time, we can use adaptations on the machine resource- or the application resource level. On the machine resource level we can change the priority of the respective thread (e.g., by a so called *increasePriority* adaptation), the main memory allocation for the respective operators (e.g., by a so called *increaseMemory* adaptation) or we can renegotiate the network service quality (e.g., by a so called *alterNetServiceQuality* adaptation), if the underlying network itself supports QoS handling like an ATM network. The adaptations on the application resource level comprise the activation of compression at runtime for the data sent through a network link (the *useCompression* adaptation) or the movement of plan fragments together with their state from one cycle provider to another (the *movePlan* adaptation)—again, during the runtime of the query. For example, if a monitor detects that a plan fragment suffers from a lack of computing power at its current cycle provider, the QoS management component can decide to move this plan fragment with all its state information to another, better suited cycle provider. This adaptation is depicted in Figure 7.11, where the scale-union plan fragment is moved from the cycle provider $CP3$ to $CP2$. The remainder of the plan fragment's work is then performed on the new cycle provider. All the other plan fragments of the same query above and beneath that plan fragment are not affected by this move operation, because the relevant communication links between them are disconnected and reestablished automatically by the runtime system of our query processor.

Prevention of Cardinality or Completeness Violation If the cardinality constraints or the completeness constraints are in danger, a possible adaptation is *addSubPlan* which integrates additional data sources in the query execution which were not involved in the original query execution plan. Of course the information about these additional data sources has to be annotated in the plan during the optimization phase. To accommodate this adaptation we have a union operator that can dynamically establish an additional sub-plan at runtime.

Prevention of Cost Violation If the costs of a plan fragment seem to exceed the corresponding limit, we can try to reduce the amount of processed data, for example, by a so called *reduceCompleteness* adaptation which stops input plans before they are finished. Other ways for reducing cost consumption are the movement of a plan fragment to a cycle provider which charges less for the execution (e.g., to the client's site), or the exchange of externally loaded functions , like

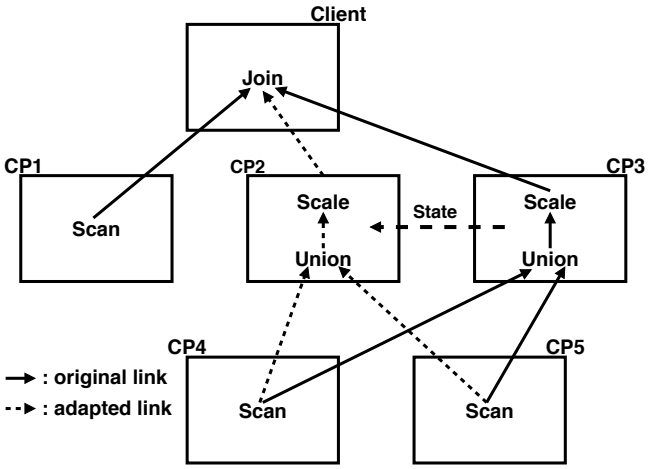


Figure 7.11: Moving a Plan Fragment from one Cycle Provider to another.

thumbnail encoders, with versions, that use a more lossy compression technique but which consume less CPU time. The type of the latter adaptation is called *reduceStrength*.

Naturally, there is also an *abort* adaptation which is initiated, when a quality violation in one of the quality dimensions seems inevitable. The decision about the execution of an adaptation depends on specific conditions like the forecasts of the quality parameters or information about the state of the query plan. For example, if for a sub-plan the time quality parameter is jeopardized and there is also little scope for the cost quality parameter, it does not make sense to move this sub-plan to a faster, but more expensive cycle provider.

7.3.2 Fuzzy Control

Plan adaptation during runtime is controlled by the monitor operators themselves because they are placed at the most interesting positions for adaptations in the query evaluation plan and also gather most of the information which is needed for this task. As shown in Figure 7.12, a monitor operator uses a rule-based fuzzy controller which gets the forecasts of the quality parameters and other state descriptions of the respective plan fragment as input. After the application of a rule-based fuzzy control technique which we describe later, the controller determines, if an adaptation should be applied and what adaptation this should be. There are proposals in the literature [IFF+99] to control query plan adaptations by event-condition-action (ECA) rules which also allow to construct a flexible controller. But such a controller is not suited for making decisions on the basis of uncertain information which is common in a distributed and heterogeneous environment.

The application of fuzzy controllers has been studied in different application areas, for example, in [LN99] for a visual tracking system. One reason for the use of a fuzzy controller in our system is that our runtime adaptations are discrete and this is quite easy to support in the inference rules which form the basis of the fuzzy controller's decisions. Furthermore, estimation errors and resource fluctuations introduce some uncertainty factors in the control process which

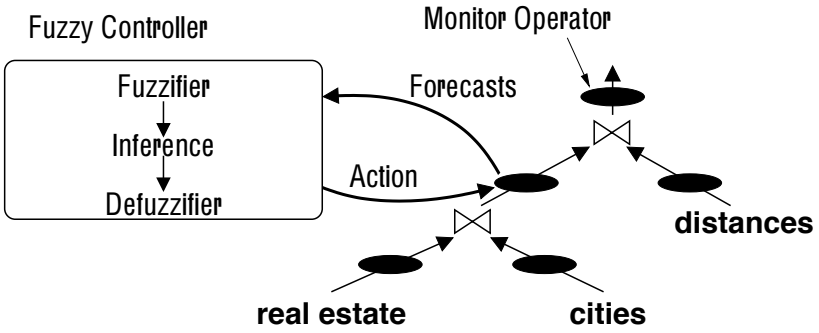


Figure 7.12: Feedback Loop for QoS Adaptations.

can be modeled by fuzzy logic [KY95] quite easily. The inference rules of a fuzzy controller also provide a highly configurable means to incorporate expert knowledge for the adaptation process in a very intuitive way. Therefore, it becomes possible to experiment with varying adaptation strategies by just changing the inference rules and perhaps the definition of the underlying fuzzy sets. In the following we present the three components of a fuzzy controller in our context: fuzzifier, fuzzy inference rules and defuzzifier.

Fuzzifier The task of the fuzzifier is to transform the input of the controller in a representation which can be used to trigger the inference rules. In our case the input consists of values for forecasts of quality parameters and values for state information of the plan fragment. This numeric values must be transformed to *linguistic values* of *linguistic variables*. For each input variable of the controller there exists one such *linguistic variable*. This transformation is depicted for the forecasted cost parameter in Figure 7.13—we call the corresponding linguistic variable *lfc*. We see, that the domain of the input is partitioned into different fuzzy sets, each fuzzy set represents a linguistic value of *lfc*, and the *weights* of the forecasted value which specify the level of containment to the fuzzy sets, can be determined. The linguistic values which correspond to these sets are normally adjectives which qualify the corresponding input variable. In our case the linguistic values *hopeless*, *endangered* and *compliant* qualify the chance of meeting the QoS constraints for this quality parameter. Figure 7.14 shows in more detail the partitioning of an input variable in fuzzy sets. A fuzzy set is described by its member function and an input value need not belong to exactly one set, but can belong to different sets with different weights—the input value in the figure belongs to the *endangered* fuzzy set with weight μ_e and to the *compliant* fuzzy set with weight μ_c.

Fuzzy Inference Rules The inference rules of a fuzzy controller are of the form

$$\textbf{if } X_1 \text{ is } A_1 \text{ and } \ldots \text{ and } X_n \text{ is } A_n \textbf{ then } Y \text{ is } B$$

where $X_1, \ldots, X_n$ and Y are linguistic variables and $A_1, \ldots, A_n$ and B are linguistic values with accompanying fuzzy sets. A small portion of an example rule set is shown below. Here *lfc*, *lfr*

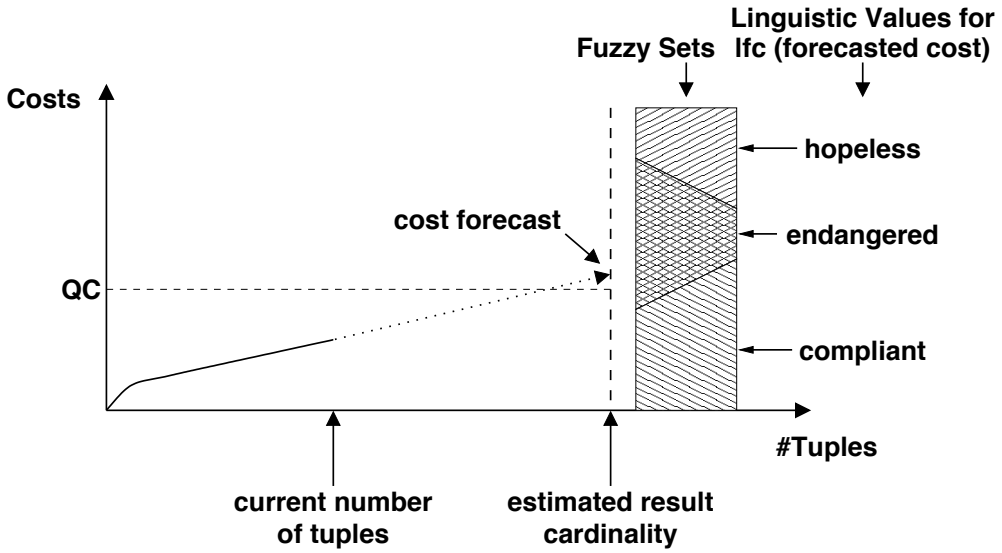


Figure 7.13: Mapping Quality Parameter Forecasts on Fuzzy Sets.

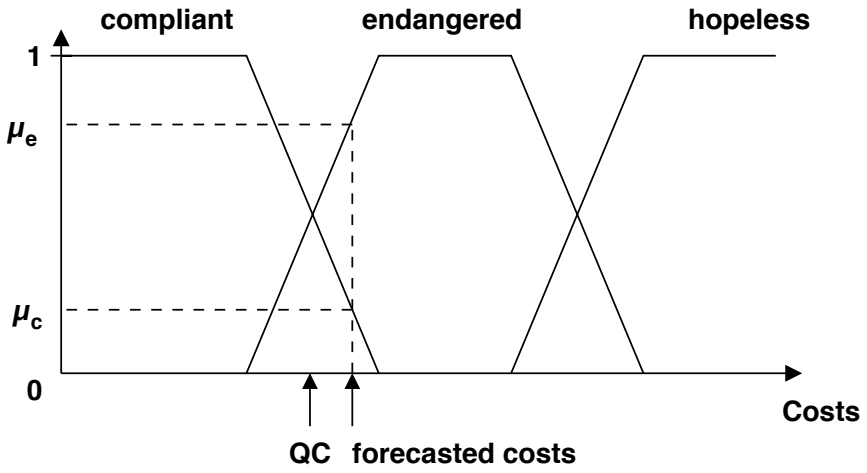


Figure 7.14: The Fuzzy Sets for the Cost Forecast Value.

and *lft* are the linguistic variables for cost-, result- and time quality parameters; *lep*, *lbp* and *lss* correspond to the execution progress, the buffer pressure and the state size.

if *lfc* is *endangered* and *lep* is *late* **then** *abort* is *preferred*
if *lft* is *endangered* and *lbp* is *high* and *lss* is *small* **then** *movePlan* is *preferred*
if *lft* is *endangered* and *lep* is *middle* and *lbp* is *high* **then** *increasePriority* is *possible*
if *lfr* is *endangered* and *lep* is *early* and *lfc* is *compliant* **then** *addSubPlan* is *preferred*

The variables in the **then** part of a rule represent specific adaptations. Every adaptation defines its own linguistic variable and the corresponding values, for example *not recommended*, *possible* and *preferred*, denote the applicability of the adaptation. Thus, our rule base consists of rules, whose heads describe the premise of specific adaptations. The combined weights of the linguistic variables of a rule's premise then determine the weight of its conclusion. In our case, the *and* operator in the premises of the rules is implemented by a *minimum* operation on the weights of the linguistic values. For example:

$$\mu_{endangered}(lfc) = 0.8 \ \wedge \ \mu_{late}(lep) = 0.9 \ => \mu_{preferred}(abort) = minimum(0.8, 0.9) = 0.8$$

Defuzzifier After all rules have been applied, all the resulting weights for an output linguistic variable are combined with an application-specific defuzzification method in order to get an overall value for the corresponding, virtual applicability scale of an adaptation. Here we assign every linguistic value a corresponding singleton out of the applicability scale; for our example, -1 is assigned to *not recommended*, 0.5 to *possible* and 1 to *preferred*. The overall applicability value x_a for an adaptation *a* is then determined by:

$$x_a = \mu_{not\ recommended}(a) * (-1.0) + \mu_{possible}(a) * 0.5 + \mu_{preferred}(a) * 1.0$$

We then choose the adaptation with the highest applicability value x_a, if this value is above 0.5. If there is no such adaptation, nothing will be done. For example, when for a certain adaptation, say *movePlan*, the combination of the inference rules yields $\mu_{not\ recommended}(movePlan) = 0.6$ and $\mu_{preferred}(movePlan) = 0.7$ then

$$x_{movePlan} = 0.6 * (-1.0) + 0.7 * 1.0 = 0.1$$

The adaptation will not be executed because $x_{movePlan}$ is smaller than the threshold of 0.5. This shows, that during defuzzification the output of different rules regarding the same adaptation are balanced. This is a special feature of a fuzzy controller in comparison to a pure ECA-rule based mechanism, where the rules are evaluated in isolation. The balancing of all rules for a certain adaptation helps in the design of the rule base. For example, for specific exceptional cases we can add *"veto rules"* which inhibit the activation of the corresponding (default) adaptation.

Chapter 8

QoS Experiments

Naturally, if the optimizer makes the right choices during plan generation, enforcing QoS constraints during run-time is not so complicated then. But, as we already mentioned, there are a lot of situations where these choices become disadvantageous at the end, for example, due to an unforeseen load increase at a cycle provider or a network link. In the following experiments we want to show that the local QoS management components at a cycle provider can also provide acceptable QoS support even when the initial assumptions and estimations of the optimizer do not hold.

8.1 The Effectiveness of Adaptations in a Distributed Environment

In these experiments we want to investigate to what extent monitor operators can forecast quality violations and if our adaptations can be used to react to such violations. For space limitations only two experiments are presented. We use three Internet hosts in our distributed benchmark environment which are located in *Passau*, *Mannheim*, and *Maryland*. In the following descriptions we use the locations to name the respective hosts. Each host is used as a cycle provider and additionally as a data provider for the *order* and *lineitem* relations of the TPC-D benchmark. In the first experiments, we also use wrappers for the data providers HotelBook (`www.hotelbook.com`) and HotelGuide (`www.hotelguide.com`) which deliver tuples with information about hotels.

8.1.1 Monitoring and Adapting Wrapper Plans

This experiment assesses the efficacy of the monitoring component in collaboration with the *movePlan* adaptation. The monitored sub-plan uses the two hotel data providers in order to look for hotels in a given city and combines the results with a union operator. We assume, that the sub-plan is executed under cost- and time constraints and that *Passau* is the client machine itself, therefore, inducing low costs (if at all), and *Maryland* is a commercial cycle provider. The execution time of the sub-plan is restricted to 200 seconds.

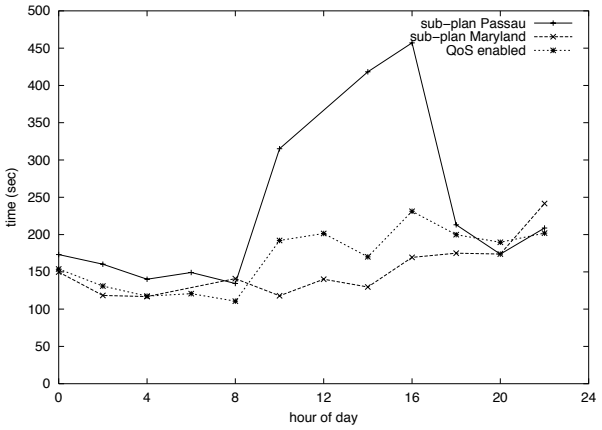


Figure 8.1: Retrieving Data through Wrappers.

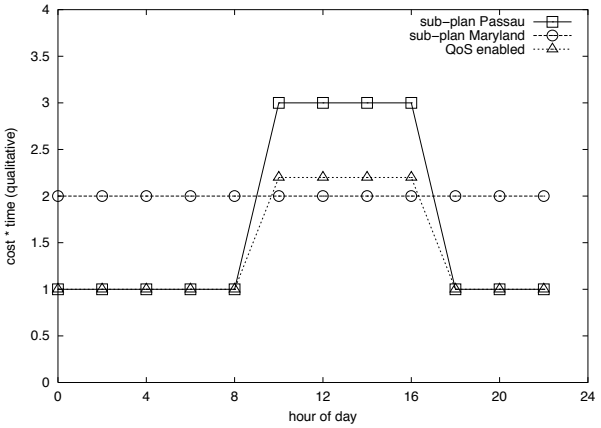


Figure 8.2: Qualitative View on the Wrapper Experiments.

We see in Figure 8.1, that the execution of the sub-plan at the client (in short: *Passau* plan) can compete most of the time with the execution at the apparently better placed cycle provider *Maryland* (in short: *Maryland* plan). Due to the low cost cycle provider *Passau*, the initial plan chosen by a QoS-aware optimizer would be the *Passau* plan and by a *movePlan* adaptation, this plan could be transformed into the *Maryland* plan. We used the daily rush hours on the Internet as a substitution for an unforeseen resource fluctuation on the network. Normally a QoS aware optimizer equipped with time dependent statistics would choose the *Maryland* plan instead of the *Passau* plan during these hours.

We executed the *Passau* sub-plan, the *Maryland* sub-plan and the QoS sub-plan every two hours in a 24 hour range. Between 8:00 and 20:00 the monitor operator can forecast the imminent quality miss and the *movePlan* adaptation is activated. This adaptation causes a decrease of running time and an increase of execution costs, since the wrapper execution gets faster, but also more expensive. A qualitative analysis of this experiment can be seen in Figure 8.2. This figure shows that the QoS management here tries to find the best balance between cost and time consumption. The cheaper plan is favoured as long as the time consumption does not exceed our time limit and if this happens the execution switches to a more expensive, but faster plan. Obviously, such an adaptation cannot always save a jeopardized quality constraint. For our experiment one can find executions of the QoS sub-plan which do not fulfill the 200 seconds limit. Normally, these failed executions would be aborted as soon as the quality miss appears inevitable, but in order to get reasonable graphs, we disabled the *abort* adaptation during these experiments.

8.1.2 Monitoring and Adapting Remote Sub-Plans

The query for this experiment uses a sub-plan with a scan operation at *Mannheim*. The client machine is *Passau* and the scan operation at *Mannheim* is performed on a *lineitem* partition with

a size of 10MB. Again, we assume that the query is executed under cost- and time constraints and that *Mannheim* is a cycle- and here also a data provider, whose services must be paid. The adaptation which we test here is called *useCompression* and turns on compression for a network connection at an arbitrary point during the query execution. Compression is performed with the help of the ZLIB library on blocks of the respective intermediate result. The time constraint for the sub-plan was set to 60 seconds.

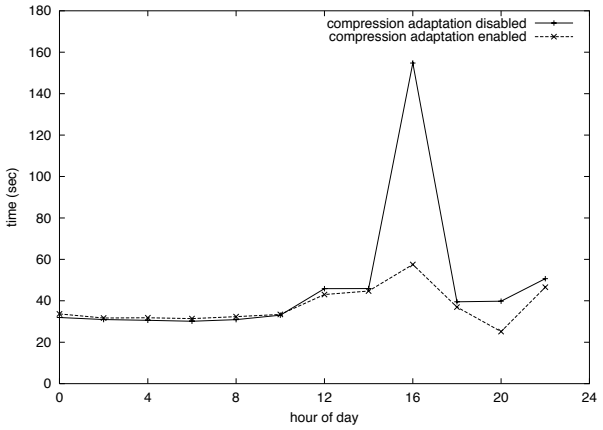


Figure 8.3: Executing a remote Scan Operation

The compression of data increases the CPU time consumption at the receiver's and even more at the sender's site. In our tests, the CPU time consumption at the sender increased by a factor of three, when compression is turned on. The sender site is *Mannheim* whose services have to be paid for, and as we see in Figure 8.3 the plan without compression is also able to fulfill the time constraint most of the time. Therefore, analogously to the experiment before the initial plan is that without compression turned on and the alternative for the resource *network* is to turn on compression.

As before, we executed the QoS plan and the one without QoS support every two hours in a 24 hour range and again used the rush hours on the Internet to simulate resource degradations on the network. In this experiment the sub-plan without compression missed the time constraint just once. The monitor operator in the QoS plan detected this situation and activated the compression on the network link which is clearly to see in the graph. The savings in time has to be paid for by an increased cost for the execution at *Mannheim*.

8.2 The Effectiveness of Run-Time QoS Management in Heavily Loaded Multi-User Environments

Due to the challenging environment in the previous experiments, monitoring the quality parameters was difficult but the control operations were rather simple, since the necessary decisions were of the form "adaptation on or off". Therefore, we set up a further test scenario with one

server and three clients, where the clients concurrently initiate queries on the server. Each experiment lasts a fixed amount of time *et* and within the resulting time frame each client initiates a fixed number of queries *nq*. All clients use the same query on the server but the quality parameter for the overall execution time of the query is varied among the clients. The query works on a data set of about 2000 tuples which occupy about 23 MByte of memory. Each tuple contains a picture in JPEG format which is scaled down in the query execution by a specialized operator. The execution times of the queries are CPU-bound since this scale operation is very computing intensive. The client *c1* issues queries with an overall execution time limit of 185 seconds and due to the 130 seconds CPU time our query needs, it induces an average CPU load of 70% if the query completely exhausts its time limit. The clients *c2* and *c3* use execution time limits of 370 (twice the value of *c1*) and 555 seconds (three times the value of *c1*) and the induced loads on the CPU are 35% and 23%, respectively. During our experiments we use an *nq* value of 20 which means that each client issues 20 queries and we used *et* values of 1, 2, 2.5 and 3 hours. At the beginning of an experiment, each client randomly chooses the *nq* starting times in the time frame given by *et*. In this way, the resulting arrival process for each client approximates a Poisson process with parameter $\lambda = nq/et$. A client starts its queries at the randomly chosen points in time and thus it is possible that two query executions of the same client overlap[1] and of course, queries from different clients overlap with a very high probability in this scenario. The imaginary average loads on the server which would be reached if all the queries could be executed in the corresponding time frames given by a specific *et* value, would be 217% ($et = 1$), 108% ($et = 2$), 87% ($et = 2.5$) and 72% ($et = 3$). This means, that for $et = 2.5$ and $et = 3$ all the queries could be executed within their time limits if their starting times are distributed in an appropriate manner. But this is most probably not the case in our experimantal set-up and the experiments without QoS management also show this.

8.2.1 Experimental Results without QoS Management

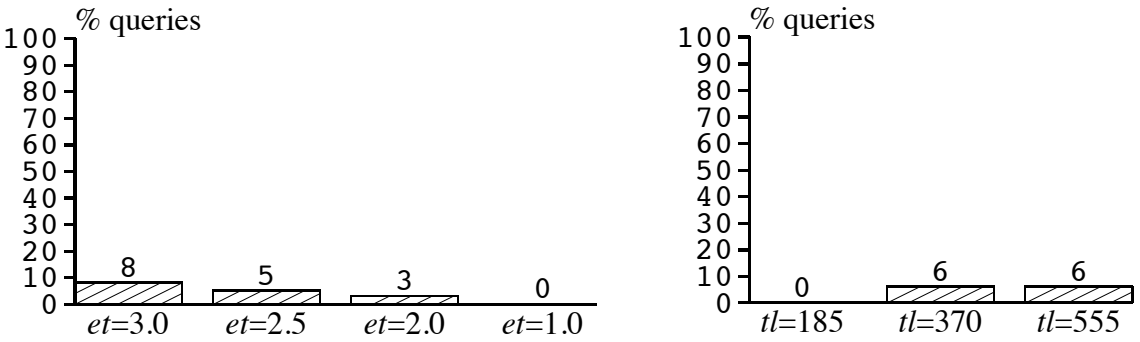
In Figure 8.4 the results are shown for an experiment as sketched above, where no QoS management was activated. This means, that

- no admission control is used.
- all queries have the same priority.
- plans are neither adapted nor stopped if they miss their quality constraints.

This is a typical best-effort configuration as found in many distributed information systems today, e.g., the WWW.

The left bar chart in Figure 8.4 shows the percentages of successful queries for different *et* values and in the right bar chart the percentage of successful queries is broken down for different *tl* values. As the charts show, a very small fraction of queries fulfill their QoS requirements. Naturally, queries with lower demands have a better chance to meet their constraints and in experiments with a lower average load also more queries can succeed in this task. One of the

[1] Each query is issued by its own client process, so this overlap is possible.



Successful Executions for different *tl* values. Successful Executions for different *et* values.

Figure 8.4: The Success of Queries with no QoS Techniques applied.

main problems of this configuration is the overload situation which arises due to the unrestricted access of queries to the server. More and more queries are admitted and accordingly, the share of the CPU usage of each query in the system decreases. This causes a longer individual service time for each running query and this again aggravates the congestion situation. For example, at the end of the experiment with an *et* value of 2 hours, about 40 queries were still active and the last of these queries finished its processing about 1 hour later. The overall running time of some queries rose up to several thousands of seconds.

This result shows, that without a local QoS management cycle providers can get overloaded very easily and the QoS properties become really poor in such a case. It should be noted that in these experiments we just test the ability of cycle providers to execute queries within their quality constraints. Of course, in a complete execution environment, the selection of appropriate cycle providers based on load statistics as done by the optimizer could result in less overloaded cycle providers. However, since we assume a global environment where each client uses an own optimizer instance, the selections of cycle providers are performed independently by each such instance and may nevertheless lead to highly loaded cycle providers. Therefore, it is inevitable that the local QoS management components can handle overload situations in a graceful way. Admission control is the corresponding means to limit the resource usage in a way that each admitted query has a chance to fulfill its constraints. The effects of an activated admission control are examined in the next experiments.

8.2.2 Experimental Results with Admission Control Activated

For the experiment with activated admission control we are interested in the percentages of queries which

- succeeded in executing within their time limits.
- were rejected by admission control.

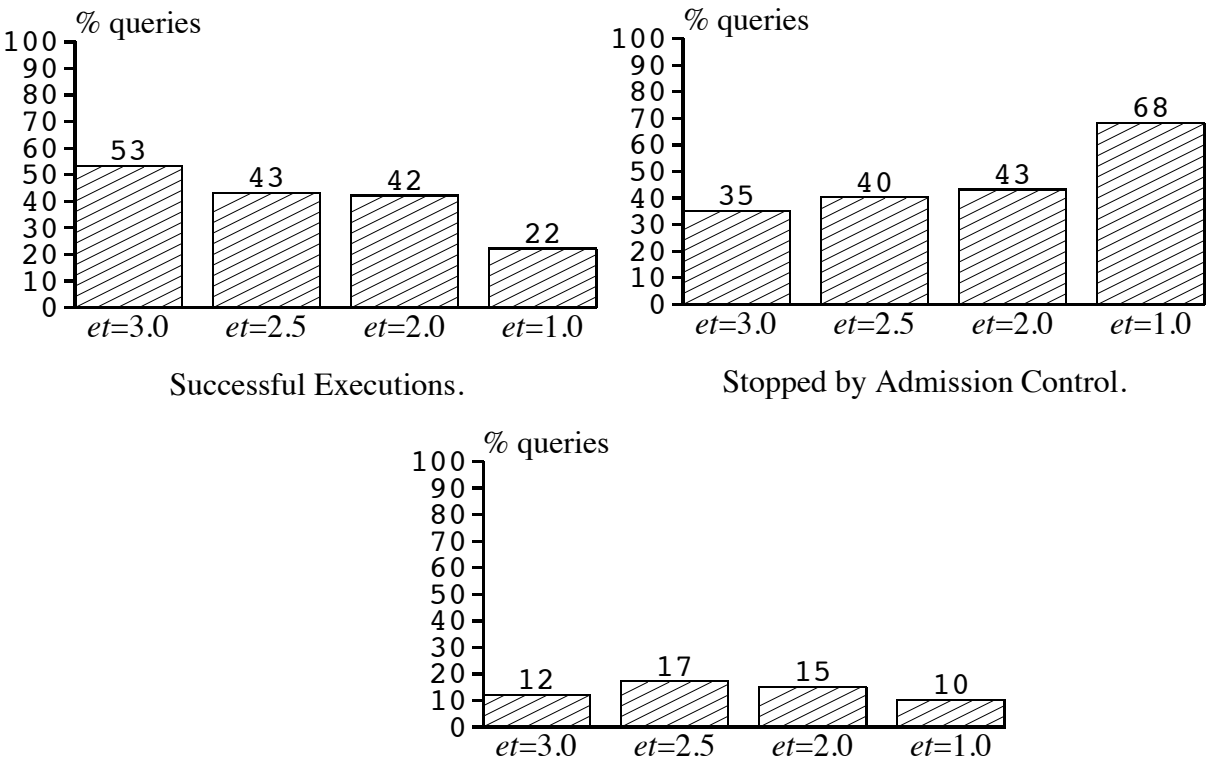


Figure 8.5: The Success of Queries for different *et* values with Admission Control activated.

- were admitted but at the end failed to meet their time limits.

In Figure 8.5 the results are broken down for different *et* values and in Figure 8.6 for different *tl* values. Obviously, the admission control has improved the results of the experiments by far. The rejection of queries results in a higher resource availability for the remaining queries. For each used *et* value an acceptable number of queries finishes execution within the time limits. Without admission control less than 10% of the queries could meet their time limits. Furthermore, none of the queries with the shortest time limit could execute successfully in the former experiments. But with activated admission control a considerable amount of these queries now do. Figure 8.6 shows that 24% of these queries were executed successfully.

The percentages of admitted but unsuccessful queries show that there is still some potential for improvement. These percentages are rather high and especially the queries with more demanding time limits suffer from a relatively high probability for an overshot time limit. The problem here is that the queries are not prioritized according to their demand. Analogously to earliest deadline scheduling in real-time systems, queries with a shorter execution period should get a higher execution priority than those with a more relaxed time limit. The results of the experiments which use run-time adaptations to adjust such priorities are discussed in the following.

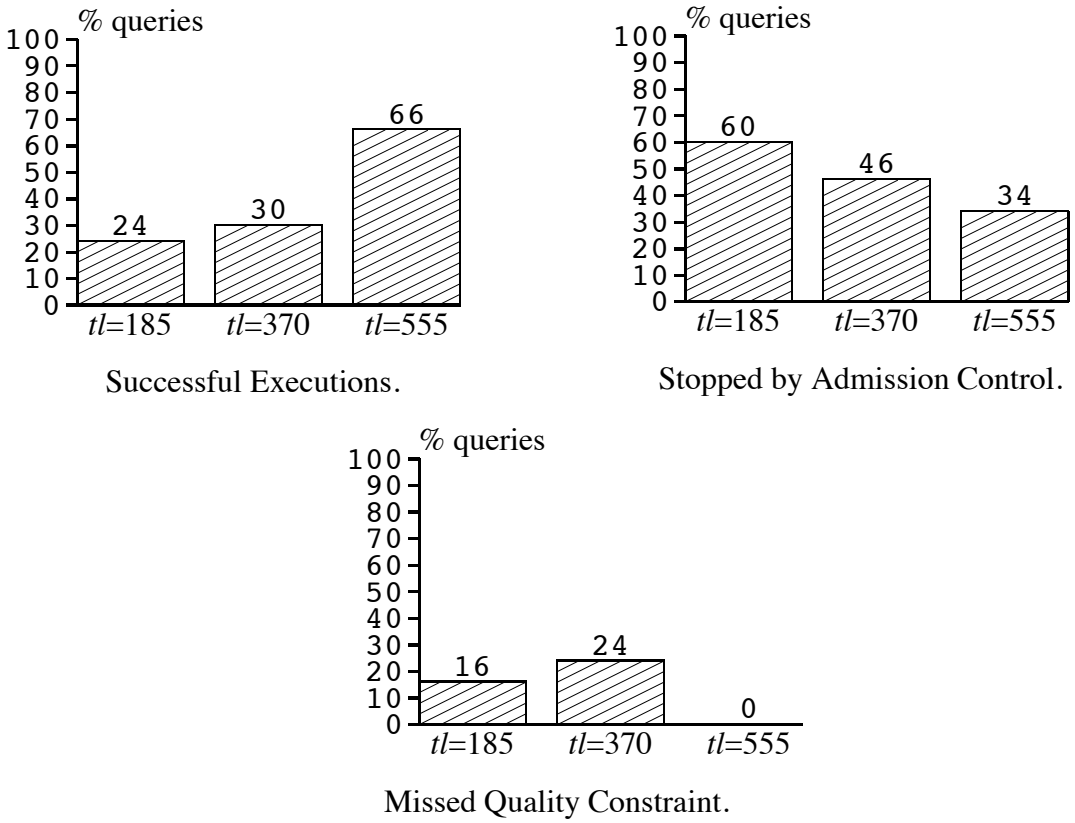


Figure 8.6: The Success of Queries for different *tl* values with Admission Control activated.

8.2.3 Experimental Results with Full QoS Management Support

The task of our QoS management in this experiment is to use the adaptations *increasePriority* and *decreasePriority* in a way that as many queries as possible are executed within their time limit. Furthermore, admission control and the usage of the *abort* adaptation must be used to stop queries which obviously will miss their time limit. This means, that the affected components of our QoS management are admission control, monitoring and adaptation control whereas the latter is performed by our fuzzy controller.

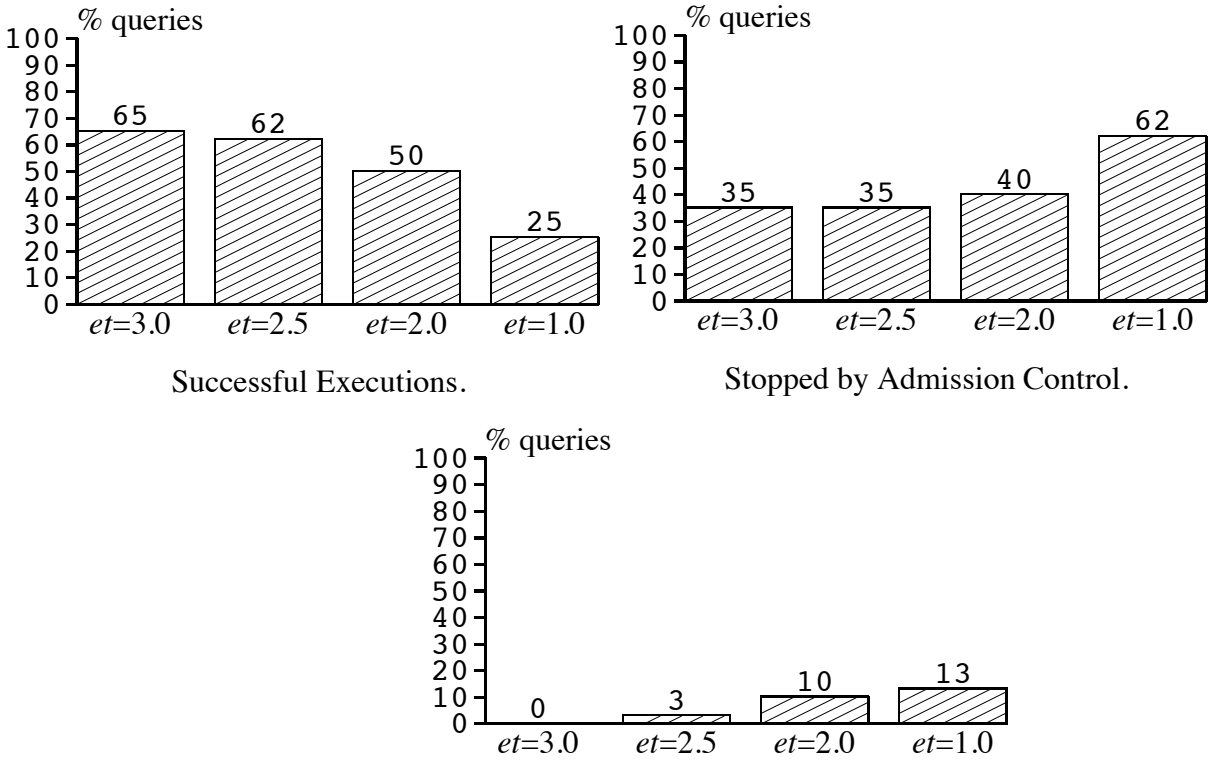


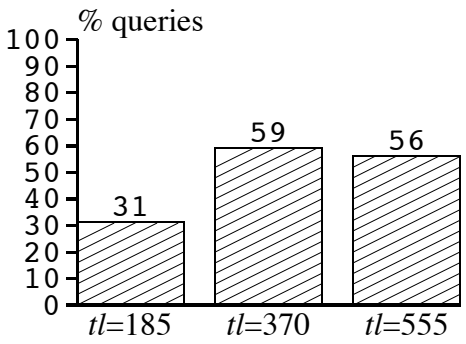
Figure 8.7: The Success of Queries with different *et* Values.

In Figure 8.7 the results for the experiment with fully activated QoS management are presented. The different bar charts show for different *et* values the percentage of queries, which were successfully executed, stopped by admission control or stopped by the *abort* adaptation. The bar charts for successful executions and for the aborts caused by admission control document the rather obvious trend in QoS management that a lower base load on critical resources improves the percentage of successful executions. This trend could also be seen in the previous experiments with no QoS management and admission control, respectively. In all of our experiments the number of initiated queries is the same but with different *et* values the execution times of the queries are distributed in differently sized time frames. Therefore, the number of collisions where several, more demanding queries are executed at the same time is lower when a larger time

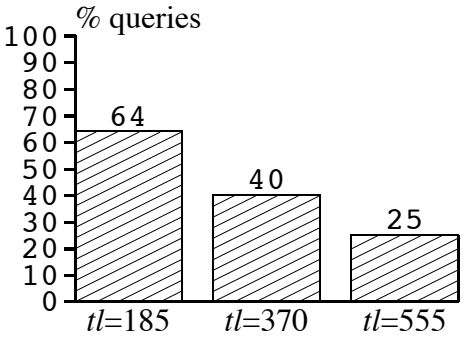
frame is used in the experiment.

The aborts initiated by the admission control are caused by the arrival process of queries and the induced load of these queries. This cannot be influenced by the QoS management of a cycle provider but depends on the decisions made by the corresponding optimizer instances. These decisions themselves depend on the quality of the load statistics and the overall risk factor P_D which is specified by the user. Therefore, in order to assess the QoS management of a cycle provider, we just have to consider the percentage of queries which were stopped by the *abort* adaptation. In our benchmarks, at most 13% of the queries have passed admission control but could not be executed within their time limits. First we must note, that the admission control as described in Section 7.2.1 is rather aggressive in accepting queries for execution. Admission control assesses the future development of already running queries in order to decide if a given query can accumulate enough resources to fulfill its quality constraints. Here, the admission control assumes that the fuzzy controller is able to stretch the execution of queries so that they are executed rather exactly within their respective time limits. Of course, this control process cannot be performed as precisely as assumed by admission control. Additionally, after a new query had been admitted or after a query had finished, the fuzzy controller needs some time to adapt the priorities of running queries to the new situation. Therefore, in a production system these effects need to be assessed and accordingly considered during admission control. But nevertheless, the number of aborted queries are even with such an aggressive admission control quite impressive, especially for the *et* values of 2.0 and 2.5.

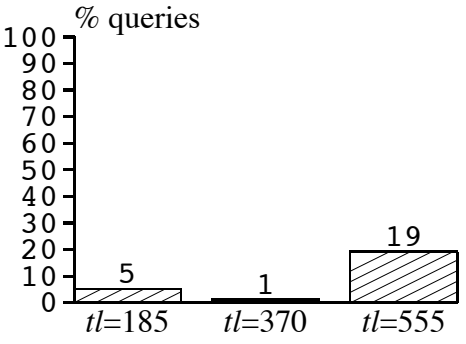
The above mentioned transition phases of the fuzzy controller especially affect long running queries. This can be seen in Figure 8.8. The percentage of queries which have to be stopped by the *abort* adaptation is much higher for the class of queries which have a time limit of 555 seconds than for the classes of queries with a time limit of 185 or 370 seconds. Queries with a longer running time are more exposed to transition phases of the fuzzy controller and so, when resources are scarce, these queries are more prone to quality misses regarding the overall execution time. A more positive conclusion which can be drawn from Figure 8.8 is that the QoS management at a cycle provider does not overly prefer a specific class of queries. Although queries with smaller demands will meet their quality constraints much easier than the others, in our experiments the queries with a time limit of 185 seconds also have a good chance to meet their quality constraint.



Successful Executions.



Stopped by Admission Control.



Stopped by Abort Adaptation.

Figure 8.8: The Success of Queries with different Time Limits.

Chapter 9

The Role of Functional Joins

The term functional join denotes specially structured join operations, where join partners for tuples of one input are retrieved by performing some kind of function call on the other input. In the next section, we will show some kinds of join operations which represent functional joins. Functional joins appear, for example, in data integration systems, where they are also named bind joins [HKWY97], and especially in object-oriented and object-relational database systems. In the context of the latter systems we introduce a new algorithm for implementing functional joins along nested sets.

9.1 Applications for Functional Joins

The term functional join does not denote a single join implementation but a special scheme for join processing. In this scheme, the tuples for one input stream (say R) of the join operation are piped into the corresponding join operator as usual. But the tuples of the other input (say S) are not available to the join operator in the same way. The tuples of S are accessed by some kind of function call implemented by a system component which we simply call *map* in the following. To find the join partners of a tuple from R, the join attributes of this tuple are used as arguments to the function call and the map delivers the join partners in S as the result of the call. This procedure is depicted in Figure 9.1. The *map* is responsible for performing the imaginary function call which delivers the join partners for the current tuple from R.

In data integration systems functional joins appear quite often and we already saw some of them in this work. Every data provider which only allows to query its data by means of point queries needs a wrapper which implicitly performs a functional join. Nearly all the data providers which are accessed through an `HTML` interface or by a `SOAP`-based protocol belong to this kind of data provider. For example, the providers for hotel data as mentioned in Chapter 5 need to be passed an identification of a city for which they should deliver the data for corresponding hotels. In these cases the wrapper works as a functional join operator and the data provider implements the *map* which is quite often a complete database system which manages the data of the provider.

As an example application for functional joins, assume that person *X* wants to tour the United States. *X* is interested in different routes, which are defined by the cities on a route, and hotels

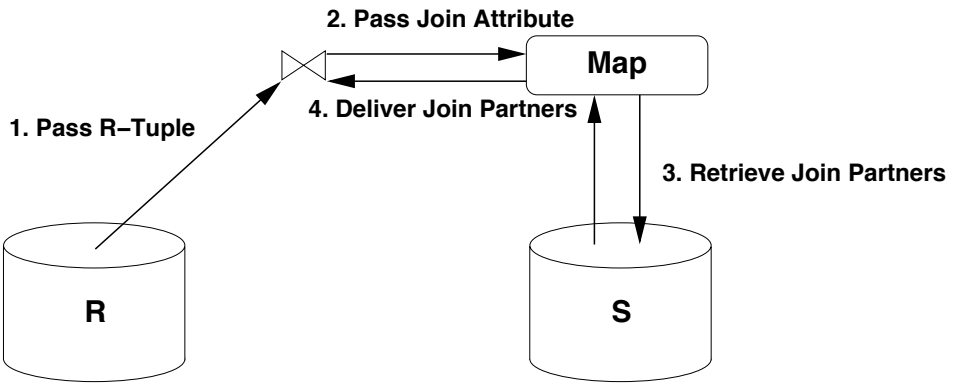


Figure 9.1: The Schema of a Functional Join.

in these cities. We assume that the ObjectGlobe system which is used in this application is able to perform a query which produces the routes with information about the means of transport, the corresponding time tables and the set of visited cities. The schema of this result could look like the following:

```
routes: {[..., visitedCities: {[cityName: String,
                                stateName: String]}]}
```

Then, the result of this query has to be joined with information from our hotel data provider. This means, we have to perform a functional join along the nested set for `visitedCities`. The execution of the functional join should fulfill the following requirements:

1. Since the access to data providers like the hotel data source is extremely costly (`http` network transfers, expensive *map* operations at the provider), multiple accesses for the same city should be avoided. This means that we have to break up the grouping of `visitedCities` in some way and reorder the cities in the intermediate result (partitioning, sorting).

2. Naturally, person *X* is interested in retaining the grouping which is given by the nested sets for `visitedCities`. Therefore, this grouping has to be reestablished after the join operation. Unfortunately, grouping is an expensive operation.

In order to fulfill these conflicting requirements in a satisfying manner we developed a new algorithm for implementing functional joins along nested sets. The problem, as sketched above, also appears in object-oriented and object-relational database systems. These systems use object identifiers (OIDs) to represent relationships between objects and these identifiers have to be mapped to the corresponding objects when a functional join on OIDs is performed. Due to the importance of OIDs in modeling databases in these systems, functional joins play an even greater role there than in data integration systems. Therefore, in the remainder of this work, we study functional joins in the context of object-oriented and object-relational database systems.

9.2 Functional Joins along Nested Reference Sets in Object-Relational and Object-Oriented Databases

Inter-object references are one of the key concepts of object-relational and object-oriented database systems. These references allow to directly traverse from one object to its associated (referenced) object(s). This is very efficient for navigating within a limited context—so-called "pointer chasing"-applications. However, in query processing a huge number of these inter-object references has to be traversed to evaluate functional joins. Therefore, naive pointer chasing techniques cannot yield competitive performance. Consequently, several researchers have investigated more advanced pointer join techniques to optimize the functional join. First of all, there are approaches to materialize (i.e., precompute) the functional joins in the form of generalized join indices [Val87]. [BK89] was the first proposal for indexing path expressions, and the access support relations [KM90] are another systematic approach for materializing the functional join along arbitrarily long path expressions; it was later augmented to join index hierarchies [XH94]. Many more proposals exist by now.

[SC90] were the first who systematically evaluated three pointer join techniques (naive, sorting, and hash partitioning) in comparison to a value-based join. Their work was augmented by [DLM93] to parallel database systems. [DLM93] also allowed nested sets of references, but they did not report on how to re-establish the grouping of the nested sets after performing the functional join. [CSL+90] incorporated some of the key ideas of pointer joins in the Starburst extensible database system. The emphasis in this work was on supporting hierarchical structures, i.e., one-to-many relationships, with hidden pointers. [GGT96] concentrate on finding the optimal evaluation order for chains of functional joins. Thus it complements our work: We devise an algorithm to efficiently evaluate a functional join within the query engine while they are concerned with finding the best evaluation order.

Unfortunately, the previous work on functional joins was constrained in two ways: (1) all approaches we know assume references being implemented as physical object identifiers (OIDs) and (2) most approaches are, in addition, limited to single-valued reference attributes. Both are severe limitations since most object-relational and all object-oriented database systems do support nested reference sets for modelling many-to-many and one-to-many object associations and many object systems do implement references as location-independent (logical) OIDs. In the following, we develop a new functional join algorithm that can be used for any realization form for OIDs (physical or logical) and is particularly geared towards supporting functional joins along reference sets. The algorithm can be applied to evaluate joins along arbitrarily long path expressions which may include one or more reference sets. The new algorithm generalizes previously proposed partition-based pointer joins by repeatedly applying partitioning with interleaved re-merging before evaluating the next functional join. Consequently, the algorithm is termed $P(PM)^*M$ where P stands for partitioning and M denotes a merging.

Before going into the technicalities let us motivate our work by a further example, this time in an object-oriented/relational context. We use a simplified order-inventory example application. In an object-oriented or object-relational schema the *LineItems* that model the many-to-many relationship between *Orders* and the ordered *Products* would most naturally be modelled as a

nested set within the *Order* objects:[1]

```
create type Order as (
  OrderNumber number,
  LineItems set(tuple( Quantity number,
                       ProductRef ref(Product)))
  ...);
```

```
create type Product as (
    ProductID number,
    Cost number,
    ...);
```

Here, *Order* refers to the ordered *Product*s via a nested set of references in attribute *LineItems*. Let *Orders* be a relation (or type extension) storing *Order* objects. Then, in an example query we could retrieve the *Orders*' total values:

```
select o.OrderNumber, (select sum(l.Quantity * l.ProductRef.Cost)
                       from o.LineItems l))
from Orders o;
```

Logically, the query starts at each *Order o* and traverses via the nested set of references to all the ordered *Products* to retrieve the *Cost* from which the total cost of the *Order* is computed. Note that, unlike in a pure (flat) relational schema, the nested set of *LineItems* constitutes an explicit grouping of the *LineItems* belonging to one *Order* that is, in most systems, also maintained at the physical level. Our new algorithm exploits this physically maintained grouping. However, we do, of course, avoid the danger of "thrashing" that is inherent in a naive nested loops pointer chasing approach. Our prototype implementation as well as a comprehensive analytical assessment based on a cost model prove that this new algorithm performs superior in almost all configurations. In particular, our $P(PM)^*M$-algorithm performs very well even for small memory sizes.

[1] Throughout the remainder of this work we will use some (pseudo) SQL syntax that is close to the commercial ORDBMS product that we used for comparison purposes. Unfortunately, the commercial ORDBMS products do not entirely obey the SQL3 standard.

Chapter 10

Implementing Functional Joins

Functional joins in object-oriented and object-relational database systems work upon object identifiers (OIDs) which are embedded in objects to represent references to other, related objects. Therefore, we first present the different implementation forms of object identifiers which appear in object-oriented and object-relational database systems. After that, we explain alternative implementations for performing a join operation on such object identifiers. Particularly, we give a detailed description of our new $P(PM)^*M$ algorithm and show some advanced applications of this algorithm.

10.1 Implementation of Object Identifiers

Object identity is a fundamental concept to enable object referencing in object-oriented and object-relational database systems. Each object has a unique object identifier (OID) that remains unchanged throughout the object's life time. There are two basic implementation concepts for OIDs: physical OIDs and logical OIDs [KC86].

10.1.1 Physical Object Identifiers

Physical OIDs contain parts of the initial permanent address of an object, e.g., the page identifier. Based on this information, an object can be directly accessed on a data page. This direct access facility is advantageous as long as the object is in fact stored at that address. Updates to the database may require, however, that objects are moved to other pages. In this case, a place holder (forward reference) is stored at the original location of the object that holds the new address of the object. When a moved object is referenced, two pages are accessed: the original page containing the forward and the page actually carrying the object. With increasing number of forwards, the performance of the DBMS gradually degrades, at some point making reorganization inevitable. O_2 [O2T94], ObjectStore [LLOW91], and (presumably) Illustra [Sto96, p. 57] are examples of commercial systems using physical OIDs.

10.1.2 Logical Object Identifiers

Logical OIDs do not contain the object address and are thus location independent. To find an object by OID, however, an additional mapping structure is required to map the logical OID to the physical address of the object. If an object is moved to a different address, only the entry in the mapping structure is updated. In the following, we describe three data structures for the mapping. [EGK95] give details and a performance comparison.

Mapping with a B^+-Tree

The logical OID serves as key to access the tree entry containing the actual object address (cf. Figure 10.1 (a)). In this graph, a letter represents a logical OID and a number denotes the physical address of the corresponding object (e.g., the object identified by a is stored at address 6). Here, we use simplified addresses; in a real system the address is composed of page identifier and location within that page. For each lookup, the tree is traversed from the root. Alternatively, if a large set of sorted logical OIDs needs to be mapped, a sequential scan of the leaves is possible. Shore [CDF+94] and (presumably) Oracle8 [LMB97] are systems employing B-trees for OID mapping.

Mapping with a Hash Table

The logical OID is used as key for a hash table lookup to find the map entry carrying the actual object address (cf. Figure 10.1 (b)). For example, Itasca [Ita93] and Versant [Ver97] implement OID mapping via hash tables.

Direct Mapping

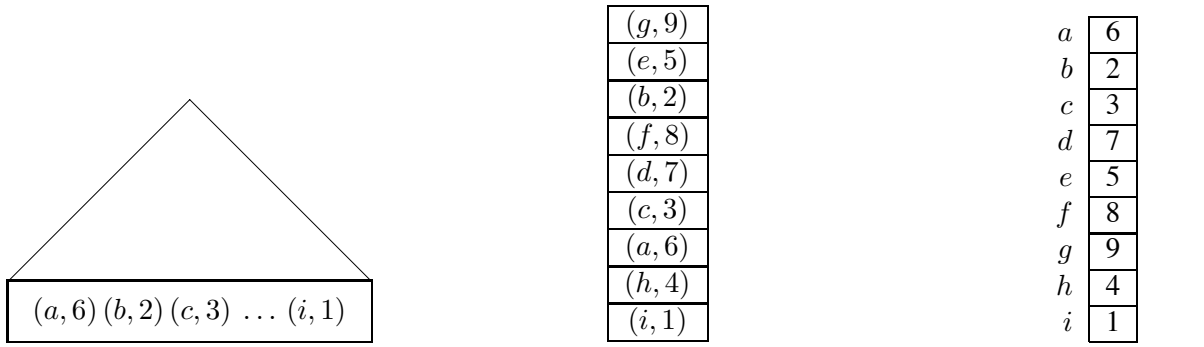
The logical OID constitutes the address of the map entry that in turn carries the object's address. In this respect, the logical OID can be seen as an index into a vector containing the mapping information. Direct mapping is immune to hash collisions and always requires only a single page access (cf. Figure 10.1 (c)). Furthermore, since the logical OIDs are not stored explicitly in the map, a higher storage density is achieved. Direct mapping was used in CODASYL database systems and is currently used in BeSS [BP95].

10.2 Functional Join Algorithms

The subsequent discussion of the algorithms is based on the following very simple abstract schema:

```
create type R_t as (
    R_Data char(200),
    SrefSet set(ref(S_t)),
    ...);
create table R of R_t;
```

```
create type S_t as (
    S_Attr number,
    S_Data char(200),
    ...);
create table S of S_t;
```



(a) B^+-tree (b) Hash-Table (c) Direct Mapping

Figure 10.1: Mapping Techniques

The example queries we wish to discuss are the following—one with an aggregation, the other without:[1]

select r.R_Data,
(**select sum**(s.S_Attr)
from r.SrefSet s)
from R r;

select r.R_Data,
(**select** s.S_Attr
from r.SrefSet s)
from R r;

10.2.1 Known Algorithms

The Naive Pointer-Chasing Algorithm

The naive, pointer chasing algorithm scans R and traverses every reference stored in the nested set *SrefSet* individually. For logical OIDs, first the *Map* is looked up to obtain the address of the referenced S object which is then accessed. If the combined size of the *Map* and S exceeds the memory capacity this algorithm performs very poorly.

In a system employing physical OIDs the naive algorithm does not need to perform the lookup in the *Map*. However, the access to the page the physical OID is referring to may reveal that the object has moved to a different page. In this case, the *forward* pointer has to be traversed in order to retrieve the object. Again, the algorithm performs very poorly if the size of S exceeds the memory capacity.

The Flatten-Algorithms

These algorithms flatten (unnest) the *SrefSet* attribute and partition or sort the flat tuples to achieve locality. For logical OIDs the evaluation plan looks as follows:

$$\nu_{S\_Attr:SattrSet}((\mu_{SrefSet:Sref}(R) \bowtie_{\rightsquigarrow} \mathit{Map}) \bowtie_{\rightsquigarrow} S)$$

[1] Note that the query on the right-hand side is not standard SQL because the nested query returns a set of tuples. However, some ORDBMS products do already support this—and in OQL this query is also possible (in a slightly different syntax, though).

Here, $\mu_{SrefSet:Sref}$ denotes the unnest (flatten) operator which replicates the R objects and replaces the set-valued attribute *SrefSet* with the single-valued attribute *Sref*. The nest operator $\nu_{S\_Attr:SattrSet}$ forms a set-valued attribute *SattrSet* from *S_Attr* [SS86]. The functional join is denoted by $\bowtie_{\rightsquigarrow}$ to indicate that for every (left) argument the corresponding join partner of the right argument is "looked up." To perform the two functional joins with the *Map* and with S, respectively, two techniques can be applied to achieve locality: partitioning and sorting.

If partitioning is applied, the flattened R tuples are partitioned such that each partition refers to a memory-sized partition of the *Map*. Upon replacing the logical OID in *Sref* by the address obtained from the *Map* the tuples are once again partitioned for the next functional join with S. Instead of partitioning, one could also sort the flattened R tuples. For the *Map* lookup the tuples are sorted on the *Sref* attribute and for the second functional join they are sorted on the addresses of S objects.

The final ν (nesting) operation is evaluated by grouping the flattened R tuples based on the OIDs of the original R objects. Grouping can be done by a sort-based or by a hash-based algorithm.

For physical OIDs the evaluation plan omits the first functional join with the *Map*.

Value-Based Join

The value-based join plan is as follows:

$$\nu_{S\_Attr:SattrSet}(\mu_{SrefSet:Sref}(R) \bowtie_{R.Sref=S.OID} S)$$

We assume that every object "knows" its OID—here *S.OID*. Note that this plan is equally applicable for logical and physical OID realizations because the object references are not traversed but only compared.

10.2.2 The Partition/Merge-Algorithm *P(PM)*M*

The partition/merge-algorithm is an adaptation of the above flatten/partition-algorithm in the way that it retains the grouping of the flattened R tuples across an arbitrary number of functional joins. This is achieved by interleaving partitioning and merging in order to retain (very cheaply) the grouping after every intermediate partitioning step. This is captured in the notation *P(PM)*M*. We will first describe the basic *P(PM)*^{1}*M*-algorithm which is applied when evaluating a single functional join under logical OIDs. More intermediate *PM*-steps are needed when longer functional join chains are evaluated (cf. Section 10.2.4).

In the *P(PM)*^{1}*M*-algorithm two joins are performed: (1) R is joined with the *Map* to replace the logical OIDs by their physical counterparts and (2) the result is joined with S. For evaluating the joins we will adapt the hash join algorithm. The *probe input* is R for the first join and R with the logical OIDs replaced by their physical counterparts—then called *RM*—in the second phase. Unlike the original hash join algorithm, only the probe input is explicitly partitioned.[2] The *build*

[2]For simplifying the presentation, we assume that the partitioning can be done in one recursion level—however, this is not required for the algorithm to work.

input, i.e., the *Map* and S, are either faulted into the buffer or—if range partitioning is applied—loaded explicitly (i.e., prefetched) into the buffer. In both cases, however, a partitioning step for the *Map* and S involving additional disk I/O is not required.

The successive steps of the partition/merge-algorithm can be visualized as follows:

$$\text{flatten and partition } R \text{ } N\text{-way} \rightarrow \text{join with } \textit{Map} \text{ to obtain } \textit{RM} \text{ and partition } RM \text{ } (N * K)\text{-way} \rightarrow \cdots$$

$$\cdots \rightarrow \text{re-merge } \textit{RM} \text{ to } K \text{ partitions and join with } S \rightarrow \text{merge}$$

That is, the partition/merge-algorithm first flattens the R objects and partitions them, then applies the mapping from logical to physical OIDs, partitions the resulting *RM*, then re-merges the initial partitioning and performs the join with S, and finally merges the partitions to restore the over-all grouping of the flat R tuples belonging to the same R object.

We need two partitioning functions h_M and h_S:

- h_M partitions the *Map* into N memory-sized chunks by mapping logical OIDs of S to the partition numbers 1 to N and
- h_S partitions S into K memory-sized chunks by mapping addresses of S objects to the partition numbers 1 to K.

That is, *Map* is partitioned into partitions $M_1, \ldots, M_N$ and S into $S_1, \ldots, S_K$. Actually, these partitioning functions are not applied on *Map* and S but on the logical OIDs stored in the nested sets of R and on their physical counterparts—in *RM* after applying the mapping.

In more detail, the algorithm performs the following four steps:

1. Flatten the nested *SrefSet*s and partition the flat R objects/replicas into N partitions, denoted $R_1, \ldots, R_N$. That is, for every object [r,{*Sref*$_1$,...,*Sref*$_l$}] $\in R$ generate the l flat tuples [r,*Sref*$_1$], ..., [r,*Sref*$_l$] and insert these tuples into their corresponding partitions h_M(*Sref*$_1$), ..., h_M(*Sref*$_l$), respectively. Of course, the R attributes (*R_Data* in our example query) need not be replicated. It is sufficient to include them in one of the flat tuples or, often even better, to leave them out and re-merge them at the end (cf. Section 10.2.5). The partitions are written to disk.
2. For all $1 \leq i \leq N$ do:
   - For (every) partition R_i the K initially empty partitions denoted $RM_{i1}, \ldots, RM_{iK}$ are generated.
   - Scan R_i and for every element [r,*Sref*] $\in R_i$ do:
     - Replace the logical OID *Sref* by its physical counterpart *Saddr* obtained (probed) from the i-th partition M_i of the *Map*.
     - Insert the tuple [r,*Saddr*] into the partition RM_{ij} where $j = h_S$(*Saddr*).

     Note that all OID mapping performed in this step concerns only partition M_i of the *Map*, which is either prefetched or faulted into the buffer.

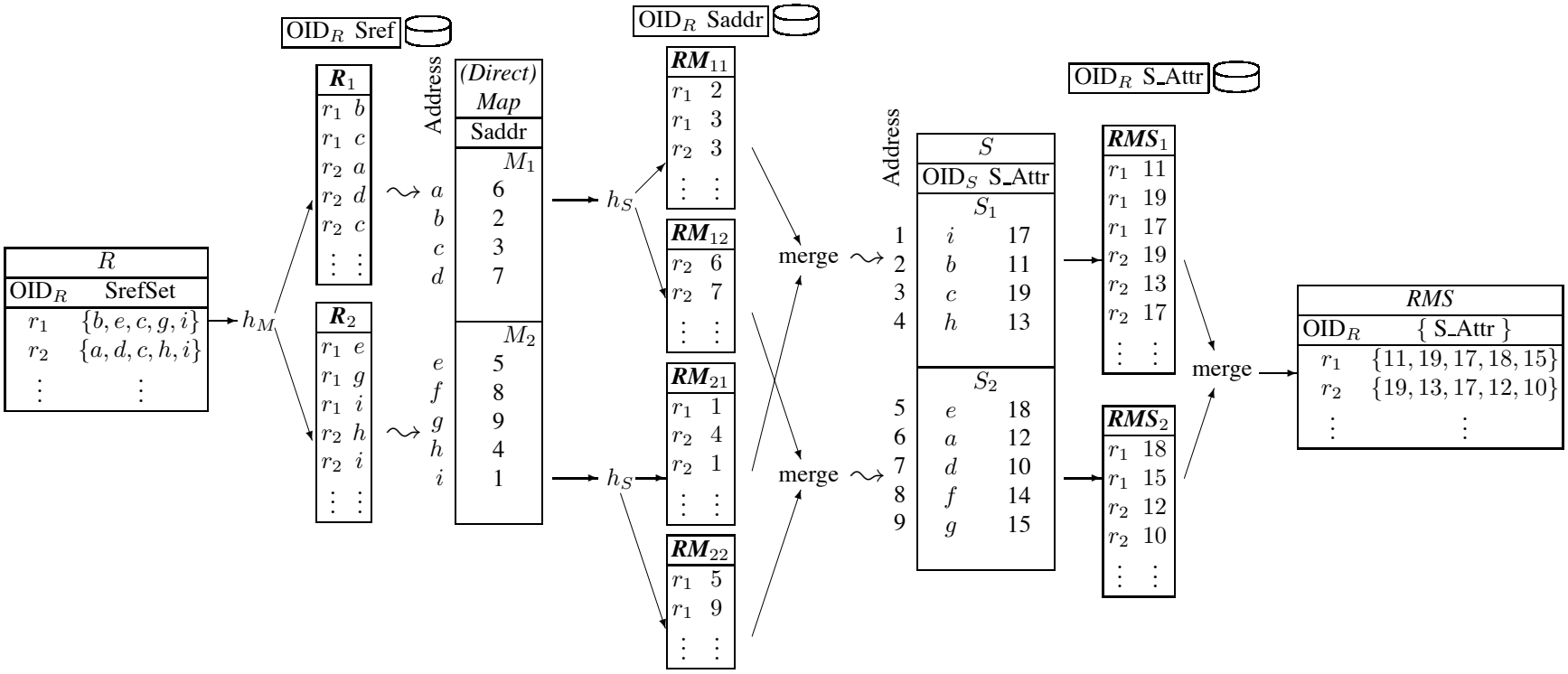


Figure 10.2: An Example Application of the Partition/Merge-Algorithm *P*(*PM*)**M* ($\rightsquigarrow$ denotes the lookup of the functional join; for simplicity, the handling of the additional R attribute *R_Data* is not shown)

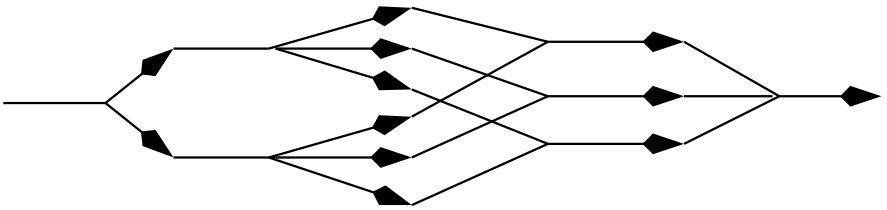


Figure 10.3: The Partition/Merge-Pattern of the *P(PM)*M* Algorithm

Having completed step 2., all the $N * K$ partitions $RM_{11}, \ldots, RM_{1K}$, $RM_{21}, \ldots, RM_{NK}$ are on disk.

3. For all $1 \leq j \leq K$ do:
   - Scan the N partitions $RM_{1j}, \ldots, RM_{Nj}$ simultaneously and merge them into a single tuple stream. The merging is done to restore the grouping of the flat R tuples according to R OIDs; that is, the merging generates the tuple stream $[r_1, \ldots]$, ..., $[r_1, \ldots]$, $[r_2, \ldots]$, ...
   - For every tuple [r,*Saddr*] the functional join with S is performed by looking up the S object at location *Saddr* and the relevant information, here *S_Attr*, is retrieved. Insert the tuple [r,*S_Attr*] into partition RMS_j.

     All S objects referenced in this step belong to the j-th partition S_j of S which is prefetched or faulted into the buffer—again, the partitioning ensures that the entire S_j fits into memory.

   After completion of step 3., the K partitions RMS_1,...,RMS_K are on disk.
4. Scan all partitions $RMS_1, \ldots, RMS_K$ simultaneously and re-assemble the flat tuples into the nested representation, i.e., group the tuples according to R-OIDs.

We note that the partition/merge-algorithm writes the (augmented) R to disk three times: (1) to generate the N partitions of the probe input for the application of the *Map*, (2) to generate the $N * K$ partitions after applying the *Map*, and (3) the K partitions obtained after joining with S. The intermediate $N*K$-way partitioning and subsequent N-fold merging of the $N*K$ partitions into K partitions is the key idea of this algorithm. This way the grouping of the flattened R tuples is preserved across the two partitioning steps with different partitioning functions h_M and h_S. Please observe that immediately distributing the objects into the K partitions after applying the *Map* would have destroyed the grouping on R that we want to retain in every partition. It is essential that the fine-grained partitions are generated first and that the re-merge is performed afterwards, as highlighted in Figure 10.3.

In comparison, the partition/merge-algorithm induces the same I/O-overhead as the basic flatten-algorithms of Section 10.2.1. However, the CPU cost of the partition/merge-algorithm is far lower than for the basic flatten-algorithms because there is no in-memory re-grouping involved. The flat tuples of the same R object are always in sequential order in all the partitions—it may only happen that some partitions do not contain any tuple. Furthermore, the *P(PM)*M*-

algorithm gives room for optimizations based on the retained grouping that are not applicable to other algorithms (cf. Section 10.2.5).

10.2.3 An Example of the *P(PM)***M*-Algorithm

Figure 10.2 shows a concrete example application of the *P(PM)***M*-algorithm with two partitioning steps. The tables R_i, RM_{ij} and RMS_j are labelled by a disk symbol to indicate that these temporary partitions are stored on disk.

We start with table R containing two objects with logical OIDs r_1 and r_2—for simplicity, any additional R attributes are omitted. The set-valued attribute *SrefSet* contains sets of references (logical OIDs) to S. The first processing step flattens these sets and partitions the stream of flat tuples. In our example, the partitioning function h_M maps $\{a, \ldots, d\}$ to partition R_1 and $\{e, \ldots, i\}$ to partition R_2. The next processing step starts with reading R_1 from disk, maps the logical OIDs in attribute *Sref* to object addresses using the portion M_1 of the *Map* (note that the *Map* is not explicitly partitioned) and in the same step partitions the tuple streams again with partitioning function h_S (h_S maps $\{1, \ldots, 4\}$ to partition 1 and $\{5, \ldots, 9\}$ to partition 2. The resulting partitions RM_{1j} (here $1 \leq j \leq 2$) are written to disk. Processing then continues with partition R_2 whose tuples are partitioned into RM_{2j} ($1 \leq j \leq 2$). Once again, let us emphasize that the fine-grained partitioning into the $N * K$ (here $2 * 2$) partitions is essential to preserve the order of the flat R tuples belonging to the same R object. The subsequent merge scans N (here 2) of these partitions in parallel in order to re-merge the fine-grained partitioning into the K partitions needed for the next functional join step. Skipping the fine-grained partitioning into $N * K$ partitions and, instead, partitioning *RM* into the K partitions right away would not preserve the ordering of the R tuples. In detail, the third phase starts with merging RM_{11} and RM_{21} and simultaneously dereferences the S objects referred to in the tuples. In the example, [r_1,2] is fetched from RM_{11} and the S object at address 2 is dereferenced. The requested attribute value (*S_Attr*) of the S object—here 11—is then written to partition RMS_1 as tuple [r_1,11]. After processing [r_1,3] from partition RM_{11}, [r_1,1] is retrieved from RM_{21} and the object address 1 is dereferenced, yielding a tuple [r_1, 17] in partition RMS_1. Now that all flattened tuples belonging to r_1 from RM_{11} and RM_{21} are processed, the merge continues with r_2. After the partitions RM_{11} and RM_{21} are processed, RM_{12} and RM_{22} are merged in the same way to yield a single partition RMS_2. As a final step, the partitions RMS_1 and RMS_2 are merged to form the result *RMS*. During this step, the flat tuples [r,*S_Attr*] are nested (grouped) to form set-valued attributes [r,{*S_Attr*}]. If aggregation of the nested *S_Attr* values had been requested in the query, it would be carried out in this final merge.

10.2.4 *P(PM)***M*, Physical OIDs, Path Expressions

At first glance, repeatedly partitioning and re-merging appears unnecessary for systems employing physical OIDs where the intermediate join with the *Map* is not needed. In fact, in the simple case of a one-step functional join the variant *P(PM)*^{0}*M* is applied. However, the full-fledged *P(PM)***M*-algorithm is necessary if the query traverses a longer path expression. Consider, for

example, the following query where we want to group to each *Customer* the set of *Manufacturers* from which he or she has ever ordered goods:

select c.Name, (**select** l.ProductRef.ManufRef.Name
from c.OrderRefSet o, o.LineItems l)
from Customers c

Here we assume additional types *Customer* and *Manufacturer* with the attribute *Name*. *Customers* refer via a nested reference set *OrderRefSet* to the given *Orders*. *Manufacturers* are referenced from *Products* via the reference attribute *ManufRef*.

Another query would be to determine the *Customers*' aggregated *Order* volumes:

select c.Name, (**select sum** (l.Quantity * l.ProductRef.Cost)
from c.Orders o, o.LineItems l)
from Customers c

Let us, however, concentrate on the Manufacturer query. The *P*(*PM*)**M* evaluation plans for physical OIDs and logical OIDs are outlined in Figure 10.4 (a) and (b), respectively. Both plans contain two unnesting operations to flatten the *OrderRefSet* and the *LineItems* sets, respectively. When comparing the two plans, they differ mainly in the higher number of functional joins needed for mapping logical OIDs. We assume different *Maps* Map_O, Map_P, and Map_M for *Orders*, *Products*, and *Manufacturers*, respectively. The plan based on physical OIDs draws profit from the interleaved partition/merge (*PM*) steps in the same way as the one based on logical OIDs, i.e., the grouping of *Customers*' *Orders* and their *LineItems* is retained across the successive functional joins. Therefore, the final grouping operation is realized as a (very cheap) merge, in both plans.

10.2.5 Fine Points of the *P*(*PM*)**M*-Algorithm

There are still some fine points in the design of the *P*(*PM*)**M*-algorithm that we have to address.

Obtaining an Order on R The algorithm requires an order on the R tuples for the merge iterators. When comparing tuples from different partitions—e.g., [r_2,6] from RM_{12} and [r_1,5] from RM_{22} in Figure 10.2—it has to be determined in what order r_1 and r_2 were contained in the original R. If there is no such order given by the key on R, an additional sequence number is inserted during the first "flatten and partition" step and used for the succeeding merge steps. Note that all flattened tuples of one R object are assigned the same sequence number.

***Map* Access** For the first partitioning phase of the *P*(*PM*)**M*-algorithm the particular mapping technique has to be taken into account. In general, a partitioning function h_M that achieves range partitioning is favorable. For direct mapping and hash table mapping the difference between range partitioning and "dispersed" partitioning is highlighted as follows:

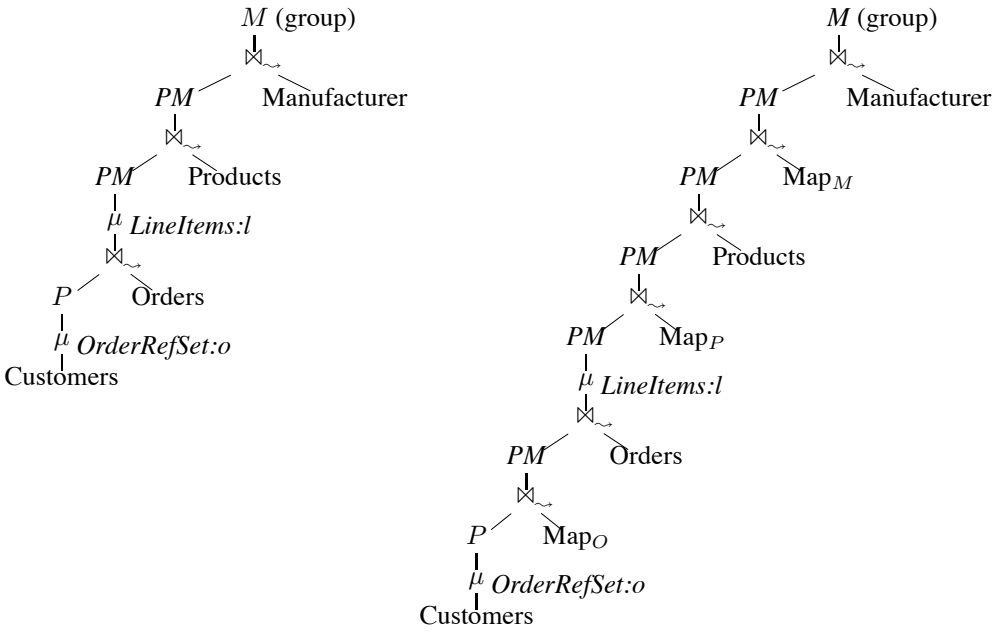
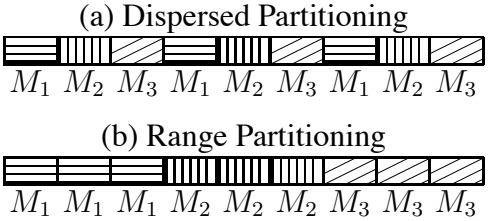


Figure 10.4: Manufacturer Query Using (a) Physical and (b) Logical OIDs



Range partitioning has two advantages: (1) Even if the pages of one partition are individually faulted into the buffer, the disk accesses are all in the same vicinity. (2) Instead of demand paging, range partitioning allows to prefetch the entire partition from disk, thereby transforming random I/O into chained I/O. However, prefetching is only reasonable if (almost) all pages of the *Map* are actually accessed in performing the OID mapping. In a system where the same *Map* is shared by many object types prefetching would not be reasonable.

Achieving range partitioning for direct mapping is quite simple given the range of pages in the *Map* because every logical OID is composed of page identifier and slot within that page.

If a hash table mapping scheme is used, the same hash function has to be applied to the logical OIDs and then the hashed values (containing the page identifier) can be range partitioned.

For a B^+-tree mapping scheme prefetching a partition is only feasible if the pages of one partition are physically adjacent which is typically not the case because of the dynamic growth of the B^+-tree. However, for OIDs that are sequentially generated the B^+-tree may be built (or reorganized) such that adjacent leaf pages are actually physically adjacent

Access to S Similarly to the partitioning step for the *Map* access, range partitioning is beneficial in any following partitioning step if the accessed data structure is clustered and the query engine has enough knowledge about the physical organization. If the partitioning function for the access to S achieves range partitioning on the pages S is stored on, the accesses to S objects belonging to one partition are more localized. Furthermore, instead of faulting in each individual page of S, the whole range of S that is used for a particular partition can be prefetched in large chunks of sequential I/O.

Full Unnesting vs. Retaining Sets The *P*(*PM*)***M*-algorithm as described above flattens the *SrefSet* attribute to a single-valued attribute *Sref*. Instead of fully unnesting *SrefSet*, it is also feasible to keep the set intact as much as possible, i.e., the partitioning operators split the set into subsets each of which containing those elements that belong to the same partition. That is, for every object $r \in R$ there is at most one tuple in every partition. This tuple contains a nested set containing those references belonging to the particular partition. For example, the first partitioning operator gets the complete set *SrefSet* as input and partitions it into at most N tuples and at most one tuple per partition, each of which again contains a set-valued attribute. Referring to the example in Figure 10.2, the first R object [r_1,$\{b, e, c, g, i\}$] would be split into [r_1, $\{b, c\}$] (written to partition R_1) and [r_1, $\{e, g, i\}$] (written to R_2).

This approach avoids multiple flat tuples for the same R object in the same partition; thus it is most beneficial for larger sets *SrefSet*, for small partitioning fan-outs and for non-uniformly distributed references. Of course, keeping the sets requires higher implementation effort. The query engine has to offer "set-aware" variants of some iterators: The partitioning iterator must be capable of splitting nested sets, and the join iterators must iterate through all elements of the nested sets. Our query engine—described briefly in Section 11.1—is capable of processing nested sets.

Projecting R Attributes If "bulky" attributes of R are requested in the result, they may severely inflate the amount of data that is written three times to partition files. To reduce this effect, several measures can be taken: First, the replication of attributes during flattening is unnecessary. Instead, for every $r_i \in R$ the attributes are written only once. Second, since the algorithm retains the order of R, the attributes could be projected out and merged in later for the final result. In contrast to the value-based join and the standard flatten-algorithm, the re-insertion of R attributes is in fact very cheap, since both R and the result have the same order and the R attributes are simply handled as an additional—$(K+1)$-st—input stream of the last merge operator. If the second scan on R would be expensive (e.g., because of high selectivity on R), the bulky attributes of the qualifying R objects might be saved in a temporary segment during the initial scan for reuse in the final merge.

Early Aggregation If aggregation is requested on the result sets in addition to grouping, the aggregation can be folded such that it is already applied to the subgroups belonging to the same R object before they are written to RMS_j. This may result in storage savings for RMS_j. During the final merge, the intermediate aggregation results are then combined. This is easily achieved

for the aggregations *sum*, *min*, *max*, *count* which constitute commutative monoids [GKG+97]—i.e., operations that satisfy associativity and have an identity. For, e.g., *avg* more information has to be maintained to enable early aggregation.

Buffer Allocation The algorithm consists of several consecutive phases, each of which stores its intermediate results entirely on disk. This simplifies database buffer allocation, since the memory available to the query can be allocated exclusively to the current phase. The four phases may be easily derived from the example in Figure 10.2: They are delimited by the three sets of partitions R_i, RM_{ij}, and RMS_j that are stored on disk. Consequently, the four phases are: (1) initial processing of R ending with the first set of partitions R_i, (2) *Map* lookup, (3) dereferencing S, and (4) final merge. For phases (2) and (3), the major amount of memory is allocated to cache the *Map* and S, respectively, and only a small amount is allocated to input and output buffers for the partitions. Summarizing, the $P(PM)^*M$ algorithm is very modest in memory requirements; that is, because of its phased "stop and go"-approach and since it does not require a costly grouping, it tolerates small main memory sizes very well whereas other algorithms easily degrade if main memory is scarce in comparison to the database size.

Chapter 11

Evaluation of Functional Join Algorithms

In order to show the benefits of our *P*(*PM*)**M* algorithm, we implemented the algorithm in a prototype of an object-oriented database system. We compare our algorithm with other join implementations for functional joins which are also implemented in our prototype. Futhermore, we compare these results with measurements for corresponding tests on a commercial object-relational database system. For a broader analysis of functional join algorithms we use a detailed cost model in order to compare the alternative implementations of functional joins under varying scenarios.

11.1 Proof of Concept

To compare the evaluation algorithms, we have implemented them in our iterator-based query engine. In this section we will first outline the implementation of the *P*(*PM*)**M*-algorithm and then describe a few performance measurements we have taken with our query engine.

11.1.1 Partition/Merge-Implementation

Our query engine is based on the iterator model [Gra93] and is implemented in C++. Figure 11.1 gives an outline of the implementation of our *P*(*PM*)**M*-algorithm. The dashed boxes indicate the new special-purpose *PM*-operators that are composed of two "off-the-shelf" iterators. For the basic functional join from R to S along *SrefSet*, processing starts with a scan of R, applying an optional selection predicate and projecting out unwanted attributes, but keeping at least the set-valued attribute *SrefSet* of R and a key for R. *SrefSet* is then flattened by the unnest operator μ yielding tuples with single-valued attribute *Sref*. The first partitioning iterator P_N partitions the input into N partitions based on a partitioning function h_M—as introduced before. The second half of the iterator scans the partitions one at a time and passes the tuples to the functional join with the *Map*, probing every logical OID in *Sref* against the *Map* and replacing it by the address *Saddr*. The partition size and the partitioning function h_M are chosen depending on the OID mapping technique such that the *Map* lookup can be evaluated in memory (see Section 10.2.5). The join output is directly fed into the next partitioning operator PM_{NK}. This time the physical

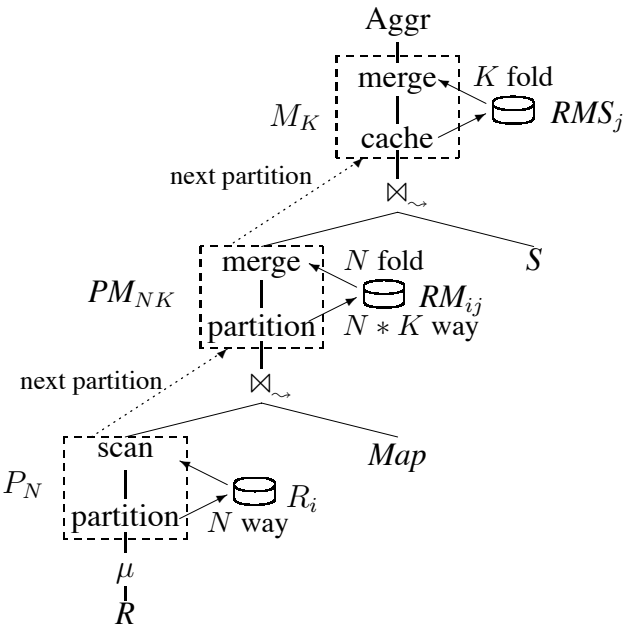


Figure 11.1: *P(PM)*M*-Implementation

OIDs, now stored in *Saddr*, serve as partitioning key, and each of the N input partitions obtained from the prior join is split into K output partitions yielding totally $N * K$ partitions stored on disk. The dotted arrow between P_N and PM_{NK} symbolizes a communication channel that tells PM_{NK} to start a new set of partitions each time an input partition has been completed. Instead of processing all $N * K$ partitions consecutively, we first re-merge those N partitions referring to the same partition of S objects. That is, we merge one matching subpartition from each of the initial N partitions, yielding K partitions as output of the merge operator. Each of the K partitions is then in turn joined with S and the join result is again written to a partition file by an operator called "cache." In a final step, the K partitions are merged to form a single output stream. If we projected out some "bulky" R attributes that are needed in the final result, we would very cheaply re-merge this information in this final merge by simply adding R (or the temporary segment containing the projected data of R) as the $(K + 1)$-st merge stream to the merge operator. If required, an aggregation is performed on the result, as indicated by "Aggr." Note that since the order of R is retained, the result is already grouped by the key of R such that only the aggregation function itself (like *sum*) has to be computed.

11.1.2 Benchmark Setup

The benchmarks were performed on a Sun SparcStation 20 running under Solaris 2.6. The database was held on the operating system disk together with the database system, and another disk was used for temporary files. In order to avoid side effects because of file system caching, the direct I/O option was turned on for all file accesses. Thus, we ensured that *each* file access of the database system definitely caused disk I/O and could not be satisfied from the OS file system

cache. This is especially important for our first experiments on the small database. Writing the result tuples to display/file was suppressed for all queries—also the relational ones (see below).

The database buffer cache was segmented and configured according to the optimizer (cost model) estimation individually for each query plan. For the run time experiments, the total amount of cache available to a query did not exceed 2 MB at any time. Direct mapping was employed to resolve logical OIDs.

We restrict ourselves to the query with aggregation given in Section 10.2. For the query without aggregation, the final aggregation operator would be replaced by a grouping operator. Since for each R object the complete group of *S_Attr* had to be conserved instead of a single aggregation result, all algorithms requiring an explicit grouping, i.e., the flatten-algorithms and the value-based join, would become even more expensive, whereas the algorithms that retain the initial grouping would not suffer much from the larger result.

We have tried to evaluate the query on a commercial relational DBMS with object-relational extensions (O/RDBMS).[1] For this purpose, we have created types and tables as described in Section 10.2. The references were scoped, i.e., they were constrained to point only to a single table (S) using the SQL3-like *scope* clause. However, the query crashed after a few hours with DBMS errors for our initial (larger) database, such that we had to fall back to a smaller database in order to get any results for comparison. The cardinalities of both the small and the larger database are given in Table 11.1. The references in *SrefSet* were distributed uniformly over the full extent of S.

The O/RDBMS was installed on a faster machine (SS20 with CPUs clocked 50% higher and faster disks) and configured with 2 MB buffer—as in our other setup. For comparison, we have also used the same commercial O/RDBMS for a pure (flat) relational representation without using references and nested sets. In this schema, the reference set *SrefSet* has been omitted and an additional table *RS* has been created to implement the association between R and S. Consequently, the *RS* table contained $|R| * 10$ rows. To compensate for the lacking OIDs, the tables R and S were extended to contain additional integer (key) attributes *R_key* and *S_key*, respectively. Tuples of R and S had a size of 204 byte each—the smaller R tuple size in comparison to the R object size of 348 bytes is due to the missing nested reference set in the pure relational schema. The table *RS* contained only the two attributes *RS_Rkey* and *RS_Skey* that served as foreign keys for R and S, respectively. The following query, which (apart from *R_key*) yields the same result as the object-relational one, was measured:

select r.R_key, r.R_Data, **sum**(s.S_Attr)
from R r, S s, RS rs
where rs.RS_Skey = s.S_key **and** rs.RS_Rkey = r.R_key
group by r.R_Key, r.R_Data

11.1.3 Comparison of Measured Running Times

We have created both databases described in Table 11.1 on our prototypical OODBMS and implemented all algorithms described in the previous section. For the plans either a cheap "stream"

[1] Our license prohibits to identify the particular product.

database	$\|R\|$	$\|S\|$	$\|SrefSet\|$	R pages	S pages
small	10000	10000	10	994	667
larger	100000	100000	10	9933	6667

database	*Map* pages	R object size	S object size
small	61	348	228
larger	591	348	228

Table 11.1: Database Cardinalities

aggregation algorithm (only calculating the sum) was employed if the grouping on R was retained (*P*(*PM*)**M* and naive) or a hash aggregation was used if the initial grouping has been destroyed (sort, partition and value-based plans). The value-based plan was implemented using a hybrid hash join with S as build input. For the *P*(*PM*)**M*-algorithm, some of the optimizations described in Section 10.2.5 were implemented: The sets were retained (no full unnesting), range partitioning was applied for the access to S, and the bulky *R_Data* attribute was projected out and re-merged in the final step.

Table 11.2 gives an overview of the observed run times for all algorithms. For comparison, the predictions of our cost model (cf. Section 11.2) are also shown. Furthermore, the run times of the queries on the O/RDBMS are given for two variants: (1) based on the object-relational schema of Section 10.2 with the nested reference set *SrefSet* and (2) on the pure flat relational schema with the additional table *RS*.

The value-based join performs quite well on the small database since the build input (S) is projected to contain only two attributes, the OID and *S_Attr*. It is, however, not cheaper than the *P*(*PM*)**M*-algorithm since the final hash aggregation causes additional cost that does not occur in the *P*(*PM*)**M* plan. On the larger database, it can no longer keep its complete build input in memory and, as a consequence, has to perform an expensive hash aggregation. When comparing the *P*(*PM*)**M* run time to the naive algorithm, there is a performance gap of more than an order of magnitude: On the larger database, the absolute run time of the naive algorithm amounts to more than five *hours*, while our *P*(*PM*)**M*-algorithm requires only less than five *minutes*. The *P*(*PM*)**M*-algorithm also outperforms all the flatten-algorithms, though not as drastically as the naive pointer chasing algorithm. The sort-based flatten plan suffers from high CPU cost for sorting and small run files due to the restricted amount of memory.

For the object-relational schema, the commercial O/RDBMS shows an even worse performance than the naive algorithm. On the other hand, the query on the flat relational schema takes reasonable run time, although for the larger database still more than twice as much as *P*(*PM*)**M* (in spite of the faster host for the O/RDBMS).

small database				
method	our prototype	cost model	commercial O/RDBMS with ref. sets	commercial O/RDBMS flat rels w/o refs
naive	356	461	1110	
flatten/partition	125	136		
flatten/sort	140	168		
value-based	40	56		51
$P(PM)^*M$	29	34		

larger database				
method	our prototype	cost model	commercial O/RDBMS with ref. sets	commercial O/RDBMS flat rels w/o refs
naive	14893	18219	–	
flatten/partition	1868	2029		
flatten/sort	4874	5432		
value-based	1811	1389		721
$P(PM)^*M$	289	295		

Table 11.2: Run Times in Seconds (2 MB Memory, avg. $|SrefSet|$=10, Direct Mapping)

11.2 Analytical Evaluation

In this section, we first present the basics of a cost model which was used to assess functional join algorithms for different scenarios. Thus, this cost model is used as vehicle for a broader analysis. The remainder of this section provides analyses which were conducted with this cost model. Unless stated otherwise, these analyses are based on the larger database as described in Table 11.1. Similarly, the default configuration was chosen as before, i.e., 2 MB of memory was available and logical OIDs were resolved using direct mapping. The labels of the plots are constructed from two parts, the first one describing the access method to the *Map*, the second part describing the access to the S extent. The access methods are: no partitioning (N), i.e., directly chasing each individual pointer, partitioning (P) and merging (M), and sorting (S). The value-based hash join does not fit into this classification and is simply labelled "hashjoin."

11.2.1 The Cost Model

The design of our cost model is strongly influenced by the structure of modern query engines implementing the iterator model. This means that cost estimations are calculated on a per iterator basis. I/O costs are modeled according to [HCLS97] and the CPU operation assumptions are mostly based on [PCV94] and [HR96]. Our cost model contains extensions to deal with set-valued attributes and our new $P(PM)^*M$ algorithm. The cost formulae model disk I/O quite precisely by means of differentiating between seek, latency, and transfer time. As a consequence, we are able to grasp the difference between sequential and random I/O and the influence of the

T_S	average seek time	10.2 ms
T_L	average latency time	5.54 ms
T_T	transfer time for a page (4 KB)	1.7 ms
T_{IO}	time to initiate an I/O operation	1.21 ms
T_{hash}	time to execute a hash function	0.285 ms
T_{add}	time to add two integers	0.00719 ms
T_{probe}	time to test a hash table	0.02339 ms
T_{copy}	time to copy a byte	0.000115 ms
T_{comp}	time to compare two OIDs	0.00719 ms

Table 11.3: Parameters Describing the Hardware

P_R	number of pages in table R (equivalently for *RM* and *RMS*)
$\|R\|$	cardinality of table R
r	average size of an R object
b	read/write buffer size (in pages)
N	number of partitions
$\|SrefSet\|$	average number of elements in the nested reference set

Table 11.4: Variables Used in the Cost Model Formulae

transfer block size. In modeling the CPU costs, we have included those operations that have major influence on CPU time, e.g., sorting, hashing, buffer management (page hit/page fault) and iterator calls.

For ease of presentation, we only describe the cost formulae of the iterators needed for implementing the partition/merge algorithm as shown in Figure 10.2. That is, we only present the formulae for direct mapping and if range partitioning on the references to the *Map* and S and prefetching for reading these sets into main memory are used. For other strategies other formulae apply. We would like to emphasize, however, that we did use the right formulae in order to obtain performance results.

The cost model parameters for modeling the CPU and I/O costs are described in Table 11.3. Note that the constants regarding CPU costs include all instructions related to the operations, e.g., T_{comp} involves pointer arithmetics etc. and not only a single CPU instruction. The cost model variables that describe characteristics of the database are described in Table 11.4.

Analysis of I/O Cost

The *P*(*PM*)***M* algorithm with the above mentioned premises has very similar I/O access patterns throughout all its phases. Therefore, we describe the patterns and list the phases in the algorithm where the pattern shows up. In the cost formulae it is assumed that no inter-operator interference

occurs. The number of additional seeks caused by interference would be calculated separately and added to the cost of the algorithm. Up to now we are only able to model interference if just one disk is used at all. In our benchmarks, however, we used two disks and—although three disks would be necessary to avoid all interference effects in the investigated algorithms—we decided to neglect interference. Furthermore, we assume constant seek times here.

Reading from Disk We denote the number of pages read in one I/O operation as b. The merge operator uses a buffer of b pages for each input partition. The cost for reading *RM* (and also analogously for reading *RMS*) by the merge operator is then given by the following formula:

$$\left\lceil \frac{P_{RM}}{b} \right\rceil \cdot (T_S + T_L + T_{IO}) + P_{RM} \cdot T_T$$

For the initial reading of R the cost can be computed as:

$$T_S + \left\lceil \frac{P_R}{b} \right\rceil \cdot (T_L + T_{IO}) + P_R \cdot T_T$$

For the scan operator reading the first set of partitions R_i $(1 \leq i \leq N)$ we get:

$$N \cdot T_S + \left\lceil \frac{P_R}{b} \right\rceil \cdot (T_L + T_{IO}) + P_R \cdot T_T$$

Here N is the number of partitions generated by the preceding partition operator. For the join operator[2] the same formula can be used, except that R has to be replaced by *Map* or S, respectively.

Writing to Disk We use the same variable b for the buffer size as before. The cost for the write operations of the partition iterator for partitioning R (and also analogously for partitioning *RM*) can be computed by the following formula:

$$\left\lceil \frac{P_R}{b} \right\rceil \cdot (T_S + T_L + T_{IO}) + P_R \cdot T_T$$

The cache iterator which is applied on *RMS* produces smaller cost with its writing operations:

$$T_S + \left\lceil \frac{P_{RMS}}{b} \right\rceil \cdot (T_L + T_{IO}) + P_{RMS} \cdot T_T$$

Analysis of CPU Cost

Again the actions consuming CPU time are listed together with their cost formulae and the iterators performing those actions.

[2]The join operators read *Map* and S.

Copying of Elements The cost formula for copying all elements of a set X in main memory:

$$|X| \cdot x \cdot T_{copy}$$

The small x denotes the size of an element in a set $X \in \{R, \mathit{RM}, \mathit{RMS}\}$. This action is performed by all the iterators writing temporary sets to disk, especially the partition and the cache iterator, and by operators using in-memory working areas like sort and hash.

Comparing Elements The merge iterator has to compare sequence numbers attached to each element for detecting those stemming from the same element of an initial input set. The cost for such an operation is:

$$|R| \cdot |\mathit{SrefSet}| \cdot \log_2(N) \cdot T_{comp}$$

Here the variable N denotes the number of partitions merged into one partition by the merge iterator. Since the ordering of elements is done by a tournament tree, we only have to perform $\log_2(N)$ comparisons for each element in R.

Computing Hash Functions For each join attribute in its input set, the partition iterator has to call a hash function:

$$|R| \cdot |\mathit{SrefSet}| \cdot T_{hash}$$

Performing Aggregation For each element in R we have to add an integer value for every element in the nested set:

$$|R| \cdot |\mathit{SrefSet}| \cdot T_{add}$$

Testing the Buffer Each join iterator in a partition/ merge algorithm uses a buffer for reading S and *Map*. For each join attribute in the set R the join iterator has to look up the buffer for the appropriate page. We assume that this look up is done by accessing a hash table. Then the cost can be computed by:

$$|R| \cdot |\mathit{SrefSet}| \cdot T_{probe}$$

11.2.2 Varying the Memory Size

The running times of the various algorithms under varying memory sizes are reported in Figure 11.2. The NN plan using (naive) pointer chasing both for *Map* lookup and dereferencing S does not even show up in the plot due to its running time of 6'20 hours for 1 MB to 4'10 hours for a 6 MB buffer. The NS query still uses naive *Map* lookup, but sorts the physical OIDs before accessing S. When comparing NS with SS, sorting the flattened R tuples for the *Map* lookup does not pay off because the *Map* is smaller than 2 MB (For 1 MB the sort-based plans are out of the range of the curve because for such small memory configurations they need several merge phases.) Both sort variants suffer from high CPU costs for sorting. The partition plan PP yields already significantly better performance than sort-based plans for small memory sizes. The performance advantage of partitioning over sorting for small memory sizes is due to the large

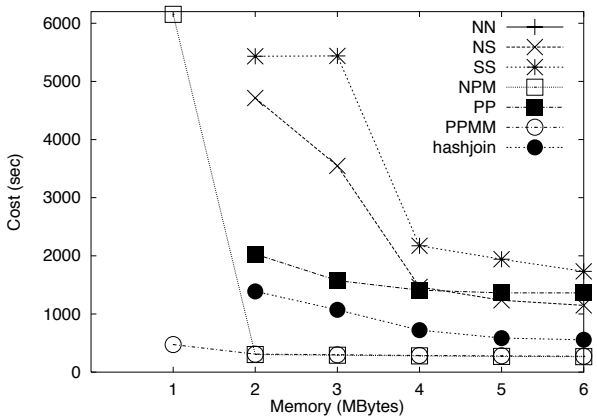


Figure 11.2: Cost Model Results for Larger Database (Direct Mapping, $|SrefSet|$=10)

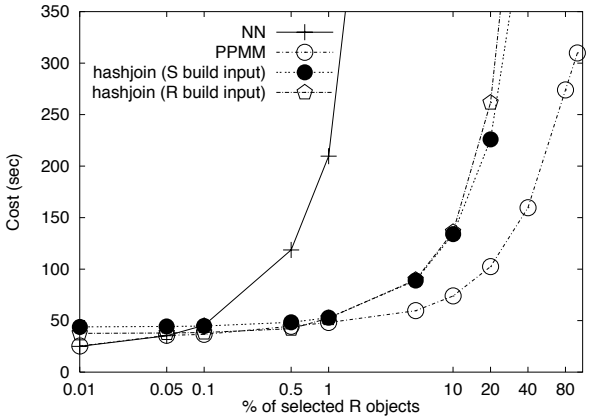


Figure 11.3: Selection on R (Direct Mapping, $|SrefSet|$=10, 2 MB Memory)

number of run files generated for sorting. The value-based hash join performs even better than PP, but is still quite costly compared to the winners PPMM (=$P(PM)^1M$) and NPM (=$P(PM)^0M$). The latter one omits the first partitioning step and shows poor performance for very small memory sizes. For 2 MB and larger, the two plans have the same running time since PPMM uses only one partition for the *Map* access anyway and, therefore, coincides with NPM. The most impressive result of this curve is that the $P(PM)^*M$-algorithm tolerates very small memory sizes under which all other algorithms degrade.

11.2.3 Varying the Selectivity on *R*

In Figure 11.3 the percentage of R objects taking part in the functional joins is varied on the (logarithmically scaled) x-axis. For a small number of R objects, most pages of the *Map* are hit at most once and some pages of S are not referenced at all, such that one might expect a break-even point between $P(PM)^*M$ and the naive algorithm. However, for a high selectivity (e.g., 0.01% corresponding to 10 R objects) they have nearly the same running time. That is, even if there are only very few references to be resolved, there is no significant overhead induced by our $P(PM)^*M$-algorithm. On the other hand, the naive algorithm very quickly degrades if the number of references to be mapped increases. Furthermore, we have plotted the value-based hash join with two configurations, using either R or S as build input. Both variants are, however, worse than $P(PM)^*M$ over the full selectivity range, and for a small number of R objects they are—due to the fix cost for the hash join and hash aggregation—even worse than the naive plan.

11.2.4 Varying the Set Cardinality

In the previous experiments, the number of elements in *SrefSet* was constantly 10. Figure 11.4 shows running times of the algorithms with different set sizes. While the $P(PM)^*M$-algorithm scales linearly, the running times for all others explode. The flatten variants behave poorly. The naive plan suffers from an enormous amount of random I/O (up to $50 * 100,000$ references,

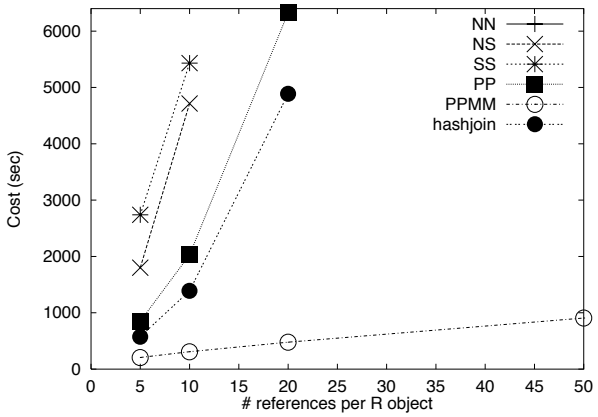


Figure 11.4: Varying the Cardinality of *SrefSet* (2 MB Memory, Direct Mapping)

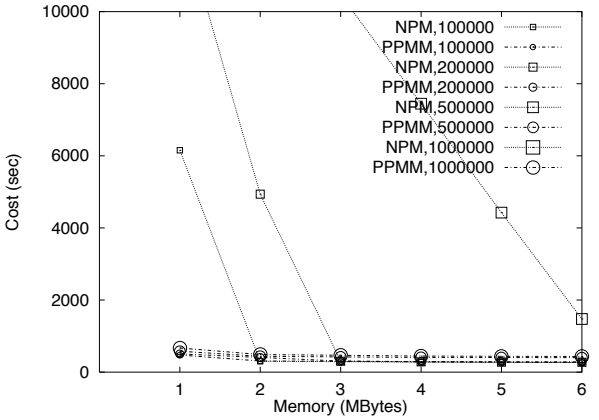


Figure 11.5: Inflating the OID *Map* under Varying Memory Sizes (Direct Mapping)

calculated running time of roughly 25 hours and is therefore not shown) and the flatten plans suffer from large temporary files.

11.2.5 Inflating the OID *Map*

So far we assumed a distinct *Map* for the S objects which, as a consequence, is perfectly clustered. In the following experiment, we analyze the behavior of *P*(*PM*)**M*-algorithms for not-so-well clustered OID *Maps*, as they may occur if there is one global OID *Map* or if only a small fraction of S is referenced, e.g., because of a selection on R. The OID *Map* for S—previously containing 100,000 entries—has been inflated by inserting unused entries—uniformly distributed over all pages of the *Map*—to contain up to one million entries. The NPM and PPMM queries have been run on the standard database (100,000 objects of R and S each, 10 elements in *SrefSet*) with different amounts of memory available. The legend of Figure 11.5 indicates the size of the *Map* (100000, ..., 1000000). The smallest symbols denote the configuration that was used in Figure 11.2, i.e., the *Map* was optimally clustered. For larger *Maps*, the PPMM plan shows only a slight running time increase, caused by the inevitably higher number of I/O accesses to the larger *Map*. However, each *Map* page is fetched from disk only once, since the number of partitions in the first partitioning step is adapted such that one partition of the *Map* can be cached in memory. On the other hand, NPM cannot cope with larger *Maps* since it induces an enormous number of page faults as long as the *Map* does not entirely fit into memory.

Figure 11.6 compares the *P*(*PM*)**M*-algorithm with the value-based hash join in an extreme scenario: The set-valued attribute *SrefSet* contains only three references on average and the *Map* is inflated to contain one million entries—of which 900,000 are obsolete. The number of R and S objects remains at 100,000, respectively. This set-up favors the value-based hash join extremely, since it does not use the *Map* anyway. Furthermore, the hash join draws profit from larger amounts of memory in a larger scale than *P*(*PM*)**M* because of the projection on S: The (projected) S that serves as build input for the hash join can be kept in memory for large memory configurations (beyond 4 MB) such that the join is an in-memory operation. On the other hand,

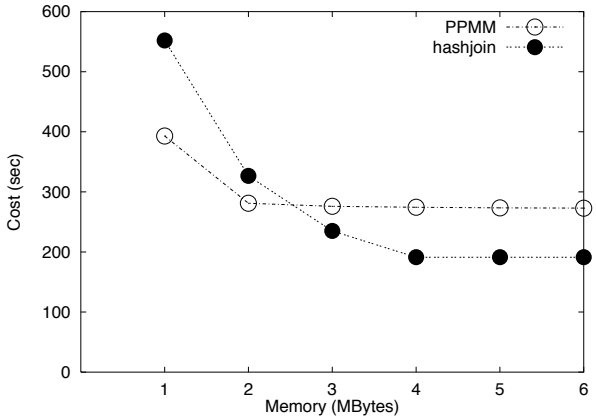


Figure 11.6: Value-Based vs. $P(PM)^*M$ Pointer Join: $|SrefSet|$=3, Direct Mapping, 1,000,000 *Map* Entries

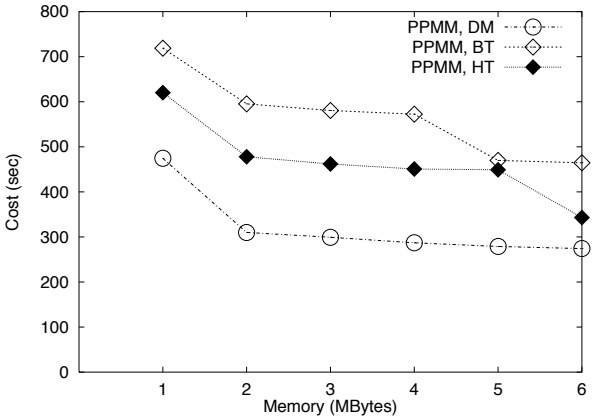


Figure 11.7: Comparison of Different OID Mapping Techniques: DM, BT, and HT, all with $P(PM)^*M$-Algorithm

the $P(PM)^*M$-algorithm loads and keeps the S pages in their entirety in memory. Since the whole S extent of ca. 26 MB still does not fit in memory, the additional memory does not avoid a partitioning step of $P(PM)^*M$ and the flattened R must still be written to disk partitions.

11.2.6 Comparing Different OID Mapping Techniques

Figure 11.7 compares the three OID mapping techniques that we have discussed in Section 10.1.2 for our application, i.e., in combination with the $P(PM)^*M$-algorithm. Both B^+-tree (BT) and hash table mapping (HT) show two performance steps. The first step occurs when increasing memory from 1 MB to 2 MB. Here, the scan and merge operators reach their optimal amount of memory. The second step occurs when the $P(PM)^*M$-algorithm omits the first partitioning phase since the OID mapping structure can be completely cached in memory. Since the total size of the B^+-tree is smaller than that of the hash table,[3] this point is reached with a smaller memory size for the BT curve. In addition, BT is generally more expensive due to higher CPU cost for the tree lookup. The direct mapping (DM) approach is the cheapest: The first partitioning step can already be omitted at a memory size of 2 MB due to the compact representation of the *Map*. Furthermore, the compact storage of the (direct) *Map* reduces the total number of I/O calls. In addition, the CPU overhead for a single *Map* lookup is cheaper for DM than for the other two mapping techniques.

11.2.7 Logical OIDs in Comparison to Physical OIDs

So far, we have assessed our algorithms for different scenarios using logical OIDs. Next, we turn to physical OIDs. This simplifies all algorithms since the extra *Map* lookup operation is omitted. Thus, the algorithms are no partitioning (N), sorting (S), partitioning (P), and $P(PM)^0M$

[3] Due to prefix compression and a specialized splitting procedure [EGK95] the B^+-tree contains more entries per page than the hash table.

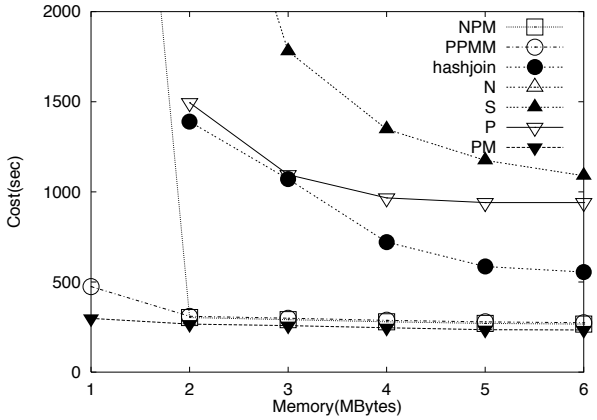


Figure 11.8: Physical OIDs vs. Logical OIDs with Direct Mapping

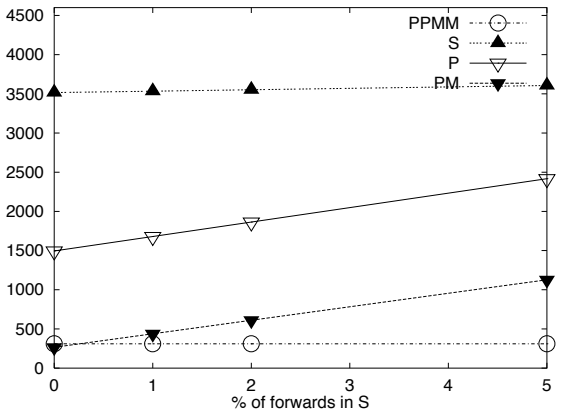


Figure 11.9: Effect of Forwards

(labelled PM). The value-based hash join is independent of the underlying OID realization. For comparison, Figure 11.8 additionally includes the NPM and PPMM plans for logical OIDs realized with direct mapping. The naive plan does not show up in the plot since it ranges between four and five hours. The running time for the partition plan P is similar to the value-based hash join while the sort-based query performs still significantly worse. Not surprisingly, the PM plan performs slightly better than the $P(PM)^*M$ plan for logical OIDs. However, the additional cost of the *Map* lookup is kept at a low level. For example, for 3 MB of memory the PM plan was only 14% cheaper than $P(PM)^*M$.

While physical OIDs are definitely advantageous on a "clean" database, they incur a severe performance penalty in the presence of forwards. We have created a varying percentage of forward references (0% to 5%) in the S extent. Figure 11.9 shows that the sort-based plans are fairly robust against forwards—although at a high cost level—because they "hit" the same forwarded object consecutively whereas the multiple hits of the forwarded object are non-consecutive for partition-based plans. Therefore, sort-based plans need to allocate only one additional page for loading the currently "active" forwarded object whereas partition-based plans need to allocate more buffer for a partition containing forwards. P and PM behave similarly (the lines are parallel), such that partition/merge retains its advantage. For comparison, the PPMM plan under logical OIDs is also shown. Evidently, even for very low levels of forward references (e.g., 1%) logical OIDs are superior to physical OIDs.

Chapter 12

Conclusions

We presented the architecture of a new kind of data integration system which represents the blueprint of the ObjectGlobe system, an open, distributed and secure query processing system targeted for data integration. The goal of the ObjectGlobe architecture is to establish an open marketplace in which data, function, and cycle providers can *offer*/*sell* their services, following some business model which can be implemented on top of ObjectGlobe.

ObjectGlobe provides the technology for a global data integration system which should make proprietary small-scale data integration systems unnecessary. The overall efforts for data integration will be reduced by such a global system since the integration of different data sources has to be performed once and not for every data integration system again. Furthermore, the system supported classification of providers in cycle, data and function providers facilitates a division of work which is a prerequisite for an Internet-wide scalability. Consequently, users and providers can benefit from the resulting synergy effects. Providers can concentrate on their fields of expertise and need not care about the provision of the remaining services which are necessary for a complete query execution. Users profit from query executions which can request services from a broad range of providers.

Techniques to enable such a global data integration system were developed in this work. Our lookup service defines vocabularies for the registration data for all kinds of providers. This meta-data covers all the information necessary to consider the respective service for an automatic composition in a query execution. The query parser extracts relevant meta-data from the lookup-service according to the specification of a query. The optimizer then compiles query evaluation plans which contain all the necessary processing information in order to access the services of selected providers. The number of cycle and data providers in our system can get rather large and therefore, a cluster tree guides the selection of these providers during optimization. The cluster tree contains condensed information about the network neighbourhood of providers and thus it supports the construction of compact QEPs with low network costs. We also showed that the cluster tree can be used to further determine the structure of a QEP which results in a reduced overall response time of resulting query executions. We also showed, how the annotations in a QEP are used to instantiate and distribute a query in our system.

Besides the techniques which provide the functionality of the system, we also eliminated some obstacles which could cause that providers and users are not interested in using our system.

For providers, security concerns have to be taken into account. Authentication and authorization techniques are supplied to protect their services from unauthorized usage. Furthermore, data and function providers can specify restrictions on the cycle providers which are allowed to see their data or code. Cycle providers use an extended sandbox which protects their machines against malicious code which could be loaded from function providers.

On the user side, the effects of query executions must be foreseeable and also manageable. Therefore, we presented extensions for the query optimization, query plan instantiation and query plan execution phases of our query processor in order to support user-defined QoS constraints for queries. One of the main challenges here is that a query uses resources and services of independent and autonomous providers for CPU cycles, data and functions. Based on statistics about providers the optimizer constructs a plan which combines the services of selected providers in a way, that its quality estimates are compliant with the user-defined quality constraints. For each requested service, necessary resource requirements and quality constraints are annotated in the plan and these are monitored during query plan instantiation and execution. If a service quality forecast misses these constraints a fuzzy controller tries to reestablish the constraints by the use of adaptations. In many cases an adaptation claims additional resources which are not used by other queries at that time, in order to react to a predicted quality violation. If no adaptation can be applied, the query is aborted.

Processing functional joins effectively is a key to good performing data integration systems. Each data source which only accepts point queries has to be accessed by a functional join implemented by the corresponding wrapper. A consequent usage of the nested data models which are mainly used for data integration system, will very often require that these functional joins get their arguments from nested sets. We have developed a new algorithm called $P(PM)^*M$ that is based on successively partitioning and merging. This algorithms retains the grouping given by the nested sets within the partitions and restores the overall grouping by efficient merge operations. Our prototype implementation and the quantitative assessment based on a cost model have proven that the algorithm is superior to other methods.

Bibliography

[AC99] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In SIGMOD [SIG99], pages 181–192.

[ACH98] C. Aurrecoechea, A. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems Journal*, 6(3):138–151, 1998.

[AH00] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In SIGMOD [SIG00], pages 261–272.

[ANS92] Database language SQL. Document ANSI X3.135-1992, 1992. Also available as: International Standards Organization Document ISO/IEC 9075:1992.

[AZ96] G. Antoshenkov and M. Ziauddin. Query processing and optimization in Oracle RDB. *VLDB Journal*, 5(4):229–237, 1996.

[BCK98] R. Braumandl, J. Claussen, and A. Kemper. Evaluating functional joins along nested reference sets in object-relational and object-oriented databases. In VLDB [VLD98], pages 110–121.

[BCKK00] R. Braumandl, J. Claussen, A. Kemper, and D. Kossmann. Functional join processing. *The VLDB Journal*, 8(3-4):156–177, 2000. Invited Contribution to the Special Issue "Best of VLDB 98".

[BCV99] S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. *ACM SIGMOD Record*, 28(1):54–59, March 1999.

[Bel52] R. Bellman. On the theory of dynamic programming. In *Proceedings of the National Academy of Sciences*, volume 38, pages 716 – 719, Washington, D.C., 1952.

[BG99] D. Brickley and R. V. Guha. Resource Description Framework (RDF) Schema Specification. Proposed Recommendation `http://www.w3.org/TR/PR-rdf-schema`, WWW-Consortium, March 1999.

[BJS99] E. Bertino, S. Jajodia, and P. Samarati. A flexible authorization mechanism for relational data management systems. *ACM Transactions on Information Systems*, 17(2):101–140, 1999.

[BK89] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. Knowledge and Data Engineering*, 1(2):196–214, 1989.

[BKK99] R. Braumandl, A. Kemper, and D. Kossmann. Database patchwork on the Internet (project demo description). In SIGMOD [SIG99], pages 550–552.

[BKK+01a] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. *The VLDB Journal: Special Issue on E-Services*, 10(3):48–71, August 2001.

[BKK+01b] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, S. Seltzsam, and K. Stocker. ObjectGlobe: Open Distributed Query Processing Services on the Internet. pages 64–70. March 2001.

[BLR98] K. S. Beyer, M. Livny, and R. Ramakrishnan. Protecting the quality of service of existing information systems. In *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems*, New York, USA, August 1998.

[BMCL94] K. P. Brown, M. Mehta, M. J. Carey, and M. Livny. Towards automated performance tuning for complex workloads. In VLDB [VLD94], pages 72–84.

[BP95] A. Biliris and E. Panagos. A high performance configurable storage manager. In *Proc. IEEE Conf. on Data Engineering*, pages 35–43, Taipeh, Taiwan, 1995.

[BSG00] Tom Barclay, Donald R. Slutz, and Jim Gray. Terraserver: A spatial data warehouse. In Chen et al. [SIG00], pages 307–318.

[C+95] M. Carey et al. Towards heterogeneous multimedia information systems. In *Proc. of the Intl. Workshop on Research Issues in Data Engineering*, pages 124–131, March 1995.

[CCI88] CCITT International Telegraph and Telephone Consultative Committee. The Directory. Technical Report Recommendations X.500, X.501, X.509, X.511, X.518-X.521, CCITT, 1988.

[CD99] L. Cardelli and R. Davies. Service combinators for Web computing. *IEEE Trans. Software Eng.*, 25(3):309–316, May 1999.

[CDF+94] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up persistent applications. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–394, Minneapolis, MI, USA, May 1994.

[CHS99] Francis Chu, Joseph Y. Halpern, and Praveen Seshadri. Least expected cost query optimization: An exercise in utility. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 138–147, Philadelphia, Pennsylvania, USA, May 1999. ACM Press.

[CK98] M. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In VLDB [VLD98], pages 158–169.

[Cod70] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[CS96] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In VLDB [VLD96], pages 87–98.

[CSL+90] M. J. Carey, E. Shekita, G. Lapis, B. Lindsay, and J. McPherson. An incremental join attachment for Starburst. In VLDB [VLD90], pages 662–673.

[CZH+99] S. Czerwinsky, B. Zhao, T. Hodes, A. Joseph, and R. H. Katz. An Architecture for a Secure Service Discovery Service. In *Proc. of ACM MOBICOM Conference*, pages 24–35, Seattle, USA, August 1999.

[DA99] T. Dierks and C. Allen. The TLS Protocol Version 1.0. `ftp://ftp.isi.edu/in-notes/rfc2246.txt`, January 1999.

[DLM93] D. DeWitt, D. Lieuwen, and M. Mehta. Parallel pointer-based join techniques for object-oriented databases. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, San Diego, CA, USA, January 1993.

[EGK95] A. Eickler, C. Gerlhof, and D. Kossmann. A performance evaluation of OID mapping techniques. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 18–29, Zürich, Switzerland, September 1995.

[FFK+98] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: experiences with a web-site management system. In SIGMOD [SIG98], pages 414–425.

[FJK96] M. Franklin, B. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In SIGMOD [SIG96], pages 149–160.

[FKK96] A. Frier, P. Karlton, and P. Kocher. *The SSL 3.0 Protocol*. Netscape Communications Corp., `http://home.netscape.com/eng/ssl3`, November 1996.

[FKL97] D. Florescu, D. Koller, and A. Y. Levy. Using probabilistic information in data integration. In VLDB [VLD97], pages 216–225.

[FLMS99] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In SIGMOD [SIG99], pages 311–322.

[Ger96] C. A. Gerlhof. *Optimierung von Speicherzugriffskosten in Objektbanken: Clustering und Prefetching*. PhD thesis, Universität Passau, Fakultät für Mathematik und Informatik, D-94030 Passau, 1996. Dissertation, Universität Passau.

[GGT96] G. Gardarin, J.-R. Gruser, and Z.-H. Tang. Cost-based selection of path expression processing algorithms in object-oriented databases. In VLDB [VLD96], pages 390–401.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 9–18, San Diego, CA, USA, June 1992.

[GHR97] A. Gupta, V. Harinarayan, and A. Rajaraman. Virtual data technology. *ACM SIGMOD Record*, 26(4):57–61, December 1997.

[GI97] M. Garofalakis and Y. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In VLDB [VLD97], pages 296–305.

[GKG+97] T. Grust, J. Kröger, D. Gluche, A. Heuer, and M. H. Scholl. Query evaluation in CROQUE – calculus and algebra coincide. In *Proc. British National Conference on Databases (BNCOD)*, pages 84–100, London, UK, July 1997.

[GLSW94] P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang. Query optimization in the IBM DB2 family. Technical Report RJ9734, IBM Almaden Research Center, March 1994.

[GMSvE98] M. Godfrey, T. Mayr, P. Seshadri, and T. v. Eicken. Secure and portable database extensibility. In SIGMOD [SIG98], pages 390–401.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[GRZZ00] J. R. Gruser, L. Raschid, V. Zadorozhny, and T. Zhan. Learning response time for websources using query feedback and application in query optimization. *The VLDB Journal*, 9(1):18–37, May 2000.

[GW89] G. Graefe and K. Ward. Dynamic query evaluation plans. In SIGMOD [SIG89], pages 358–366.

[GWBC99] S. Gribble, M. Welsh, E. Brewer, and D. Culler. The MultiSpace: an evolutionary platform for infrastructural services. In *Proc. of the Usenix Annual Technical Conference*, Monterey, CA, June 1999.

[HCL+90] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[HCLS97] L. Haas, M. Carey, M. Livny, and A. Shukla. Seeking the truth about *ad hoc* join costs. *The VLDB Journal*, 6(3):241–256, 1997.

[HFLP89] L. Haas, J. C. Freytag, G. Lohman, and H. Pirahesh. Extensible query processing in starburst. In SIGMOD [SIG89], pages 377–388.

[HFPS99] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. `http://www.rfc-editor.org/rfc/rfc2459.txt`, January 1999.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In VLDB [VLD97], pages 276–285.

[HPI99] Hewlett Packard Inc. Chai: Internet business solutions. `http://www.chai.hp.com/`, 1999.

[HR96] E. Harris and K. Ramamohanarao. Join algorithm costs revisited. *The VLDB Journal*, 5(1):64–84, 1996.

[IFF+99] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An Adaptive Query Execution Engine for Data Integration. In SIGMOD [SIG99], pages 299–310.

[IK91] Y. Ioannidis and Y. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 168–177, Denver, CO, USA, May 1991.

[INSS92] Y. Ioannidis, R. Ng, K. Shim, and T. Sellis. Parametric query optimization. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 103–114, Vancouver, Canada, August 1992.

[Ita93] Itasca Systems Inc. Technical summary for release 2.2, 1993. Itasca Systems, Inc., 7850 Metro Drive, Mineapolis, MN 55425, USA.

[JKR99] V. Josifovski, T. Katchaounov, and T. Risch. Optimizing queries in distributed and composable mediators. In *Proc. of the IFCIS International Conference on Cooperative Information Systems*, pages 291 – 302, Edinburgh, Scotland, 1999.

[KC86] S. N. Khoshafian and G. P. Copeland. Object identity. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 408–416, November 1986.

[KD98] N. Kabra and D. DeWitt. Efficient mid-query re-optimization for sub-optimal query execution plans. In SIGMOD [SIG98], pages 106–117.

[KE99] A. Kemper and A. Eickler. *Datenbanksysteme – Eine Einführung (3. Auflage)*. R. Oldenbourg Verlag, 1999.

[KKKK02] M. Keidl, A. Kreutz, A. Kemper, and D. Kossmann. A publish & subscribe architecture for distributed metadata management. In *Proc. IEEE Conf. on Data Engineering*, San Jose, Ca, USA, 2002.

[KM90] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 364–374, Atlantic City, NJ, USA, April 1990.

[KM94] A. Kemper and G. Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science*. Prentice Hall, Englewood Cliffs, NJ, USA, 1994.

[Kos01] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 2001. Accepted for publication. To appear.

[Kri98] N. Krivokapić. *Control mechanisms in distributed object bases: Synchronization, deadlock detection, migration*, volume 54 of *Dissertationen zu Datenbanken und Informationssystemen*. infix-Verlag, Ringstr. 32, 53757 Sankt Augustin, 1998. ISBN: 3-89601-454-4, Dissertation, Universität Passau, Germany.

[KS98] D. Konopnicki and O. Shmueli. Information gathering in the world wide web: The W3QL query languge and the W3QS system. *ACM Trans. on Database Systems*, 23(4):369–410, December 1998.

[KS00] D. Kossmann and K. Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Trans. on Database Systems*, 25(1):43–82, March 2000.

[KY95] G. J. Klir and B. Yuan. *Fuzzy Sets and Fuzzy Logic*. Prentice Hall, 1995.

[LKK+97] P. Lockemann, U. Kölsch, A. Koschel, R. Kramer, R. Nikolai, M. Wallrath, and H.-D. Walter. The network as a global database: Challenges of interoperability, proactivity, interactiveness, legacy. In VLDB [VLD97], pages 567–574.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, 1991.

[LMB97] L. Leverenz, R. Mateosian, and S. Bobrowski. *Oracle8 Server – Concepts Manual*. Oracle Corporation, Redwood Shores, CA, USA, 1997.

[LN99] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal of Selected Areas in Communication*, 17(9):1632–1650, 1999.

[Loh88] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, Chicago, IL, USA, May 1988.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In VLDB [VLD96], pages 251–262.

[MMM97] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the World Wide Web. *Int. Journal on Digital Libraries*, 1(1):54–67, 1997.

[MRT98] G. A. Mihaila, L. Raschid, and A. Tomasic. Equal Time for Data on the Internet with WebSemantics. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, volume 1377 of *Lecture Notes in Computer Science (LNCS)*, pages 87–101, Valencia, Spain, March 1998. Springer-Verlag.

[MZ95] T. J. Mowbray and R. Zahavi. *The Essential Corba – Systems Integration Using Distributed Objects*. John Wiley & Sons, Chichester, UK, 1995.

[NLF99] Felix Naumann, Ulf Leser, and Johann Christoph Freytag. Quality-driven integration of heterogenous information systems. In VLDB [VLD99], pages 447–458.

[O2T94] O_2 Technology, Versailles Cedex, France. *A Technical Overview of the O_2 System*, July 1994.

[Oak98] S. Oaks. *Java Security*. O'Reilly & Associates, Sebastopol, CA, USA, 1998.

[OL90] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In VLDB [VLD90], pages 314–325.

[PCL95] H. Pang, M. J. Carey, and M. Livny. Multiclass query scheduling in real-time database systems. *IEEE Trans. Knowledge and Data Engineering*, 7(4), August 1995.

[PCV94] J. Patel, M. Carey, and M. Vernon. Accurate modeling of the hybrid hash join algorithm. In *Proc. of the ACM SIGMETRICS*, pages 56–66, Santa Clara, CA, May 1994.

[Pet99] J. Petit. Real Estate DTD. `http://www.4thworldtele.com`, May 1999.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A Query Translation Scheme for Rapid Implementation of Wrappers. In *Proc. of the Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 161–186, December 1995.

[PH92] J. Hellerstein H. Pirahesh and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In Michael Stonebraker, editor, *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 39–48, San Diego, USA, June 1992.

[PIHS96] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In SIGMOD [SIG96], pages 294–305.

[PKI] Public-Key Infrastructure (X.509) (PKIX). `http://www.ietf.org/html.charters/pkix-charter.html`.

[PY00] C. H. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access of web sources. In *Proc. of the 41st Annual Symposium on Foundations of Computer Science*, November 2000.

[RBKW91] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Trans. on Database Systems*, 16(1):88–131, March 1991.

[ROH99] M. Tork Roth, F. Ozcan, and L. Haas. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In VLDB [VLD99], pages 599–610.

[RSA99] RSA Laboratories. PKCS #5 v2.0: Password-Based Cryptography Standard. `ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf`, March 1999.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, May 1979.

[SAL+96] M. Stonebraker, P. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A Wide-Area Distributed Database System. *The VLDB Journal*, 5(1):48–63, January 1996.

[SBK01] S. Seltzsam, S. Börzsönyi, and A. Kemper. Security for distributed E-Service Composition. In *Workshop on Technologies for E-Services*, Rome, Italy, September 2001.

[SC90] E. Shekita and M. Carey. A Performance Evaluation of Pointer-Based Joins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 300–311, Atlantic City, NJ, May 1990.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[SIG89] *Proc. of the ACM SIGMOD Conf. on Management of Data*, Portland, OR, USA, May 1989.

[SIG96] *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, June 1996.

[SIG98] *Proc. of the ACM SIGMOD Conf. on Management of Data*, Seattle, WA, USA, June 1998.

[SIG99] *Proc. of the ACM SIGMOD Conf. on Management of Data*, Philadelphia, PA, USA, June 1999.

[SIG00] *Proc. of the ACM SIGMOD Conf. on Management of Data*, Dallas, USA, June 2000.

[SLR97] P. Seshadri, M. Livny, and R. Ramakrishnan. The case for enhanced abstract data types. In VLDB [VLD97], pages 66–75.

[SR86] M. Stonebraker and L. Rowe. The design of POSTGRES. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 340–355, Washington, USA, June 1986.

[SS86] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[Ste95] M. Steinbrunn. *Heuristic and Randomised Optimisation Techniques in Object-Oriented Database Systems*. Dissertation, Universität Passau, 94030 Passau, Germany, 1995.

[Sto96] M. Stonebraker. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1996.

[Sun99] Sun Microsystems, `http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html`. *Java Virtual Machine Profiler Interface (JVMPI)*, 1999.

[TLS] Transport Layer Security (TLS). `http://www.ietf.org/html.charters/tls-charter.html`.

[TRV98] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Acccess to Distributed Heterogeneous Data Sources with DISCO. *IEEE Trans. Knowledge and Data Engineering*, 10(5):808–823, October 1998.

[UDD00] Universal Description, Discovery and Integration (UDDI) Technical White Paper. White Paper, Ariba Inc., IBM Corp., and Microsoft Corp., September 2000. `http://www.uddi.org/`.

[UFA98] T. Urhan, M. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In SIGMOD [SIG98], pages 130–141.

[Val87] P. Valduriez. Join indices. *ACM Trans. on Database Systems*, 12(2):218–246, June 1987.

[Ver97] Versant Object Technology. Versant release 5, October 1997. `http://www.versant.com/`.

[VLD90] *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Brisbane, Australia, August 1990.

[VLD94] *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Santiago, Chile, September 1994.

[VLD96] *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Bombay, India, September 1996.

[VLD97] *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, August 1997.

[VLD98] *Proc. of the Conf. on Very Large Data Bases (VLDB)*, New York, USA, August 1998.

[VLD99] *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Edinburgh, GB, September 1999.

[VM96] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian product. In SIGMOD [SIG96], pages 35–46.

[WA91] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, pages 68–77, Miami, Fl, USA, December 1991.

[Wal99] J. Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82, 1999.

[Wei98] G. Weikum. On the ubiquity of information services and the absence of guaranteed service quality. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, volume 1377 of *Lecture Notes in Computer Science (LNCS)*, pages 3–6. Springer, 1998.

[Wei99] G. Weikum. Towards guaranteed quality and dependability of information services. In *Proc. GI Conf. on Database Systems for Office, Engineering, and Scientific Applications (BTW)*, Informatik aktuell, New York, Berlin, etc., 1999. Springer-Verlag.

[WHK97] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). `ftp://ftp.isi.edu/in-notes/rfc2251.txt`, December 1997.

[Wie93] G. Wiederhold. Intelligent integration of information. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 434–437, Washington, DC, USA, May 1993.

[WSP97] R. Wolski, N. Spring, and C. Peterson. Implementing a performance forecasting system for metacomputing: The network weather service. In *Proceedings of Supercomputing'97 (CD-ROM)*, San Jose, CA, November 1997. ACM SIGARCH and IEEE.

[XH94] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In VLDB [VLD94], pages 522–533.

Appendix A

The XML Representation of a Query Execution Plan

```
<?xml version="1.0" encoding='ISO-8859-1'?>

<plan>
  <iterator id="display" code="iterators.display" cycle-provider="client">
    <iterator id="join1" code="iterators.NestedLoops"
              codebase="functionProvider"
              cycle-provider="client">
      <predicate>Sb = Tb</predicate>
      <iterator id="join2" code="iterators.NestedLoops"
                cycle-provider="alpha">
        <predicate>Ra = Sa</predicate>
        <iterator id="wrapperS" code="wrapper.wrap_S"
                  codebase="functionProvider" cycle-provider="alpha">
        </iterator>
        <iterator id="tbscanR" code="iterators.TbScan" cycle-provider="alpha">
          <partition>R</partition>
        </iterator>
      </iterator>
      <iterator id="thumb1" code="thumbnail" codebase="functionProvider"
                cycle-provider="beta">
        <toThumbNail>picture</toThumbNail>
        <iterator id="tbscanT" code="iterators.TbScan" cycle-provider="beta">
          <partition>T</partition>
        </iterator>
      </iterator>
    </iterator>
  </iterator>

  <provider-information>
    <og-provider id="client">
     <dn-name>C=DE,O=University of Passau,OU=Department
              for Mathematics and Computer Science,
              CN=Mets.fmi.uni-passau.de
     </dn-name>
     <host-dns>Mets.fmi.uni-passau.de</host-dns>
    </og-provider>
    <og-provider id="alpha">
     <dn-name>C=COM,O=A Incorporated,OU=Computing Center,CN=alpha.A.com
     </dn-name>
     <host-dns>alpha.A.com</host-dns>
    </og-provider>
    <og-provider id="beta">
     <dn-name>C=COM,O=B Incorporated,OU=Computing Center,CN=beta.B.com
```

```
     </dn-name>
     <host-dns>beta.B.com</host-dns>
    </og-provider>
    <og-provider id="functionProvider">
     <dn-name>C=COM,O=FctProv Incorporated,OU=Software Development,
              CN=FctProv.com
     </dn-name>
     <code-location>http://www.FctProv.com/forGlobalUse</code-location>
    </og-provider>
  </provider-information>
</plan>
```

Appendix B

The RDF Registration Code for a Collection

In the sample RDF-description shown below, the relevant information about a data provider can be found enclosed in the `DataProvider` element. It contains information about the name of the provider and a URL with which the data provider can be contacted. The `Partition` element contains information about a collection that the data provider makes available. At the beginning of the collection description we can find the data provider of the collection, a plain-text description of the content of the collection, the theme (i.e., HotelTheme) this collection is associated with, etc. The element `wrapper` specifies a reference for the wrapper which performs the necessary transformations to integrate the collection into an ObjectGlobe system. More interesting is the content of the `attributes` element. It contains the description of the type of the tuples, given by the collection. In our case the type contains three attributes and for each attribute the name and the type of the attribute are specified. It is possible to insert additional information about attributes which is omitted for brevity.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns="http://www.db.fmi.uni-passau.de/~objglobe/ObjectGlobe-Metaschema.rdf#">

<DataProvider rdf:ID="HotelBook">
  <dataProviderName>HotelBook</dataProviderName>
  <dataProviderUrl>http://www.hotelbook.com</dataProviderUrl>
</DataProvider>

<Partition rdf:ID="HotelBookPartition">
  <dataProvider rdf:resource="#HotelBook"/>
  <partitionDescription>Description of hotels worldwide</partitionDescription>
  <theme rdf:resource="file:/home/objglobe/Themes.rdf#HotelTheme"/>
  <localName>hotelBookPartition</localName>
  <wrapper rdf:resource="file:/home/objglobe/Operators.rdf#HotelBookWrapper"/>
  <uniqueID>4711</uniqueID>
  <cardinality>30000</cardinality>

  <attributes>
    <rdf:Bag>
      <rdf:li><Attribute>
```

```
        <topic rdf:resource="file:/home/objglobe/Themes.rdf#cityTopic" />
        <domain rdf:resource="file:/home/objglobe/Themes.rdf#StringDomain" />
      </Attribute></rdf:li>
      <rdf:li><Attribute>
        <topic rdf:resource="file:/home/objglobe/Themes.rdf#addressTopic" />
        <domain rdf:resource="file:/home/objglobe/Themes.rdf#StringDomain" />
      </Attribute></rdf:li>
      <rdf:li><Attribute>
        <topic rdf:resource="file:/home/objglobe/Themes.rdf#priceTopic" />
        <domain rdf:resource="file:/home/objglobe/Themes.rdf#IntegerDomain" />
      </Attribute></rdf:li>
     </rdf:Bag>
  </attributes>
</Partition>
</rdf:RDF>
```