Distributed Optimization of Nested Queries

Jacek Skowronek

CTIT Ph.D. Thesis series No. 97-20
P.O. Box 217, 7500 AE Enschede, The Netherlands
telephone +31-53-4893779, fax +31-53-4894524

Samenstelling van de promotiecommissie:

Prof. dr. P.M.G. Apers (promotor)
Dr. H.M. Blanken
Prof. dr. P.M.E. de Bra, Technische Universiteit Eindhoven
Prof. dr. S.J. Mullender
Prof. M. Tamer Özsu, University of Alberta
Dr. A.N. Wilschut (referent)

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems.

ISBN: 90 365 10686
ISSN: 1381-3617; no. 97-20 (CTIT Ph.D.-thesis series)

DISTRIBUTED OPTIMIZATION OF NESTED QUERIES

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. F. A. van Vught,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 4 december 1997 te 13.15 uur

door

Jacek Skowronek

geboren op 11 april 1966
te Gdynia

Dit proefschrift is goedgekeurd door:

Prof. dr. P.M.G. Apers (promotor)

Table of contents

I. Introduction I-1
1. Preliminaries. I-2
1.1 Query optimization I-2
1.2 Scalability I-3
1.3 Distributed computation I-3
2. Scope I-4
3. Contributions I-5
3.1 Unnesting strategy. I-5
3.2 Analytical model and measurements. I-6
3.3 Distributed allocation model I-7
4. Outline I-7
II. Query optimization and resource allocation in client-server database systems. . . . II-1
1. Optimization of database query languages. II-2
1.1 Data models, query languages, and algebras. II-4
1.2 Optimizer architectures II-6
1.2.1 Traditional query optimizer: System R II-7
1.2.2 Rule-extensible optimizer: Starburst II-8
1.2.3 Optimizer generators: EXODUS & Volcano. II-9
1.2.4 Strategy-extensible optimizers II-11
1.2.5 Formal representation of transformations II-13
1.2.6 Formal representations of optimization strategy II-14
2. Distributed query processing II-15
2.1 Query processing in client-server systems II-15
2.2 Solving scalability problems in database systems. II-16
2.2.1 Hierarchic models of resource allocation. II-17
2.2.2 Market-based approaches II-18
2.2.3 Existing solutions to scalability problems of query processing systems II-20
III. Optimization of nested queries III-1
1. Strategy goal. III-1
2. Input and output of the optimizer III-2
3. Strategy for unnesting and splitting III-6
3.1 Global description of the strategy III-6
3.2 Notational conventions III-8
3.3 Unnesting component III-8
3.3.1 Unnesting rule set. III-8
3.3.2 Applicability conditions for unnesting rules III-11
3.3.3 Ordering of unnesting transformations III-12
3.4 Splitting component III-18
3.4.1 Splitting rules III-18
3.4.2 Splitting alternatives. III-19

3.4.3 Splitting of disjunctive predicates III-20
3.4.4 Splitting of conjunctive predicates III-23
3.4.5 Splitting strategy: remarks III-26
3.5 Overall unnesting algorithm III-26
3.5.1 Unnesting algorithm: example III-28
3.6 Other optimization stages III-31
3.6.1 Standardization III-31
3.6.2 Normalization III-32
4. Evaluation of the strategy III-33
4.1 Nested quantifiers III-34
4.2 Nested collects III-35
4.3 Comparison with existing approaches III-36
4.3.1 Cluet & Moerkotte III-36
4.3.2 Steenhagen et al. III-36
4.3.3 Mitchell et al. III-36
4.3.4 Cherniack et al. III-37
5. Extensions and further research III-38
5.1 Handling quantifiers within collect functions III-38
5.2 Remaining problems and extensions III-41
6. Conclusions III-43
IV. Scalable distributed optimization IV-1
1. Introduction IV-1
2. Requirements for database applications on internet and intranet IV-2
2.1 Environmental analysis: conclusions IV-3
3. Component allocation models IV-4
3.1 Centralized allocation of optimizer components IV-4
3.2 Distributed allocation IV-7
3.2.1 Principles of query optimizer economy IV-8
3.2.2 Optimizer economy: definitions IV-10
3.2.3 Funding and pricing IV-10
3.2.4 Consumer bidding IV-11
3.2.5 Component broker: auctioning component sequences IV-15
3.2.6 Reluctant consumers IV-16
3.2.7 Eager consumers IV-16
3.2.8 Deformations in consumer behavior IV-17
3.2.9 Partition allocation IV-18
3.3 Comparison with current approaches IV-18
4. Conclusions IV-19
V. Architecture for distributed query optimization V-1
1. Requirements V-1
2. Representation of query expression V-3
3. Representation of transformations and rulesets V-5
3.1 Applicability checks and transformation functions V-5
3.2 Representation of rule sets V-6
4. Distribution aspects V-6
4.1 Implementation platform V-6

4.1.1 Distributed computing platforms V-7
4.1.2 Rationale for optimizer implementation platform V-9
4.2 Transport of query representation V-10
4.3 Client-side data dictionary V-11
4.4 Operation of the Joquer query optimizer. V-13
5. Evaluation of implementation V-15
5.1 Comparison to existing approaches. V-15
5.2 Optimizer architecture evaluation V-18
5.3 Distribution aspects V-18
6. Summary V-20
VI. Performance analysis of the Joquer distributed query optimizer VI-1
1. Approach VI-1
2. Intuitive description of the performance problem VI-3
2.1 Client population as a factor VI-8
2.2 Component partition as a factor VI-9
3. Analytical model VI-11
3.1 Terminology VI-11
3.2 The model formulas VI-13
4. Measurements. VI-15
4.1 Measurement setting VI-16
4.2 Measurement: time differences within the query batch VI-17
4.3 Measurement: elements of query processing VI-19
4.4 Measurement: elapsed batch optimization time in relation to batch size VI-21
4.5 Measurement: elapsed batch optimization time in relation to client population. . VI-24
5. Model results VI-26
5.1 Model parameter levels VI-26
5.2 Model: elapsed time as a function of query batch size VI-28
5.3 Model: elapsed batch optimization time in relation to client population size. . . . VI-29
6. Model extrapolation VI-30
6.1 Extrapolation: higher marshalling speed. VI-31
6.2 Extrapolation: addition of physical optimization VI-31
6.3 Extrapolation: higher processing costs VI-33
6.4 Extrapolation: no deployment costs VI-34
6.5 Extrapolation: separate component server VI-34
7. Possible extensions and further research VI-36
8. Conclusions VI-38
VII. Summary and future research VII-1
1. Summary VII-1
2. Proposals for future research VII-2
3. Final remarks VII-3
Appendix A. Rule set analysis A-1
1. Modifications of the initial rule set A-1
1.1 Inconsistencies between rule sets A-1
1.2 Inconsistent splitting rules. A-2
1.3 Rule notation A-3
1.4 Analysis of the initial optimization A-4

1.4.1 Improper rule structuring A-4
Appendix B. Transformation rules B-1
1. Standardization rules B-1
2. Normalization rules B-1
2.1 Transformation to PNF B-2
2.2 Transformation to MNF B-3
3. Translation rules B-3
3.1 Unnesting rules B-3
3.2 Splitting rules B-4
3.2.1 Splitting predicates B-4
3.2.2 Splitting functions B-5
3.3 Pushing rules B-5
Appendix C. Example optimizations C-1
1. Test queries C-1
2. Example queries C-2
2.1 c = disjunction, Q = existential C-2
2.2 c = disjunction, Q = negated existential C-5
2.3 c = conjunction, Q = existential C-7
2.4 c = conjunction, Q = negated existential C-10
Appendix D. Analytical model for distinct component server D-1
Appendix E. Performance model analysis E-1
1. Model: batch size as factor in the Internet context. E-1
2. Model: client population as a factor in the Internet case E-3
3. Model: elapsed time as a function of marshalling speed. E-4
4. Model: elapsed batch optimization time as a function of the difference between the client and the server E-5
5. Model: elapsed time as a function of network transfer speed E-7
Literature list i-1
Index ii-1
Samenvatting iii-1
Acknowledgments iv-1

I. Introduction

Among different applications of computing, database systems play an increasingly important role: storage and retrieval of growing amounts of data. The database systems are used in day-to-day operation of commercial organizations, in control of real-time processes, and in management of huge quantities of measurement results. As application areas of database systems continue to expand, new requirements are posed on their modeling capabilities and their ability to retrieve the desired data in a timely manner. The techniques used for this retrieval, known under the names of query processing and query optimization, are the focus of our research. In our research, we analyze the techniques allowing query optimization to cope with two of the biggest challenges posed in modern database applications: the challenges of **complexity** and **scalability**.

The **complexity challenge** has been the focus of many research efforts; those efforts attempt to provide the query processing techniques for increasingly complex modeling instruments and query languages. These instruments often involve introduction of intermediary languages (algebras), used for manipulation and transformation of the query into a more efficient form. Although these languages and transformations allow more efficient execution of the query, they also make the process of its optimization more complex. It is often the case that the optimization process itself has to be guided by complex strategies to be successful and efficient: such a strategy, and its validation through implementation and testing, is one of the contributions of this thesis.

A database system cannot be designed in isolation of the environment in which it is intended to operate. With the emergence of easily-accessible, world-wide networks, the environment of the database system has changed, posing new requirements on the construction of the system. Those new requirements constitute the **scalability challenge**, in which the database systems

are increasingly confronted with highly variable workloads. The variability of workload manifests itself in many dimensions: the fluctuations in the client population, variations in the quality of network connections, and differences in the complexity of the queries posed. Those variations can be accommodated by the database system by server-side techniques, such as distribution and parallelization of database activities. In this thesis, we argue for another technique: distribution of query optimization components to the client side, which we deem to be more flexible than server-side-only solutions. We propose to tackle the scalability challenge by dynamically adapting the amount of distributed components to the changing workload parameters, according to the **component allocation model**. This model is the second main contribution of this thesis. As with the strategy, we validate the allocation model through measurements on the implemented optimizer: the **performance model and the measurements** form the third contribution of our thesis. All three contributions of this thesis are discussed in more detail in Section 3 of this introduction; we begin, however, by introducing the research areas which form the foundations of our approach.

1. PRELIMINARIES

1.1 Query optimization

Query optimization [Jar84] refers to the subactivity of query processing, concentrating on transformation of a query to a form allowing for efficient execution. Query processing often involves translation of the query into an internal representation and transformation of that representation into an efficient form. In the relational model [Cod70], the query expressed in a user-level language such as SQL, is translated into an expression containing operators of the relational algebra. It is then optimized by application of **logical transformations**, which are based on heuristic assumptions of performance of alternative algebra expressions. The such-transformed algebra expression can then be translated into a sequence of basic file access operators: the query execution plan. The translation can make use of multiple alternative implementations of algebra operators with differing performance characteristics, leading to the concept of **physical optimization**. The choice of alternative implementations can be based on the calculation and comparison of the cost of the alternatives, using a cost model**.**

The basic optimization strategy in relational databases has evolved as a result of the introduction of new data models, such as the object-oriented data model [Catt94]. The evolution concerned addition of new algebraic operators and transformations [Leu93, Van93, Ste95], to account for increased capabilities of query languages. It has also led to the concept of **optimizer generators** [Gra87, Gra87, Gra93b], which allow to create a query optimizer tailored to specific set of operators and transformations. The addition of operators and transformations often requires modification of the **optimization strategy**, which determines the order of application of transformations. In this thesis, we propose and evaluate such a strategy, geared towards planned introduction of algebra operators in place of nested queries.

1.2 Scalability

When operating in distributed environment, database systems have to take into account the problems of scale arising in distributed systems [Neu94]. Scalability problems may manifest themselves in the variations of node population, distances and quality of connections between nodes, and the number of organizations exerting control over the nodes. Those problems can be solved in distributed systems by allocating resources in such a way as to adapt the system to its varying environment. Resource allocation can be guided in a centralized manner, or can be determined by market-like interactions of autonomous agents [Agor95, Fer88, Fer95, Hube88]. Resource allocation techniques include replication, distribution and caching of resources: all of those techniques have also been used in the context of database systems [ÖzVa91]. Specific solutions involve distribution of data among multiple server hosts, and corresponding distributed execution of the query to facilitate data placement. Increased workload demands in database systems have been approached by improvements to the server-side architecture of the database system: making use of parallel processing and specialized hardware [Aper92, WiAp91, WiFA95].

1.3 Distributed computation

Besides distribution of data, techniques have been proposed to distribute computation entities: modules, processes, and programs. Proposals included systems supporting compiled versions of programs for all available platforms [SmHu96] and high-level languages which capture the notion of computing engines [Car95], which can be made available to remote programs. In

some systems solutions have been proposed to transfer executing processes together with their state [CDK94]. Recent solutions use the notion of platform-independent bytecode to cope with heterogeneity problems in world-wide networks. Some of them design a network operating system providing uniform means of access to local and remote resources [Infer97], while others support only the notion of a remote virtual machine that can import bytecode [Gos94]. In this way, distributed computing platforms have increased their scope from closed proprietary solutions to global networks, encompassing not only computing hosts, but also devices such as set-top boxes and hand-held organizers.

2. SCOPE

The scope of this thesis is optimization of nested queries and distribution of optimization components to improve scalability.

We base the first part of the thesis on the principle of algebraic optimization of queries. We concentrate on one aspect of that optimization: introduction of join operators in place of nested subqueries. This is achieved by applying transformations to the query, which replace nested subquery occurrences with invocations of different join kinds. To allow joins to be introduced, predicate expressions in the query have to be split to isolate subexpressions which are suitable as join predicates. We gather our transformation set from examples known in literature [Bry89, Jar84, Ste94a, Ste94b, Ste95]. However, the set of transformations alone is insufficient for construction of the optimizer. We also need the strategy for application of the transformations, which determines how the rules are structured, and at what stage in the optimization process transformations have to be applied. This is motivated by two reasons: the fact that multiple transformations are possible at a given moment, and the need to transform (split) parameter expressions before joins can be introduced. We propose an **unnesting strategy** which addresses those two requirements. Our strategy bases the ordering of transformations on performance heuristics, not on the cost model. It groups the transformations in rule sets, and provides the algorithm for invocation of the rule sets. It determines the choice of predicate expressions to split based on the possibility of subsequent join introduction, and avoids unnecessary splitting of disjunctive predicate expressions. The strategy is a result of experiments

using the implemented optimizer with test queries, containing different forms of nested subqueries and variable dependencies.

The second aspect of our research concerns the distribution of optimizer components as a means of improving scalability. We restrict ourselves to component sequences (of which a query optimizer is a good example) and their partitioning between the client and the server. We provide a component allocation model, which allows to flexibly adapt the partition of components to the workload parameters. We provide two forms of the model, suitable for different operating environments. In **centralized allocation**, the component placement decisions are based on the results of an **analytical performance model**, which gathers parameters of the system and the workload, and uses analytical formulas to determine the metric (a single performance criterion, such as the time to deploy components and optimize a batch of queries). We assess the accuracy of the model by performing **measurements** in the intranet setting, and comparing it with the results of the analytical model.

We propose to use **distributed allocation** in situations where the clients and the server have conflicting interests, which make it impossible to establish a single performance criterion. In this case, the placement decisions are based on the principles of **computational economy** [Agor95, Fer88, Fer95, Hube88], in which the clients and the server reach a satisfactory allocation by market-like interactions. The economic approach has been already used in the database context for allocation of query fragments to autonomous server hosts [Ston96]: we draw from this and other research results [Agor95, Fer88, Fer95] to build the principles of the economy and auction-based exchange mechanisms.

3. CONTRIBUTIONS

The major contributions of this thesis are the unnesting strategy, the analytical model of component allocation and its comparison to measurements, and the distributed allocation model.

3.1 Unnesting strategy

Our unnesting strategy encompasses the structuring of join transformations, and the splitting strategy for conjunctive and disjunctive predicate expressions. The contribution of join **transformation ordering** lies in the fact that it is based not only on the heuristic comparison of

operators (different forms of joins and products), but also on the comparison of expressions resulting from alternative transformations. The identification of alternative transformations, and their subsequent ordering, allowed us to group transformations in rule sets, which can be applied in one traversal of the expression and in which no alternatives exist.

The contribution of the **splitting strategy** lies primarily in its approach to processing of both conjunctive and disjunctive predicate expressions. For conjunctive predicate expressions, we contribute a “greedy” approach to splitting choices, in which we guide the splitting decisions by the next possibility of join introduction. The contribution of the strategy for disjunctive predicate expressions lies in the fact that it helps avoiding unions and multiple instances of base tables in the resulting expression.

The structuring of transformations, and the splitting strategies have been integrated in the unnesting algorithm, which has been implemented in our Joquer query optimizer prototype. The unnesting algorithm manages to remove all base table references from the nested subqueries and replace many of the nested subqueries with join invocations. The implemented optimizer is one of the few existing prototypes of complex query optimization architectures. The Joquer implementation allowed us to improve the strategy, and contribute to the software engineering aspects of query optimizer construction.

3.2 Analytical model and measurements

The analytical model of component allocation expresses the time to deploy and optimize a batch of queries as a function of system and workload parameters: the size and processing power of the client population, network transfer speed, characteristics of the query batch. To our knowledge, it is the first such model which is applied to the problem of client-side distribution of component sequences. The model and its analysis contributes the insight in the activities involved in the client-side distribution of components: we express this insight in the form of intuitive activity diagrams, which aid in understanding the relative contribution of system activities (marshalling, processing and transfer) to overall system performance. The measurements, and their comparison to the model, form another important contribution. Similar measurements [Hag86] have been performed only in the context of front-end/back-end partitioning of the query processing activities, and have not been accompanied by a performance model. Our measurements extend the existing ones by considering client-side partitioning, for differ-

ent amounts of clients. The results of the measurements show the benefits of client-side distribution of components for high levels of system parameters (sizes of query batches and client populations). The results allowed us to understand the relative importance of different activities in query processing, and identify the activities amenable to improvements. The comparison of the measurement results to the model allowed us to improve on the first versions of the model, and eventually provided reasonable consistency between the two: a consistency which allowed us to extrapolate the model for non-measured parameter values, and which eventually may lead to adoption of such models as means of determining component placement in a centralized allocation regime.

3.3 Distributed allocation model

The third contribution of our thesis is the distributed allocation model based on economic principles. We deem the economic models to be more suitable in the context of the Internet, where determination of a single criterion of allocation choice can be difficult. Although economic models have already been applied in the context of query processing [Ston96], those efforts were restricted to the server side, and did not consider the possibility to delegate those activities to the clients. This difference led to the novel choice of the principles of the economy: its product, and its pricing and exchange mechanisms. Although the proposed exchange mechanism draws from auction systems known in the literature, we contribute the integration of analytical performance modeling in our economy: we use an analytical performance model not to determine globally optimal allocations as in the centralized model, but **locally preferable allocations** in each of the market agents. The model-based preference determination is not the only way in which market agents can express their local goals: they can use heuristics to provide a simple expression of preference.

4. OUTLINE

Chapter II provides the review of current research in the three areas which form the foundation of our research. We first review the evolution of data models, query languages and algebras used in query optimization, and then describe different approaches to construction of optimizers, also those based on the notion of optimizer generators. In the second part of the chapter we

consider the approaches to query processing in current client-server systems. We also turn our attention to known models of allocating resources in distributed computing systems; their division into hierarchical and economic models leads to our dual approach to component allocation.

Chapter III is where our approach to optimization of nested queries is presented. It is here that we gather the transformation rules used in the optimizer, provide their structuring, and define the unnesting and splitting strategy. We evaluate the strategy by comparing it with existing approaches, and through experimentation with example queries. In Appendix A, we present the differences of our rule set to that of its immediate predecessor in [Ste95]. In Appendix B we present the full listing of transformations used in the implemented optimizer, and in Appendix C we provide the commented listings of optimizer processing of our example queries: the listings motivate strategy decisions taken in our approach by illustrating the differences in alternative strategies.

In Chapter IV we turn to the scalability problems, and propose component allocation as a way of tackling it. We begin by analyzing the requirements arising from the new environments of database systems: the intranet and the Internet. We then propose two allocation models which can be applied in those environments: the centralized allocation model and the distributed allocation model. We briefly describe the principles of the centralized model (as it is the topic of the whole Chapter VI) and present the details of the distributed allocation model: the product and the pricing mechanisms of the economy, and the auction system used to exchange value.

In Chapter V, we outline the implementation architecture of the Joquer query optimizer. We present the object-oriented representation of strategy elements, and the implementation techniques used to facilitate the distribution of components and transport of query optimization results. We also provide a review of existing distributed computing platforms in terms of their suitability for the implementation of our optimizer.

Chapter VI focuses on the performance evaluation of component allocation choices, which is the basis of centralized allocation. In the analysis, we first develop an analytical model of component allocation, and then perform measurements in an intranet setting. We analyze first the general results of the measurements, and then compare them with the results of the model. We then engage in extrapolation of the model for non-measured parameter values. The analytical model assumes that components are initially located on the database server: in Appendix D we

present the general version of the model, in which the components are stored on a server distinct from the database server. Appendix E contains a number of extrapolations which have not been included in the chapter itself for reasons of space.

In Chapter VII we finally summarize the contributions of the thesis and present a number of proposals for future research.

II. Query optimization and resource allocation in client-server database systems

Our approach to scalable query optimization of nested queries builds upon results in two research areas: the area of query optimization in database systems, and the area of resource allocation in distributed systems. In this chapter, we present the evolution of those two areas, leading to the components of our architecture: the query optimization strategy and the component allocation model.

We begin with introduction of query optimization concepts, followed by the description of optimization approaches used in well-known relational database systems. We proceed by considering extensible optimization systems, in which the set of optimization rules can be tailored to the data model or the query language. Another dimension of extensibility is the possibility to influence the optimization strategy, for example by modularisation. We consider also frameworks in which optimization rules are specified in a formal manner, which allows increased validations and verifications of the rules and the strategy. This forms the introduction to the optimization strategy presented in Chapter III.

Our component allocation model is motivated by the need to solve scalability problems in distributed systems, which have long been recognized in distributed systems' research. In the realm of resource management, these problems were tackled by allocating resources to processes, as to achieve better performance of the distributed system. In the second part of this chapter, we outline the current resource management research: both using the hierarchic and autonomic models of resource allocation. Our component allocation model builds on the current results and is presented in Chapter IV.

1. OPTIMIZATION OF DATABASE QUERY LANGUAGES

One of the ways used by the users to retrieve data stored in a database system involves submitting a **query (**a description of the desired data) to the database system. The query is often formulated in an easy-to-use, declarative manner using a set-oriented **query language**, of which SQL [Mel93] is the best-known example. During **query processing**, the result of the query is retrieved by the database system, which uses basic file access operations (also called **physical operators**), to obtain the needed data. **Query optimization** forms a part of query processing: it concerns the translation of the declarative query into a corresponding sequence of file access operations. The query language and the file access operators form the input and the output of the query optimizer: the transformation from one to the other is usually achieved in a sequence of phases. This top-down approach to query processing, introduced in [Jar84], is illustrated in the following figure, adapted from [Ste95].

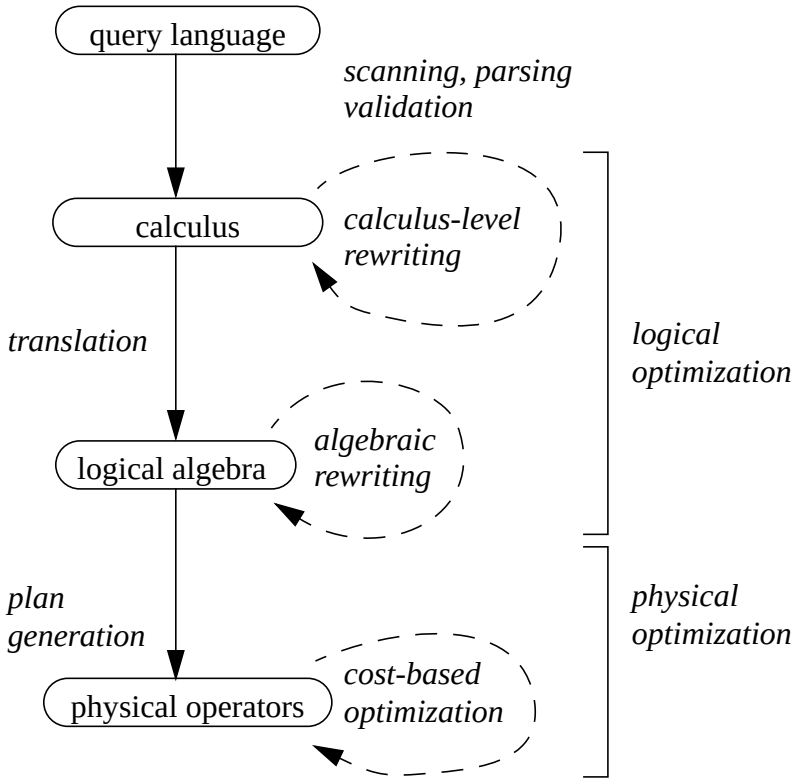

Figure 2-1 Processing of declarative queries

In top-down approach to query optimization, the declarative query is first translated into an internal representation, containing calculus-like elements such as quantifiers. This representa-

tion is then subjected to the phase of **logical optimization**, consisting of transformations within the calculus representation, translation to logical algebra, and transformations within the algebra representation. Within each of those steps, the transformations can be expressed in the form of **logical rewrite rules**, which describe on the high level the substitution of a subexpression in the input expression by another subexpression. As such, logical rewrite rules can describe transformations from one level to another, as well as transformations within one level. An example of a logical algebra-level rewrite rule is one which combines a sequence of projections into a single projection. Rewrite rules can be interpreted during the optimization of the query as in Starburst [Haa89, Loh88, Pir92], or they can be compiled by an optimizer generator into an optimizer as in EXODUS and Volcano [Gra86, Gra87, Gra93b].

The transformations on the calculus level involve standardization of predicate expressions by their translation to conjunctive or disjunctive prenex normal forms. During the translation step, the query is rewritten by replacing calculus-like expressions of the query language with **logical algebra operators**, such as projection, selection, and join. The introduction of logical operators allows for application of algebraic rewrite rules, which replace less efficient algebra expressions with more efficient ones. Logical rewrite rules are independent of the database content: they are based on heuristic considerations of the performance of alternative (algebra) expressions. Another benefit of the introduction of logical operators lies in the fact that for a given operator, the database system can choose from several possible implementation methods during the following steps of the query optimization. The order of applications of logical rewrite rules to a given query is guided by the **optimization strategy**.

In the third step of query optimization outlined in [Jar84], the algebra expression is mapped into sequences of physical operators whose costs are known. As there may be multiple possible implementations for a given logical algebra operator, the plan generation results in a set of alternative physical operator sequences. Also this mapping can be described using rewrite rules as in [Gra93b].

In the final step, the best physical operator sequence is chosen through estimation of the measure of **cost** (such as query execution time): the estimation is based on the implementation details of the physical operators and statistical information about the content of the database. The term **physical optimization** is used in this thesis to denote the last two steps.

The above framework (based on [Jar84, Gra93a]) describes the general approach to query optimization. The query optimizer research has concentrated itself on several subareas of the framework. Some proposals involve design of query languages for new data models, provision of new logical operator algebras together with corresponding transformation rules, and design of efficient implementation methods for algebra operators. Others concentrate on the architectural issues, allowing for automatic generation of query optimizers for given physical and logical operator sets. Recent proposals concentrate on the formal representation of operators, transformation rules and search strategies. In the following sections, we present a review of representative proposals and systems in the research subareas described above.

1.1 Data models, query languages, and algebras

The data model defines the type system used during modeling of database schemas. As such, it determines the characteristics of the query language, which operates on data structured according to the database schema. The prominent data model used in database applications is the relational model [Cod70], in which data is structured into relations: sets of tuples of the same structure. A tuple is a set of labeled primitive values. The data, structured according to the relational model, can be queried using a sequence of **relational algebra** operators, or by specifying a non-procedural **relational calculus** expression (which in turn is based on predicate calculus). Within the calculus, predicates are formulated using variables ranging over tuple or domain values. The declarative character of the relational calculus led to its adoption as a basis for user-level query languages such as QUEL [Ston76] and SQL [Mel93]. The calculus expression is translated into an algebra expression, and optimized using algebraic transformations.

Within the relational model, tuple attribute values can only have primitive types. The removal of this restriction led to the introduction of nested relational (NF^2) data models [ScSc86], in which relation-valued attribute types are allowed. This extension led to corresponding improvements in the query language and the algebra to allow for retrieval of data within the nested data structure. In the DASDBS algebra [ScSc86], operators nest (ν) and unnest (μ) have been added to the relational algebra operators to allow for access to nested structures. The introduced operators possess the important property that while μ is always the inverse of ν, it is not always the case that $\nu(\mu(R))=R$. The equality is achieved if the relation is in Partitioned Normal Form: its atomic attributes form a key to the relation, and all nested relations are also

in Partitioned Normal Form. Partitioned Normal Form imposes certain restrictions on the modeling freedom: to allow for arbitrary nested relations to be unnested, keying operation has been added in the AIM algebra [Dada86]. The keying operator appends a key column to the relation to enforce the Partitioned Normal Form, and ensure that nesting after an unnest will result in the original relation. Another approach (in SQL/NF algebra of [RoKS88]) to the Partitioned Normal Form restriction involves extension of standard relational operators as to make the algebra closed under the Partitioned Normal Form. The operators are defined recursively to allow for their application to nested relations. Similar approach has been followed in the VERSO database machine [Abit86]; besides nested relations, operators have formats (schemas) as operands. This allows projections to be applied at arbitrary nesting levels. Additionally, VERSO defined a *restruct* operator which allows arbitrary restructuring of nested relations, subsuming nest and unnest operators. In the algebra of [Colb90], application of operators on nested levels is achieved by allowing path expressions as operands.

The evolution of NF^2 data models led to introduction of other type constructors (tuples, multisets and lists) beside relations, in extended NF^2 models (XNF^2) [Dada86, Leu93]. Additional operators are then added to support new type constructs, and convert between different constructed types. This leads often to large number of operators, and consequently large numbers of algebraic transformations.

The query specified in SQL (or its variation for an NF^2 model) can contain nested subqueries in its **SELECT**, **FROM** or **WHERE** clause, possibly connected using aggregate expressions such as **AVERAGE** and **SUM**. Such queries have been classified into different classes depending on the location of nesting, and the connection between nested query blocks [Kim82]. In some situations, those nested queries can be translated into join operators [Kim82, GaWo87] or expressions including specialized grouping operators [Clu92, Clu94, Ste95]. To function properly, those translations require a suitable **unnesting strategy**, which allows for introduction of beneficial join types and appropriate transformation of predicate expressions to allow for join introduction. Such a strategy can be positioned in the translation phase of Figure 2-1, and is one of the contributions (Chapter III) of this thesis. In predicate transformations, of special interest is the handling of disjunctive predicates, which may lead to introduction of unions and duplicate instances of base tables in result expressions. One of the proposals [KMPS94] introduces two-stream (bypass) selection operator, which results in two collections being generated,

containing the tuples for which the selection predicate evaluates to true and false, respectively. In this case, union in the result expression can be replaced by a merge (union without duplicate elimination) of bypass selection results. In [KMPS95], bypass versions of semijoin and join have been proposed. In our strategy, we do not use bypass operators, but attempt to avoid unneeded unions by timely introduction of Cartesian products.

Following the success of programming languages such as C++ and Smalltalk, attempts have been made to introduce object-oriented modeling concepts, such as objects, classes, methods, subtyping and inheritance, in database systems. This has led to the introduction of object data models in object-oriented database management systems [Catt94, BaCD92, Zdo90], which are well-suited to new application domains such as CAD and GIS systems. Also in this area proposals have been made for query languages and algebras [Leu93, Van93]. Within those systems, optimization research has concentrated on retrieval of data specified by path expressions (attribute navigation chains such as `p.spouse.address.city`). Possible approaches include application of specialized indexes [BeKi89, KeMo90] to speed up access, and physical placement of objects to facilitate faster access (clustering). Proposals have also been presented for processing of queries containing methods. If methods are expressed in the query language itself [RoSt87, Ston86, Ston90], the query can be expanded by substituting the method expressions within the query and optimized in a standard manner. If methods are not expressed in the query language, information about the costs of the method can be revealed [Dan91] to the optimizer to allow for its (cost-based) optimization. In our approach, we do not focus on method optimization.

In the area of query optimization, we observe the increasing variety of data models, query languages and algebras. To be applicable in practice, however, those solutions have to be embedded within suitable implementation architectures for query optimizers. In optimizer research, we observe therefore the evolution of **optimizer architectures**, allowing them to be more flexible and extensible, to accommodate the evolution in data models, query languages, and algebras.

1.2 Optimizer architectures

We describe the evolution of optimizer architectures using a number of optimizer prototypes known in literature. For each of the proposals, we describe its place within the framework of

Figure 2-1. We also describe in a cursory way their relationship to our architecture: more complete evaluation is presented after the presentation of the architecture in Chapter V.
We begin with the System R query optimizer, which illustrates the basic architecture for a non-extensible query optimizer.

1.2.1 Traditional query optimizer: System R

System R [Astra76] is a relational database prototype developed at IBM San Jose Research Lab, containing an optimizer for the query language SEQUEL. The optimization process in System R follows the general framework on Figure 2-1.
The input query is parsed into component parts: lists of **`SELECT`** and **`FROM`** expressions and the predicate tree present in the **`WHERE`** clause. During logical optimization, heuristic equivalences are used to simplify the input query. The main heuristic rule is the one prescribing to apply selections and projections before join applications; the rule is motivated by the fact that join operands will typically be smaller if projections and selections will be applied first. Our architecture shares System R's heuristic approach to the logical optimization phase. Within the physical optimization phase, the catalog (data dictionary) is consulted to obtain statistics about the relations and access paths available. The System R optimizer assigns a selectivity factor to each boolean factor in the **`WHERE`** tree. The factor corresponds to the expected fraction of tuples which will satisfy the predicate; it is established using schema information from the data dictionary and formulas for selectivity factors for various forms of predicates. Next, the best way of joining any relation to any relation is found, and so forth for n-way joins. The System R optimizer starts therefore from the participating relations and build the search tree upwards, pruning the "expensive" paths underway. Access paths with minimal costs are chosen and the resulting plans are specified in the Access Specification Language (ASL). During code generation, ASLs are translated into executable code and associated data structures. The query is then executed, using System R internal Relational Storage System (RSS) calls.
In System R and other query optimizers in relational database systems, the optimization architecture was not designed to cope with changes in the data model, query language, and the algebra. Those changes manifest themselves in **extensibility dimensions**, which influence different parts of the query optimization framework of Figure 2-1:

- Additions to the set of algebraic operators to the logical optimization phase.

- Additions of new rewrite rules, expressed in a high-level manner. New rules can be added to calculus and logical algebra levels: they either may involve rewriting from one level to another (e.g. replacing quantifiers with joins), or transformations within one level (e.g. changes in the order of join evaluation).
- On the physical optimization level, additions of new physical access operators with accompanying cost estimations.
- If both physical and logical optimization phases operate in a rule-based manner, integration of logical and physical optimization within one framework can be realized. In this case, the logical and physical optimization phases merge into one phase.
- Automatic generation of optimizers for varying data models, operator and rule sets. In this case, the data model, operator set and set of rewrite rules become parameters for an optimizer generator, which compiles the parameters, together with a predefined optimization strategy, into a query optimizer.
- Besides changes to the operator and rule set, the optimization strategy itself can be modified: such changes apply to rule structuring, the order of application of rule sets, and the traversal strategy of the input expression.

We now present the research prototypes representing the above extensibility dimensions.

1.2.2 Rule-extensible optimizer: Starburst

An example of an extensible rule-based query optimizer is the Starburst query optimizer [Haa89, Loh88, Pir92]. During query optimization, the query is represented as a Query Graph Model (QGM). The QGM is first rewritten during the Query Rewrite phase (corresponding to logical optimization phase on Figure 2-1). During Query Rewrite, the query is transformed according to logical heuristic rules similar to those applied in System R. The rules can be structured in rule sets, each of which can have different scheme for choosing the next rewrite rule to apply. Our architecture builds on this approach to rule structuring: we additionally allow rule reuse in rule sets and add different query traversal options to the extensibility dimensions. Our focus is also somewhat different: while Starburst considers rewrite rules on the level of query language (calculus-like), we are interested on the phase of translation of calculus expressions to the algebra.

After Query Rewrite, alternative query plans are created, their costs are estimated and the least-cost execution plan is chosen. This corresponds to the physical optimization phase on Figure 2-1. Within the Plan Refinement phase, declarative rules (Strategy Alternative Rules: STARs) describe how to build higher-level constructs from primitive operations, e.g. a table scan operation can be realized by using a heap, B-tree or R-tree. Each STAR defines an object in terms of one or more alternative definitions (triggered by applicability conditions), which, in turn, may be defined in terms of other STARs or primitive physical operators (LOLEPOPs). An execution plan in Starburst is a nesting of invocations of LOLEPOPs.
Specification of transformation rules (both on the logical and physical level) in a declarative way is a step towards extensibility; the speed of the resulting optimizer suffers, however, from the fact that the rules are interpreted and not compiled. An alternative is the idea of optimizer generators, explored in two research prototypes: EXODUS and Volcano.

1.2.3 Optimizer generators: EXODUS & Volcano.

In EXODUS [Gra86, Gra87], the optimizer implementer specifies the transformation rules in a declarative manner: they are however not interpreted when the query is submitted, but compiled beforehand by the optimizer generator into an optimizer. In this way, EXODUS attempts to separate the optimizer logic (the search strategy) from the query model specifics (the algebra, the set of implementation methods, transformation and implementation rules), which is specified in the **model description file**. The optimizer logic is supplied by the optimizer generator, while the model description file forms the input of the optimizer generator. The model description file contains not only the transformation rules, but a description of the available operators as well, which allows generation of optimizer for diverse operator sets. Besides the transformation rules, the model description contains also implementation rules, which describe the mapping between operators and algorithms implementing them. The optimizer generator principle allows therefore to modify aspects of both the logical and the physical optimization of Figure 2-1: by adding algebraic rewrite rules and by modifying plan generation and cost-based rewriting phases.
Using the EXODUS optimizer generator, the database implementer (DBI) must provide a model description file and a set of C procedures. The file forms the input to the optimizer generator, which produces an executable optimizer (adding the predefined optimizer strategy).

During the actual optimization, after determining that the transformation rule can be applied to a given subquery, the system applies the rule, producing new query (sub-) trees. For each new node in the query tree, an implementation method is selected, based on the comparison of costs with implementations generated so far.

The selection of transformation rules is guided through the **promise** of largest cost improvement, which is calculated based on the expected cost factors. The cost factors are determined by the optimizer automatically from past experience (e.g. through various averaging of observed cost factors). The searching process is guided by two other factors:

- hill climbing factor, which is the factor of improvement expected by applying a transformation to the best one observed. The hill climbing factor is used to determine whether the optimizer will search for another solution.
- reanalyzing factor, which determines whether the parent nodes of a subquery have to have their implementation rules recalculated.

Both factors can be used to restrict the amount of alternatives considered during the optimization process.

The successor of EXODUS optimizer generator was the VOLCANO optimizer generator [Gra93b]. Similarly to EXODUS, the input of the optimizer generator is the model description, which is translated into optimizer source code. Volcano generates optimizers which map logical algebra expressions into physical algebra expressions (query evaluation plans). The mapping is expressed in transformations within the logical algebra and cost-based mappings of logical to physical operators (algorithms).

The optimization process in Volcano is guided by the notion of logical and physical properties of transformation results. An optimization goal is a pair of a logical and physical property vector. An applicability function is used to determine whether an algorithm can deliver the logical expression with physical properties that satisfy the physical property vector. The cost of the algorithm is estimated using the cost function modeled as an abstract data type.

The search algorithm employed in Volcano takes as input the logical expression, the desired property vector, sort order, and a cost limit. It manages a hash table containing all plans generated so far. In case when no suitable plan can be found, three alternative “moves” are explored: using a transformation rule to another logical expression, using implementation rule to a physical algorithm algorithm or using an enforcer of a physical property such as the sort order. The

moves are ordered, and the most promising moves are pursued. In contrast to EXODUS, the strategy can pursue any of the three moves at any given moment, which allows influencing the search strategy with data model-specific considerations. Volcano optimizer integrates therefore the logical and the physical optimization phases of Figure 2-1 using the same search strategy.

Our architecture for query optimization differs considerably from that of both EXODUS and Volcano. We do not propose an optimizer generator, but an optimizer for a specific data model. We focus also on the "upper" phases of Figure 2-1: calculus-level rewrite rules and translation from the calculus to the algebra, while both prototypes concentrate on logical and physical algebra levels (and their integration).

Optimizer generators enabled implementers to realize optimizers by specifying declaratively the target operators and transformation rules. In some cases, however, the flexibility needed extended also to the influence on the optimization process itself, the strategy in which the rules were applied, as well as the sequence of steps leading to the optimized query.

1.2.4 Strategy-extensible optimizers

The first attempts to influence the optimizer strategy involved suitable software engineering architectures, in which strategy elements could be represented using object-oriented programming concepts. In this way, the strategy can be influenced by the mechanisms of class inheritance and method redefinition.

In the frameworks by Lanzelotte & Valduriez [Lan91a, Lan91b, Lan92] as well as in OPT++ [KaDeW94], the optimization architecture elements are modeled in a hierarchy of classes describing concepts such as search space, search state and search strategy. Each of the elements can then be modified by addition of new classes and redefinition of methods. These extensibility methods describe the characteristics of the optimizer architecture: its initial search space, stopping condition and pruning actions. In this way, both cost-based and heuristic optimization can be described by redefinition of methods. Both approaches are general in the sense that they can be applied to different optimization phases on Figure 2-1; in our architecture, we also apply object-oriented methods to modeling of optimizer elements.

Another feature of the query optimization process promoting extensibility is its modularisation [ScSi90], which allows division of the optimization process into stages ("experts") with different search strategies, rule sets, and goals. In this manner, the strategy can be influenced by add-

ing new experts and changing existing ones. The query is processed on an “assembly line” of experts, each of which transforms the query according to its local goals. Each of the “experts” defines the search strategy, the rule set, cost analysis technique (if applicable), termination condition, and the conditions for the resultant expression to be included in the output of the module. Rule sets are further characterized by properties such as existence of a single normal form (Church-Rosser property). The characteristics of the experts are specified in a module description file, which can be used by the optimizer generator to create an optimizer.

Besides characterization of modules themselves, a query optimization architecture requires strategy for application of modules. Mitchell [Mit93] has extended the approach to other than linear control structures in the EPOQ architecture. The approach allows therefore to model the top-down approach of Figure 2-1 and realize complex control schemes within each of the phases. This is realized through the modular architecture, in which the optimizer is modeled as an (extensible) hierarchy of **regions**, each of which defines a strategy for transforming queries aiming at specific goals. Regions are characterized by:

- a predicate describing the set of possible input queries
- a collection of transformations and possible subordinate regions (children)
- a means of control over the application of those transformations
- a goal predicate that characterizes the output queries

During optimization, the parent region communicates with its child regions to determine whether they can be applied to a given query, and whether they can achieve the goals of the region within specified conditions. The goals of the child regions can be used in the parent’s decision-making, which is expressed in the parent’s **control**. The region control chooses transformations (region or rule invocations) to apply on the input query, needed to achieve the region’s goal. The region maintains an internal store, which allows it to save and evaluate different transformation alternatives (for example based on their cost). It also maintains a log of executed transformations, which allows to back up if transformations do not lead to region’s goal. The region control allows for specification of different termination conditions: involving the characteristics of the transformed query or the usage of resources.

Our optimizer architecture, although not using high-level modeling of optimizer control such as in EPOQ, uses a similar representation of query expressions.

The evolution of strategy-extensible optimizers proceeded from software engineering frameworks, easing addition of new optimizer phases and strategies, to architectures supporting high-level description of optimizer modules, and planning-based control of module invocations. The ability to represent strategies on a high level led to the concept of formal representation of transformations and optimization strategies, which are especially suitable in the initial stages of the optimization process.

1.2.5 Formal representation of transformations

The extensible approaches described above led to a degree of high-level formulation of optimization rules. The suitability of such rules for algebra-based optimization cannot be underestimated. However, in many cases the input query, expressed in a calculus-like representation, will require complex transformations even before it has been fully transformed into an algebra expression. Such calculus-level transformations (see Figure 2-1) often have to be supplemented with complex applicability conditions involving variable existence and usage checks. Efforts have been made to develop variable-free representations which obviate the need to use code fragments in applicability conditions. Such representations, if based on a formal basis, can be subjected to formal validations and verifications, which contribute to the quality of the optimizer.

One of the possible formal representations for calculus-level queries is the monoid comprehensions calculus of [FaMa95]. Monoid comprehension calculus allows for formal representation of different query-level elements such as collection types and quantifiers. The calculus expression can then be transformed into a canonical form using a set of transformation rules. This approach has been followed in CROQUE [Gru97], where OQL queries are translated into a mix of monoid calculus and algebra, and the resulting expressions are transformed using heuristic rewrite rules. In this approach, both calculus and algebraic operators are represented formally in the same (monoid comprehension) framework. This allows for calculus-calculus and calculus-algebra transformations to be expressed in the same manner. It therefore represents the integration of calculus- and algebra-level rewriting phases in Figure 2-1. Algebraic elements are introduced in the query where possible, with the remaining calculus-level elements being represented by invocations of *fold* operator (which applies a function to elements of collection and returns a collection of the function results). Our strategy is focused on the same

phase of the optimization process: we do not, however, use a formal framework for its representation.

Calculus-level transformations are inherently difficult to implement because of the variable interdependencies often encountered in high-level query languages. In the KOLA representation [Che96] transformations are represented in a variable-free manner using a combinator algebra. The KOLA combinator algebra consists of primitive functions and predicates, and **function formers**: functionals defining new functions in terms of existing ones. Both calculus and algebra-level expressions are expressed using combinators (applications of formers and primitive functions). Besides lack of variables, representation of transformations as KOLA rules has also the benefit of verification, as the transformations can be (semi-automatically) proven.

1.2.6 Formal representations of optimization strategy

The formal approach to the specification of transformation rules has been extended to the representation of the optimization strategy in the COKO extensions of the KOLA framework [Che97]. COKO is a language for specifying complex query transformations, which in turn consist of rewrite rules (expressed in KOLA) and a strategy for their application (firing algorithm). As KOLA rules can be proven correct, so can be COKO transformations. The COKO transformations are translated into C++ routines by a compiler. The language for firing algorithms allows for different forms of COKO statements:

- KOLA rule firings (also inversely)
- Statement execution on specified subtrees of the expression, identified by patterns.
- Iteration statements allowing for top-down and bottom-up traversal of the query tree.
- Complex statements: sequential (statement *S* before *S'*), alternative (*S'* if *S* not succeeded) and consequential (*S'* if *S* not succeeded).

COKO statements have been used to represent the unnesting transformations of [Kim82]. COKO can be seen as an attempt to formalize optimizer control schemes such as those characterized in EPOQ. The COKO representation has the potential of expressing complex optimizer strategies such as the one which is presented in this thesis. This would allow the strategies to be validated on a high level: in our approach, the strategy was designed and validated by tedious experimentation with diverse query forms.

In this section, we have reviewed the current research in the area of query processing in nested object models. We have also considered the available frameworks for extensible representation of transformation rules and optimization strategies. We have placed the current results in the context of general framework for query optimization and briefly indicated their relationship to our results.

The other aspect of our research is the distribution of query optimizer components. To place our results in context, we review the research results both in the area of query processing in client-server database systems, as well as in the area of distributed systems, with special attention to distributed computing systems.

2. DISTRIBUTED QUERY PROCESSING

Query optimization architectures are often incorporated within client-server database systems. We first review the client-server aspects of query optimization, and then observe the techniques used in distributed systems to cope with scalability problems.

2.1 Query processing in client-server systems

In [Ber96], client-server approaches to database integration (including query processing) have been categorized into two groups:

- In client-server SQL integration, the clients send SQL requests to the server (either via Remote Procedure Calls or through messaging) by invoking functions of a software library. Those library functions have been standardized in the form of SQL Call Level Interface, which is a part of the SQL standard [Mel93].
- Distributed TP managers present a consistent application-to-application (thus also database client to database server) interface for multiple server and client applications in a distributed SQL-based OLTP environment. As such, they allow interoperability between multiple database server products. Well-known products include CICS, Tuxedo, TOP END and Encina.

In current client-server systems, the bulk of query processing is performed on the server. Client participation is usually restricted to the process of formulation of the query and potential elementary verification of the query (syntax checking). Query optimization is usually performed

on the server, as it (in the case of physical optimization) requires interaction with the physical statistics of the schema.

2.2 Solving scalability problems in database systems

The emergence of ubiquitous communication networks known under the name of Internet posed new challenges on database systems. The most important of those challenges is that of scalability. A scalable system has been defined in [Neu94] as one which can handle addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity. The scale manifests itself in the dynamics of the number of clients, the distance between system nodes and the number of organizations which exert administrative control. In case of the Internet, additional elements of scale are: unpredictable usage patterns and differing client capabilities. Solutions to scalability problems lie in three broad areas:

- Replication of services and resources: creation of identical instances of services and resources on multiple nodes.
- Distribution of services, where each of the nodes is capable of handling a subset of the requests for service. Distribution of computation among nodes is a form of service distribution.
- Caching of service results, reducing the transport costs.

To allow for distribution of services, the resources and services have to be allocated to nodes in a distributed system. The problem of resource allocation has generated considerable interest in the distributed systems research community [Zed90]. The problem was often considered in the context of distributed operating systems, where computing load had to be balanced over a pool of available processors (load balancing). Each of the allocations possesses a degree of **utility** to the system: utility is the criterion (preference) used to compare and choose between different allocations.

In general, there have been two approaches to the problem of resource allocation: one based on **hierarchic** view of the distributed system and providing system-wide (global) measures of utility, and the other offering **autonomous** means of allocation based on local measures of utility. Our component allocation model contains aspects of both hierarchical and autonomous models. The difference between the two approaches lies in the method in which allocation is determined.

In hierarchic models, allocation is determined by optimizing a common utility parameter (e.g. query response time). In other words, all nodes in the system accept the same notion of “better performance”, which allows the allocation to be calculated centrally using a single optimization function, parameters to which are contributed by the nodes.

In autonomous models, there is no common measure of utility: nodes may have different and conflicting notions of “better performance”. For example, a client may be interested in minimal query response time, while the server may want to maximize the number of connected clients. Each of the nodes possesses therefore a **local measure of utility**; those differing preferences have to be reconciled to reach the allocation **satisfactory** (pareto-optimal) to all parties. The similarity of that situation to real-life economies led to the emergence of **computational economies**, in which the reconciliation is achieved through the market mechanisms: products, prices, and exchange of value. For the market rules to work, each of the nodes must conform, as in real life, to the general market rules, but may impose its own measure of utility.

We present short descriptions of both models, referring to relevant research results.

2.2.1 Hierarchic models of resource allocation

Hierarchic models of resource allocation can be further divided into static and dynamic (adaptive) models. In **static models**, the system-wide cost measures (used to calculate utility of allocations) are assumed to be known *a priori* and can be used for resource allocation without information on the current state of the system. In a **graph-theoretic approach** [Sto77], distributed systems can be modeled as *graphs* of resources, producers and consumers. If this approach is applied to the load balancing problem, the nodes represent processors and modules, edges between processors and modules represent allocation of module to a processor, and edges between module nodes represent the amount of data passing between them. As edges are annotated with costs of module storage and data transfer, graph-theoretic algorithms can be used to calculate the optimal division (cutset) of the graph, such that the sum of weights of “broken” edges are minimal. The minimal cutset corresponds to the optimal module assignment.

Alternatively, costs can be calculated using **0-1 integer programming techniques** [Cho82], where allocation of resources to processors is modeled in a 0-1 matrix (in which an allocation of resource *i* to processor *j* is represented by 1 in the matrix). Using the allocation matrix, cost

of resource allocation can be calculated and optimized. That cost can be optimized subject to constraints imposed, such as the amount of available memory on a processor.

Both the graph-theoretic and the 0-1 techniques are computationally expensive. Alternatively, in **heuristic methods**, a suboptimal allocation is attempted by identifying modules with large volumes of communication between themselves, and allocating them together on the same processor.

Alternatively to static models, **adaptive** (cooperative, consensus-based) **load balancing models** attempt to take into consideration the dynamic character (current state) of the system [Zed90]. In these models, the system-independent state (load) determination becomes a central issue, leading to several estimation methods (five-minute length of run queue, intervals between CPU allocations). Additionally, state information has to be exchanged between processors in a manner reducing state obsolescence; approaches include use of local load vectors and state broadcasts. Adaptive models propose various policies for transfer of modules between processors, either through introduction of load threshold values (load too heavy - transfer is needed), or by basing the transfer on differences in loads between processors. The transfer can be either sender-initiated (from the overloaded processor) or receiver-initiated.

2.2.2 Market-based approaches

In many cases, distributed systems are constituted from heterogeneous processors, and the communication between nodes takes place over unreliable, insecure links. In this case, it may be difficult or impossible to agree on a system-wide measure of performance or cost, as was assumed in hierarchic models. In a client-server system, for example, the client may prefer component allocations which allow for faster query processing, while the server will prefer allocations which will allow it to accept as many connections as possible. Both the client and the server have different goals: different measures of utility of allocations, which are difficult to reconcile in a centralized way. This restricts the usability of hierarchic approaches and calls for models more suited to those cases.

Recent research has proposed the use of economic paradigms for resource allocation [Fer88, Fer95, Wald92, Ston96, Mill88], as more suitable for situations where agents demonstrate a level of autonomy, and where hierarchic means of control are impossible. In economic paradigms, resource allocation environment is modeled as a **market**, with autonomous agents dis-

playing possibly selfish behavior and possessing local allocation preferences. In economic models, resources are allocated according to market forces, guided by the demand/supply equilibrium, using mechanisms of pricing and negotiation. In those models, agents attempt to achieve a preferred and affordable allocation of resources; in contrast to hierarchic models, the measures of preference (utility) and affordability are local to one agent, and not system-wide. It is important to note that while the agents in a computational economy do not have to agree on the measure of utility, they have to "play according to the market rules": they must conform to the specific market rules used. This includes arrangements as to the budget allocated to agents, the notion of price and the procedures used to reach equilibrium (auction or otherwise). This can be realized by letting each of the nodes run the component containing the "market logic", which takes care of fair allocation of budget and conformance to market rules and protocols.

Economic models offer various mechanisms which lead to the demand/supply equilibrium. In **price-base models**, resources are priced according to market forces - the forces of supply and demand. The agents buy and sell resources, which leads to appropriate price adjustments, and eventually to price equilibrium. In **exchange-based** economies, the agents agree on exchange which will provide them with increased utility. A variation of price-based mechanisms are **auctioning** schemes [Agor95], which can be used to achieve the price equilibrium. Best-known auctioning schemes are English and Dutch auctions, as well as first- and second-price sealed bid schemes. In all auction schemes, the auction participants (**bidders**) receive initial budget, which they may use to purchase resources. In English auction, the auctioneer begins with lowest acceptable price and solicits higher bids from the bidders. Bidders assess whether they can afford the price within their budget, and submit the bids (the bid price increased with **markup**) if they can. The item is sold to the highest bidder. The bidders control their preference for resources by controlling the height of the markup. The English auction is an open-outcry system: when bids are placed, they are known not only to the auctioneer, but to other bidders as well. In the Dutch auction system, bidding starts at an extremely high price. The price is progressively lowered until a bidder claims the product.

Alternatively to open-outcry systems, sealed bid systems do not allow bidders to see each other's bids. In those schemes, bids are placed only once, and the auctioneer determines the allocation based on the analysis of bids. In both sealed bid schemes, the item is allocated to the

highest bidder; however, the price that the winner has to pay can be either the highest price (first-price) or the second-highest price. The second-price sealed bid auction leads to more aggressive bidding, as bidders are less afraid of paying overly high prices for acquired items.
Market-based models have been used for allocation of circuits to connections in computer networks [Fer88], job allocation to unused workstations [Wald92] and query execution in distributed databases [Ston96]. Their characteristics make them interesting candidates for distributed query optimization.
After considering the area of general allocation of resources, we turn to the problem of query processing, and examine the solutions used to solve the problems of scale arising in this activity.

2.2.3 Existing solutions to scalability problems of query processing systems

In the area of query processing, solutions to scalability problems include partitioning of query processing activities between multiple hosts. In the area of distributed databases [ÖzVa91], processing activities were divided between multiple server hosts to take into account placement of data (vertical partitioning). Analogically, query processing activities can be partitioned (horizontally) on the server side between the frontend and backend system in [Hag86]. In this system, the query processing activity has been divided into configurations: subdivisions of functions between front-end and back-end systems. Configurations vary between one in which the whole query processing is executed on the front-end system and one in which only parsing of the query is executed on the front-end system. The front-end/back-end solutions can be characterized as hierarchic, as they employ a common measure (set of measures) of utility.
The configurations have been compared in terms of performance metrics such as the CPU time, average use of memory space, disk I/O reads, and network statistics. The results of the measurements have shown that none of the configurations was found to be better than all others for all measured metrics. The choice of configuration is based on the choice of relevant metric. The measurement results have not been supplemented with an analytical performance model, and they do not consider the possibility of distribution of processing activities to the client side. In our approach, we propose an analytical model of component allocation and compare its results with the measurements.

The autonomous models have also been applied in the area of query processing in the MARIPOSA system [Ston96]. Here, economic principles have been applied to allocate query fragments to server-side hosts for processing. In MARIPOSA, it is assumed that server sites display a degree of autonomy: their decision of accepting or rejecting processing tasks can be based on local considerations. This corresponds to contemporary situations in which cooperating server sites can be placed under distinct administrative control, and as such can be subjected to local restrictions. In such a situation, it is impossible to impose allocation decisions based on a centralized notion of utility. In the MAROPISA system, economic principles have been applied: each of the server hosts is equipped with a **bidder**: a software component which uses the common market rules to improve the host's local utility measure.

Within MARIPOSA, a query submitted to the (distributed) server is accompanied by a budget, within which the system has to evaluate the query. The system solves the query within the allocated budget by letting the broker contract the processing server sites (their bidders) to execute portions of the query. The bidders respond to the broker with bids, the broker determines allocation and transfers the query fragments to the processing sites. Unwanted fragments are either auctioned repeatedly, or pre-allocated to sites which show greatest promise to win the auction. During query execution, the partial results are collected, integrated and returned to the user. To expedite the allocation of fragments, sites announce willingness to perform certain query forms by posting advertisements. Besides buying the fragments from the broker, sites can buy and sell fragments between themselves, realizing a form of subcontracting.

We compare our approach to distribution of components with MARIPOSA in Chapter IV: we only note here that our market system distributes components between a client and a server, and not between server sites. Our auctioning scheme draws from the systems known in the literature; we additionally propose to use a local analytical performance model as a measure of utility.

In this chapter, we have presented the current research results in two areas: the processing of queries in modern database systems and the allocation of resources in a distributed system. These areas form the basis of our distributed processing architecture: an architecture in which we distribute query optimizer components between the client and the server to achieve beneficial resource allocation. In the sequel, we present the elements of our architecture: the unnest-

ing strategy, the component allocation model and the implementation architecture. We first attempt to precisely define the query optimization activities in Chapter III, and then consider the possibilities of its partitioning between the server and autonomous clients (in Chapter IV), taking into account both the centralized and market-oriented resource management. We then present an implementation architecture for the query optimizer facilitating its distribution in Chapter V, and evaluate the performance of the distribution architecture in Chapter VI.

III. Optimization of nested queries

In this chapter, we turn our attention to the strategy for logical optimization of object queries, in which we transform nested declarative queries into more efficient algebraic forms. In the development of the strategy, our goal is the removal of nested base table references and replacement of calculus-level constructs such as quantifiers by more efficient algebraic operators. We provide a strategy for planned application of transformation rule sets, by introducing the proper structuring of rules into rule sets, and by providing the algorithm for application of rule sets. The strategy forms the main contribution of this chapter.

We first introduce the goals of our optimization strategy. We then introduce the query language and the algebra used in our optimizer. In Section 3, we present the strategy, consisting of unnesting and splitting components. We then evaluate the strategy by describing its operation for example nested queries, and by comparing it to optimization frameworks known in the literature. Lastly, we describe possible directions for further research and formulate our conclusions.

1. STRATEGY GOAL

The optimization framework presented in Chapter II divides the query optimization process into two phases: logical and physical optimization. The focus of our strategy is on the logical optimization phase of query processing. In this phase, we attempt to transform calculus-like query language constructs to algebraic expressions, which allow us to choose from many efficient operator implementations in the physical optimization phase. The broad goal of our strategy is therefore the replacement, to the highest degree possible, of calculus-level elements of the query with their algebraic equivalents. Our goal is not only the introduction of algebra

expressions as much as possible: we want to take into account the relative performance of alternative calculus-algebra transformations, allowing us to introduce the more beneficial algebra elements instead of the less beneficial ones. We focus on specific forms of nested queries, containing references to base tables, quantifiers, and subqueries, on nested levels of the query. Those forms, if left as they are, can often only be implemented with a nested iteration (nested-loop) algorithm. Replacement of those forms with joins (unnesting) allows to use join implementations other (and more efficient) than nested-loop processing. We therefore rewrite those elements into different forms of join operators using **rewrite rules**, applied according to the unnesting **strategy**. The rewrite rules define the beneficial transformations of the query, and the strategy defines the choice, moment and place of rule application. The formulation of the rewrite rule set and the unnesting strategy for nested queries is the main topic of this chapter. Both the rewrite set and the strategy have been implemented in the Joquer query optimizer. The final form of the strategy has been developed during experimentation with the optimizer, in which we tested various strategy options on test queries. We describe the test queries and their processing by the optimizer in Appendix C.

We present both elements of the unnesting strategy in the following sections. We begin by first introducing the context of our query language and algebra.

2. INPUT AND OUTPUT OF THE OPTIMIZER

The input language of our optimizer is the **TMql** language, which is a subset of the database TM specification language [BaBZ93]. TM is used for object-oriented modeling of database schemas containing methods: it is the subset of the method language of TM that we use as our input language. In our research, the query language is used in the context of the nested relational data model. The model provides base types boolean, integer and string, as well as type constructors tuple, set and table (set of tuples). The schema of a database is visible as a collection of table-valued variables called **base tables**.

We list the possible expression forms in the following list:

- **Constants** of type boolean (*true* or *false*), integer or string.
- **Variables** denoted with letters. In the following, we denote table expressions (base tables, set-valued attributes, and set expressions with table expressions only) with capital letters.

- **Tuple construction** of the form $\langle a_1 = e_1, \ldots, a_n = e_n \rangle$, where a_i represent attribute labels, and e_i expressions.
- **Tuple attribute selection** *e.a*, where *e* is a tuple-valued expression, and *a* is an attribute label.
- **Set expressions** union, intersection and set minus ($X \cup Y$, $X \cap Y$, $X - Y$).
- **Boolean expressions** *and*, *or* and *not* ($p \wedge q$, $p \vee q$, $\neg p$).
- Existential and universal **quantifications** $\exists x \in X \bullet p(x)$ and $\forall x \in X \bullet p(x)$, where X represents a set-valued expression and p a predicate.
- **Equality**, **less-than**, **greater-than** and **set membership** predicates:

  $e_1 = e_2, e_1 > e_2, e_1 < e_2, e_1 \in e_2$
- **Binary arithmetic operators** +, -, *, /.
- **Collect** expression:

$$\Gamma[x: f \mid p](X) \equiv \{ f(x) \mid (x \in X \wedge p(x)) \} \tag{3-1}$$

  where X is a set-valued expression, f is called the **collect function** and p the **collect predicate**. The collect expression is a shorthand for a generalized select statement:

  `select` `f` **`for`** `x` **`in`** `X` **`where`** `p`.

TMql includes selection, map and projection as special forms of collect expression:

- **Selection** of elements from a set X based on the result of a predicate p:

$$\sigma[x : p](X) \equiv \{ x \mid x \in X \wedge p(x) \}. \tag{3-2}$$

- **Map**: application of a function f to elements of a set X:

$$\alpha[x : f](X) \equiv \{ f(x) \mid x \in X \} \tag{3-3}$$

- **Projection** of a set of tuples on a sequence of attribute labels. Here, X is a table (set of tuples) and A is a sequence of attribute labels included in the labels of X. Notation *x[A]* denotes projection of tuple *x* onto attribute labels in *A*.

$$\pi_A(X) \equiv \{ x[A] \mid x \in X \} \tag{3-4}$$

The expression forms listed until now form the query language in the traditional sense of the word: the expressions formed using them are input by the user. Naturally, the syntactic form of the expression will be different for the human user: the semantics described above, however, can be found in many well-known query languages [Catt94, BaCD92]. TMql is not, however, a user-level language such as the ODMG query language: it is the input language of the logical optimizer, and as such does not contain user-level constructs such as temporary variables.

During the optimization, a TMql expression is gradually transformed into an algebraic expression. The TM algebra (**TMa**) contains a number of algebraic operators. During the optimization process, we allow therefore a mixed query-level (TMql) and algebra-level (TMa) elements within the same query. The goal of the optimization is the replacement of nested quantifiers and collects within the query with equivalent TMa elements. The resulting expression still contains some TMql forms (e.g. boolean and arithmetic operators), but many of the disadvantageous nested TMql expressions are replaced by algebra operators.

In the semantics of operators, ++ denotes **tuple concatenation**. Further, capital letter variables denote table expressions, while *f* denotes a function and *p* a predicate.

- **Product** (extended Cartesian product):

$$X_1 \times X_2 \equiv \{x_1 \mathbin{++} x_2 \mid x_1 \in X_1 \wedge x_2 \in X_2\} \tag{3-5}$$

- **Join**.

$$X_1 \ \text{join}(x_1, x_2 ; p) \ X_2 \equiv \{x_1 \mathbin{++} x_2 \mid x_1 \in X_1 \wedge x_2 \in X_2 \wedge p(x_1, x_2)\} \tag{3-6}$$

  We assume that schemas of the operands X_1 and X_2 are disjoint.

- **Semijoin**

$$X_1 \ \text{semijoin}(x_1, x_2 ; p) \ X_2 \equiv \{x_1 \mid x_1 \in X_1 \wedge \exists x_2 \in X_2 \bullet p(x_1, x_2)\} \tag{3-7}$$

  The semijoin produces those tuples from X_1 for which exists a tuple in X_2 such that the predicate holds.

- **Antijoin**

$$X_1 \ \text{antijoin}(x_1, x_2 ; p) \ X_2 \equiv \{x_1 \mid x_1 \in X_1 \wedge \neg\exists x_2 \in X_2 \bullet p(x_1, x_2)\} \tag{3-8}$$

Antijoin produces those tuples in the left operand for which there are no tuples in the right operand which fulfill the predicate. Antijoin has been proposed among others in [Clu94].

- **Nestjoin**.

$$X_1 \ \Delta_{x_1, x_2 \,;\, f \mid p \,;\, a} \ X_2 \equiv \{ x_1 \mathbin{++} \langle a = X \rangle \mid x_1 \in X_1 \wedge (X = \{ f(x_1, x_2) \mid x_2 \in X_2 \wedge p(x_1, x_2) \}) \} \qquad (3\text{-}9)$$

Nestjoin operator has been introduced in [Ste94b] to allow optimization of nested queries. For every tuple of table X_1, it concatenates the tuple with attribute a. The value of the attribute is a set obtained by applying function f to those tuples in X_2 for which predicate p is fulfilled. We abbreviate the **nested Cartesian product** $X_1 \ \Delta_{x_1, x_2 \,;\, x_2 \mid \text{true} \,;\, a} \ X_2$ as $X_1 \ \text{❖}_a \ X_2$.

- **Division**

$$X_1 \div X_2 \equiv \{ x_1[A] \mid x_1 \in X_1 \wedge X_2 \subseteq \{ x_1'[B] \mid x_1' \in X_1 \wedge x_1'[A] = x_1[A] \} \} \qquad (3\text{-}10)$$

The collection X_1 has schema AB, and collection X_2 has schema B. The result contains those tuples from X_1 (projected on A), for which the following condition holds. Assume that table X_1 is grouped on attributes A. In every group, we check whether the group of B tuples is a superset of table X_2. We return only those tuples $x_1[A]$, for which this is true.

Our operator set is similar to algebraic frameworks known in literature [Leu93, Van93, Ste95]; it does not, however, include the whole spectrum of operators found in those proposals. We concentrate only on those operators which are used in the unnesting strategy presented below.

The goal of our unnesting strategy is the replacement of nested subqueries with join expressions. This goal is achieved by application of unnesting rewrite rules, each of which replaces an element of TMql with an expression containing an algebraic operator. The application of the rules (the choice of the rule, the time and place of application) is guided by the unnesting strategy. Both the rule set and the strategy are required elements of every optimization framework. The rule set defines the repertoire of transformations applicable to query expressions; as such, it may contain multiple transformations which can be applied to the same expressions. The choice of the actual transformation to execute is determined by the optimization strategy and can be guided by the performance characteristics of the resulting expressions (for example, we may prefer to introduce joins before Cartesian products). The unnesting strategy defines therefore the **ordering** of transformations. Both the transformation set and the strategy contribute to

the optimization goal; even if the set of transformations is complete, a wrong strategy can lead to application of less beneficial transformations before more beneficial ones.
Besides unnesting rules, our rule set contains splitting transformations, which prepare predicate expressions for unnesting. The choices in splitting strategy are guided by unnesting possibilities. The unnesting and splitting strategy form the overall optimization strategy for nested queries, which is the focus of this chapter.
The interrelation of rule set and strategy is difficult to assess without implementation: that's why we have derived the strategy from the results of experiments with various strategies in the implemented Joquer optimizer (see Appendix C).

3. STRATEGY FOR UNNESTING AND SPLITTING

In the following sections, we introduce our optimization strategy. The strategy consists of unnesting and splitting component. For unnesting, we first present the set of unnesting transformations used, and propose the structuring of the rule set, based on heuristic assessment of alternative transformation results. Presentation of the splitting component includes the splitting rule set and strategy for their application for conjunctive and disjunctive predicates.
We first describe the strategy intuitively.

3.1 Global description of the strategy

The goal of our unnesting strategy is the **replacement of nested subqueries with different kinds of joins**. This goal can be achieved through application of unnesting rules to the input query. The set of unnesting rules contains alternative rules: rules which can be applied to the same input expression. Therefore, the unnesting strategy has to define which of the transformations should be applied if there is a choice. This choice is, in our case, based on the comparison of output expressions in terms of their performance, based on heuristic assumptions of relative operator performance. Because not all unnesting rules can be applied for a given input expression, and the alternative rules can be ordered heuristically, the total ordering of unnesting rules for a given expression can be determined. In this way, we create orderings of unnesting transformations for different expression forms: nested quantifiers, nested negated quantifiers and nested collect expressions.

The ordering of transformations does not mean that every unnesting rule has to be applied in a separate traversal of the expression. Because within each ordering we cater for a different expression form (nested quantifier, nested negated quantifier, nested collect), we can group the transformations from different orderings into **rulesets**, each of which contains at most one rule for each of the three expression forms. The rules within a ruleset can be applied within one traversal of the expression. The rulesets are applied in sequence to the input expression, introducing gradually expressions containing joins in the transformation ordering defined. Within each ruleset, we do not have to worry about the ordering of transformations, as all of them are applicable to distinct expression forms.

In some cases, the expressions will have to be split before unnesting can take place, due to the conditions posed on the applicability of unnesting rules. Splitting of expressions is executed using the splitting rules, which form the second component of our rule set. As is the case in unnesting, also splitting is guided by a strategy, within which two aspects have to be determined. The first aspect determines the place of splitting within the overall strategy, and the second determines the choice of alternative applications within one splitting rule.

In the following sections, we describe the elements of the strategy: unnesting and splitting. First, we list the unnesting transformations and determine their ordering. This allows us to structure the set of unnesting transformations into rule sets, each of which can be applied in one traversal of the expression. We then present the splitting rules and the splitting strategy enabling efficient join introduction for various predicate forms. The structuring of unnesting transformations and the splitting strategy leads to the formulation of the overall unnesting algorithm, in which we integrate the unnesting and splitting transformation rules.

The starting point of our rule set are the rules introduced in [Ste95]. We have modified the rule set to make it notationally clear, and to make it consistent with other optimization phases*.

We first introduce the notational conventions used, and then present the set of unnesting rules, illustrated with examples in Section 3.3. Then, in Section 3.4 we turn to the splitting component of the strategy, and integrate the two in the overall unnesting algorithm in Section 3.5.

* We describe the differences between our set and the set of [Ste95] in Appendix A.

3.2 Notational conventions

For all transformation rules, we use the notation $lhs \equiv rhs$, meaning replacement of expression *lhs* with *rhs* (one way transformation). The application of many of the transformation rules depends on the occurrence of a variable within an expression. We attempt to capture the occurrence conditions within the rule notation itself in the following manner (see also Appendix A, Section 1.3 for a discussion).

Notation $e[\,f\,]$ indicates an expression *e* which contains at least one occurrence of expression *f*. We denote the textual substitution on right hand-sides of rewrite rules with the notation $e[\,f \to g, h \to i\,]$, which indicates the expression *e* with all instances of subexpression *f* replaced with *g*, and all instances of *h* replaced with *i*. Note that the textual substitution notation does not require that the expression *e* actually contains instances of *f* and *g*. We use single letters *f*, *g*, *p*, *q* to denote arbitrary expressions, and letters *v*, *x*, *y*, *z* to denote variable occurrences. Notation v_X is used to represent the projection of tuple variable *v* (drawn typically from the result of join or product) on the attributes of one of the operands *X*.

We now proceed with the presentation of the unnesting component of the strategy.

3.3 Unnesting component

We first list the unnesting transformations together with examples of their application, and then turn to the unnesting strategy.

3.3.1 Unnesting rule set

The unnesting rules cater for different nesting situations. The first three rules are used when an existential quantifier is nested within the **WHERE** clause of the query.

The first unnesting rule allows to replace a nested existential quantifier with a semijoin. As an illustration, consider the query:

$$\sigma[x : \exists y \in Y \bullet (x\,.\,a = y\,.\,b + y\,.\,c)](X) \tag{3-11}$$

The expression can be transformed into

$$X \text{ semijoin}(x, y\,;\,(x\,.\,a = y\,.\,b + y\,.\,c))\ Y \tag{3-12}$$

The transformation used in this situation is as follows.

Transformation III.1: (semijoin introduction)

$$\Gamma[x : f \mid (\exists y \in Y \bullet p[x, y])](X) \equiv \alpha[x : f](X \text{ semijoin}(x, y ; p)\ Y) \tag{3-13}$$

The result of the transformation is introduction of a semijoin, with the predicate *p* as the join predicate. Note that a predicate can be much more complex than in our example; however, the fact that the predicate is moved to the join imposes **applicability conditions** on the left-hand expression: not every predicate can become a join predicate. Specifically, we require that the join predicate *p* is **dyadic** with respect to variable *x* and *y*: it must contain at least one reference to each of the two variables. This is reflected in the rule notation *p[x,y]*. There are other applicability conditions which have to be valid for left-hand expressions whenever joins are introduced: we defer their detailed presentation till the following section, when all unnesting rules have been presented.

If the range of the existential quantifier contains a selection, a join can be introduced instead of a semijoin, as in the following example:

$$\Gamma[x : (x\,.\,a) \mid (\exists z \in \sigma[y : (x\,.\,b = y\,.\,c)](Y) \bullet z\,.\,d > 100)](X) \tag{3-14}$$

In this case, the expression can be transformed into

$$\Gamma[v : (v\,.\,a) \mid (v\,.\,d > 100)](X \text{ join}(x, y ; (x\,.\,b = y\,.\,c))\ Y) \tag{3-15}$$

The corresponding transformation rule is as follows:

Transformation III.2: (join introduction)

$$\begin{aligned} &\Gamma[x : f \mid (\exists z \in \sigma[y : p[x, y]](Y) \bullet q)](X) \\ &\equiv \Gamma[v : f[x \to v_x] \mid q[x \to v_x, z \to v_y]](X \text{ join}(x, y ; p)\ Y) \end{aligned} \tag{3-16}$$

As in the previous transformation, the join predicate *p* should contain references to both variables *x,y.*

Even if semijoin or join cannot be introduced (due to the rule applicability conditions described below), nested existential quantifier can always be transformed into a collect expression on a Cartesian product, using the following rule. Even such a transformation can be beneficial, as it replaces a calculus-level query element (a nested quantifier) with an algebraic expression (a product).

Transformation III.3: (product introduction)

$$\Gamma[x : f \mid \exists y \in Y \bullet p](X) \equiv \Gamma[v : f[x \to v_x] \mid p[x \to v_x, y \to v_y]](X \times Y) \tag{3-17}$$

The difference between left hand-sides of rules III.1 and III.3 is the predicate *p*, which in rule III.1 should contain references to both variables for the rule to be applied. In this way, join will be constructed only if its predicate contains references to both iteration variables. For the last rule, there are no conditions on the predicate, because it is not used as a join predicate.

We now turn to nesting of negated existential quantifiers, which are handled by the following two rules.

Transformation III.4: (antijoin introduction)

$$\Gamma[x : f \mid (\neg\exists y \in Y \bullet p[x, y])](X) \equiv \alpha[x : f](X \text{ antijoin}(x, y ; p)\ Y) \tag{3-18}$$

Transformation III.5: (division introduction):

$$\Gamma[x : f \mid (\neg\exists y \in Y \bullet p)](X) \equiv \alpha[x : f]((\Gamma[v : v \mid \neg p[x \to v_x, y \to v_y]](X \times Y)) \div Y) \tag{3-19}$$

Rules III.1-III.5 are used in situations when the quantifier is in the predicate of the query (the **WHERE** clause). Also, note that the transformations III.1-III.5 are applied only if the quantifiers occur on the nesting level adjacent to the top-level collect expression.

We now turn to nested collect expressions and nested base tables. The following two rules replace occurrences of arbitrarily nested collect expressions and nested table expressions with a nestjoin and nested Cartesian product. We consider first a collect expression nested within another collect expression, as in the following example:

$$\sigma[x : x \,.\, c \subseteq \Gamma[y : y \,.\, a \mid (y \,.\, b = x \,.\, d)](Y)](X)) \tag{3-20}$$

The expression can be transformed into a nestjoin occurrence

$$\Gamma[v : v_X \mid (v \,.\, c \subseteq v \,.\, \text{ys})](X\ \Delta_{x, y\,;\, y \,.\, a\,;\, (y \,.\, b = x \,.\, d)\,;\, ys}\ Y) \tag{3-21}$$

The transformation used for this rewrite is:

Transformation III.6: (nestjoin introduction)

$$\begin{aligned}&\Gamma[x : (g \mid q)[x, \Gamma[y : (f \mid p)[x, y]](Y)]](X) \equiv \\ &\Gamma[v : (g \mid q)[x \to v_x, \Gamma[y : f \mid p](Y) \to v \,.\, ys]](X\ \Delta_{x, y\,;\, f \mid p\,;\, ys}\ Y)\end{aligned} \tag{3-22}$$

The nested collect expression is replaced by a reference to the set-valued attribute *v.ys*. This attribute is treated as a table expression in subsequent rule applications: it can therefore take part in joins.

The notation $(f \mid p)[x, y]$ illustrates the fact that variables *x*, *y* can occur either in the nestjoin function, as well as in the nestjoin predicate (or in both of them). In all these cases, the transformation can be performed. The occurrence of the inner collect within the outer collect is represented in the same way: $(g \mid q)[x, \Gamma[y : (f \mid p)[x, y]](Y)]$, corresponding to nesting in **SELECT** and **WHERE** clause of the SQL query. Note that the inner collect expression can be arbitrarily nested within the *g* or *q* expression. This is contrary to the unnesting rules III.1-III.5, which are applied only in situations when the inner quantifier is on the nesting level adjacent to the collection *X*. Also, rule III.6 requires that outer nestjoin function or predicate contains a reference to variable *x* outside of the one already present in the nested collect expression.

If the nested collect predicate or function does not contain references to iteration variables x and y, we create a nested Cartesian product.

Transformation III.7: (nested Cartesian product introduction)

$$\Gamma[x : (f \mid p)[x, Y]](X) \equiv \Gamma[v : (f \mid p)[x \to v_x, Y \to v\,.\,ys]](X \;\mathbin{❖_{ys}}\; Y) \tag{3-23}$$

When describing the rules, we briefly introduced some of the applicability conditions which have to be evaluated before the rewrite can be effected. We now list all those conditions.

3.3.2 Applicability conditions for unnesting rules

The condition on adjacency of nesting levels in transformations III.1-III.5 is an example of an applicability condition; a condition which the left-hand expression has to fulfill to be transformed into right-hand side expression. Some of the applicability conditions can be expressed in the rule notation itself (**explicit** applicability conditions). In this way, for example, we can distinguish between an arbitrary expression *p* and the expression *p[x,y]* containing variable references *x* and *y*. However, not all applicability conditions can be expressed in the rule notation: that's why they have to be noted separately here.

The basic condition which is valid for all transformations is that no transformation should introduce free variables in the expression. Also, wherever capital-letter variables are used in transformations, table expressions are expected: base tables, set-valued attributes, set expres-

sions with base table/set-valued attribute operands or algebra expressions. Additionally, we formulate the following **implicit** applicability conditions:

1. Join predicates *p* in transformations III.1, III.2, III.4 and III.6 consist of atomic terms only (attribute references and comparisons of attribute references)
2. Nestjoin function *f* in Transformation III.6 and join predicates *p* in transformations III.1, III.2, III.4 and III.6 do not contain base table occurrences.

The conditions are motivated by performance considerations. Join implementations are most efficient if join predicates are consisting of atomic terms: indexing and sorting techniques [Mis92, Gra93a] can then be used to improve performance. As to the second condition: we want to exclude base tables from join predicates and nestjoin functions, because we want base tables to take part in joins (semi-, anti-, plain and nestjoins). If a base table reference would be transferred into a join predicate, there would be no way in which subsequent transformations would be able to form a join with that base table.

Introduction of joins in the query leads to the possibility of pushing expressions to join operands. The pushing rules are for the most part variations of standard formulations and are presented in Appendix B.

We have presented the unnesting rewrite rules used to transform nested quantifiers and subqueries into joins. Besides the rules themselves, we need a strategy of their application. Our unnesting strategy is based on the structuring of transformations in the order of their benefit for the result. To introduce the structuring, we first present the transformation ordering.

3.3.3 Ordering of unnesting transformations

Transformation ordering allows to choose between alternative unnesting transformation rules, based on the comparison of the structure of resulting expressions and heuristic assumptions on relative operator performance. To define the ordering of operators, we first have to analyze the unnesting rule set and identify the alternative transformation rules. We note here that we attempt to find the largest alternative transformation sets: once we determine the ordering within these sets, subsets of these rule sets are also ordered.

There are following alternative rules within our unnesting rule set (we provide example queries with some of the alternatives):

1. For queries with **existential quantifiers within selects**, the alternative transformations are III.1 (semijoin introduction), III.3 (Cartesian product introduction) and III.7 (nested Cartesian product). We take

$$\sigma[x : \exists y \in Y \bullet (x\,.\,a = y\,.\,b)](X) \tag{3-24}$$

as an example of a query for which all alternative transformations can be applied.
If transformation III.1 is applied, semijoin is introduced:

$$X \text{ semijoin}(x, y\,;\,(x\,.\,a = y\,.\,b))\ Y \tag{3-25}$$

Alternatively, transformation III.3 can be applied to introduce a Cartesian product:

$$\Gamma[v : v_X \mid (v_X\,.\,a = v_Y\,.\,b)](X \times Y) \tag{3-26}$$

or transformation III.7 to introduce the nested Cartesian product:

$$\Gamma[v : v_X \mid \exists y \in v\,.\,ys \bullet (v_X\,.\,a = y\,.\,b)](X\ \ \mathbin{❖_{ys}}\ \ Y) \tag{3-27}$$

Determining transformation ordering means in this case choosing between the introduction of a semijoin, selection on a Cartesian product, or a collect with a nested quantifier on a nested Cartesian product.

2. For queries with **negated existential quantifiers within selects**, the alternative transformations are III.4 (introduction of antijoin), III.5 (division and Cartesian product) and III.7 (nested Cartesian product). All of these rules can be applied for queries such as:

$$\sigma[x : \neg\exists y \in Y \bullet (x\,.\,a > y\,.\,b)](X) \tag{3-28}$$

If III.4 is applied, the resulting expression will be an antijoin:

$$X \text{ antijoin}(x, y\,;\,x\,.\,a > y\,.\,b)\ Y \tag{3-29}$$

After application of III.5 we will get a division with a selection on a Cartesian product:

$$(\Gamma[v : v \mid \neg(v_X\,.\,a > v_Y\,.\,b)](X \times Y)) \div Y \tag{3-30}$$

and III.7 will result in a collect over a nested Cartesian product with a quantifier in the collect function:

$$\Gamma[v : v_X \mid \neg\exists y \in v \,.\, \mathrm{ys} \bullet (v_X \,.\, a > y \,.\, b)](X \;\mathbin{❖_{ys}}\; Y) \tag{3-31}$$

3. For queries with **collects within collects**, the alternative transformations are III.2 (introducing a join), III.6 (introducing a nestjoin) and III.7 (introducing a nested Cartesian product). Example query is

$$\sigma[x : (\exists z \in \sigma[y : (x \,.\, a = y \,.\, b)](Y) \bullet (x \,.\, c > z \,.\, b))](X) \tag{3-32}$$

We note that rule III.1 cannot be applied to this query to construct a semijoin, even though there is an existential quantifier within a collect. This is caused by the fact that the range of the existential predicate $\sigma[y : (x \,.\, a = y \,.\, b)](Y)$ contains a reference to the iteration variable *x*: the semijoin of the selection with table *X* would make *x* a free variable.

Using III.2, we get

$$\Gamma[v : v_X \mid (v_X \,.\, c > v_Y \,.\, b)](X \;\mathrm{join}(x, y \,;\, (x \,.\, a = y \,.\, b))\; Y) \tag{3-33}$$

while III.6 produces

$$\Gamma[v : v_X \mid \exists z \in v \,.\, \mathrm{ys} \bullet (v_X \,.\, c > z \,.\, b)](X \;\Delta_{x, y \,;\, y \mid x \,.\, a = y \,.\, b \,;\, ys}\; Y) \tag{3-34}$$

Finally, III.7 produces

$$\Gamma[v : v_X \mid \exists z \in \sigma[y : (v_X \,.\, a = y \,.\, b)](v \,.\, \mathrm{ys}) \bullet (v_X \,.\, c > z \,.\, b)](X \;\mathbin{❖_{ys}}\; Y) \tag{3-35}$$

After introducing the alternative transformations, we turn to heuristic ordering of alternatives. The basis of our heuristic approach is the fact that for a given query we only have to choose between the alternative transformations, not between all transformations. We therefore do not have to base ourselves on a performance model, but we can choose the orderings within alternative rule sets based on heuristics. The heuristics can be used also because we determine the choice based not only on the relative performance of the alternative operators, but on the relative performance of the alternative resultant expressions. Therefore, in case of expressions (3-25) and (3-26), we do not compare a semijoin with a Cartesian product, but with the complete transformation result: a collect on a Cartesian product.

We begin with the ordering of alternative operators, and follow with ordering of alternative transformation results.

Our ordering of operators is based on the fact that most operand implementations can achieve much higher performance than nested-loop processing due to the richness of implementation methods for algebra operators [Mis92, Gra93a]. In most cases, the execution system will be able to choose the best implementation from a number of alternatives, depending on the characteristics of the input expression, presence of sorted collections and indexes. This is the **primary reason** for introduction of algebraic processing of queries. In most situations, introducing algebraic operators such as joins will open numerous implementation possibilities in the physical optimization phase. That is why **join introductions will almost always be more beneficial than introduction of Cartesian products or leaving the expression as is**. This forms the general heuristic used in ordering of operators.

The worst possible situation in terms of join implementation will occur if we will not be able to make use of advanced join implementation methods; in this case, we will have to fall back to nested-loop join implementation. To provide an ordering of operators, we have to consider also this case, however rare its occurrence may be. In this case, we can provide the ordering by assessing the number of tuple retrieval and storage operations for alternative transformations. This assessment leads to the following heuristics. We stress again that the heuristics describe the situation in which joins are implemented using a nested-loop algorithm.

We consider the transformation alternatives as described above. For nested (negated) existential quantifiers, we have to choose between a semi or antijoin and a Cartesian product (equations (3-25), (3-26) and (3-29),(3-30)).

Heuristic III.1:

Semijoin and antijoin have the same or better performance as the Cartesian product.

The motivation for the heuristics is as follows. Assume that both joins and the product have been implemented using a nested loop algorithm (which is the most straightforward algorithm for those operators). If the number of tuples in left and right operand is *R* and *S*, the number of tuple accesses for the Cartesian product will be $R \times S$. Both the semijoin and join implementation traverse right-hand tuples until a matching (with respect to the join predicate) tuple is found. In the worst situation, this will involve $R \times S$ accesses as well. However, even in this case the performance of semi- and anti-join will be better because right-hand tuples do not have to be included in the result. Semi-joins and antijoin will therefore have the same number of accesses in the worst case, but will have to create smaller result tuples.

For situations when a collect is nested within a collect, we have to choose between a nestjoin and a join introduction (equations (3-33),(3-34)). We use the following heuristic.

Heuristic III.2:

Join has the same or better performance as the nestjoin.

Assuming again nested-loop processing, both implementations find matching tuples in the right-hand-side operand. Let's further assume that for each left-hand tuple there are *s* matching right-hand tuples. For both operators, for each left-hand tuple, *s* right-hand tuples have to be accessed. Both operators will therefore have equal number of tuple accesses. In the join, the matching right-hand tuples are concatenated with left-hand tuples and added to the result. In the nestjoin, the matching right-hand tuples are concatenated in a set-valued attribute (temporary collection), which is later added to the left-hand tuple. In both cases, for each of the R left-hand tuples, this leads to $R \times s$ tuple creations. Nestjoin and join have therefore comparable performance in terms of tuple accesses and creations. The reason for preferring joins above nestjoins lies in the additional operations which join does not have to execute, while the nestjoin has: creation of temporary collections and evaluations of the nestjoin function.

The set-valued attribute of a nestjoin will be typically maintained in a temporary collection whose maintenance costs add to the nestjoin costs. In case of a join, we will have to create one temporary collection (the result), while in the nestjoin we will have to create at least one temporary collection for the result and at worst $R + 1$ temporary collections (for each of the left-hand tuples and for the result). Nestjoin will therefore create the same or greater amount of temporary collections than join.

Besides creation of temporary collections, the nestjoin has to perform additional activity in comparison to the join: for each of the matching right-hand tuples, it has to evaluate the nestjoin function. This means additional costs for the nestjoin: those costs can be neglected only if nestjoin function is an identity. In this case, no additional nestjoin function evaluations will take place.

These two aspects lead to the choice of join above nestjoin in the heuristic.

Heuristic III.3:

Cartesian product has the same or better performance as the nested Cartesian product.

The motivation is similar as in previous heuristics. The amount of tuple accesses and additions will also be the same in both cases. In any case, both operators will have to create one temporary collection for the result. However, nested Cartesian product will have to create additional

temporary collections for all left-hand tuples (*R* collections). This will make products more beneficial than nested Cartesian products.

We now provide the heuristic ordering of transformations for alternative sets 1-3, using operator heuristics III.1-III.3:

1. We consider nested existential quantifiers, with alternative rules III.1, III.3 and III.7. Even if the semijoin were to be executed equally fast as the product (as dictated by Heuristic III.1), the result of III.3 will be less efficient because of the selection which still has to be performed. Therefore, transformation III.1 is preferred to transformation III.3. According to Heuristic III.3, we should choose rule III.3 above III.7. The choice is also motivated by the fact that rule III.7 still leaves the quantifier in the result expression. The removal of quantifiers is one of the goals of our optimization framework. For the alternative rule set {III.1, III.3, III.7} we therefore choose to apply III.1 before III.3 and III.3 before III.7. Ordering is **<III.1, III.3, III.7>**.

2. We consider the alternative rules III.4, III.5 and III.7 (negated existential quantifiers). According to Heuristic III.1 the antijoin will almost always be better than a Cartesian product. The fact that the result of III.5 has additionally to perform a selection and division leads to the choice of III.4 before III.5. Similarly as in the previous case, we prefer to remove the existential quantifiers from the collect expression, therefore we choose III.5 above III.7. The ordering is **<III.4, III.5, III.7>**.

3. We consider the alternative rules III.2, III.6 and III.7. Heuristic III.2 dictates to introduce a join before a nestjoin. The choice is yet more obvious, if we consider that the result of III.6 still contains a quantifier in the collect function. While in the input expression we had a quantifier on selection, in the result we get a quantifier on nestjoin attribute *v.ys*. The choice between nestjoin introduction in transformation III.6 and nested Cartesian product in III.7 is less obvious. Results of both III.6 and III.7, if implemented using nested loop processing, will have the same number of tuple accesses and evaluations of function *f* and predicate *p*. However, if nestjoin is implemented with better performance than nested-loop processing (by using sorting and hashing techniques), III.6 will be preferred. Transformation ordering is **<III.2, III.6, III.7>**.

The transformation orderings allow us to structure the set of unnesting rules into rulesets containing non-conflicting rules: rules which can be applied non-ambiguously in one traversal of the expression. We derive the following rulesets from the transformation orderings:

- **Join transformations:** III.1, III.2, III.4. Those transformations introduce following operators: semijoin, join, and antijoin.
- **Nestjoin transformation:** III.6**.** Note that the nestjoin transformation cannot be placed together with join transformations due to the described ambiguity which can arise for nested collect within collect queries.
- **Product transformations**: III.3, III.5. Those transformations introduce Cartesian products and divisions.
- **Nested product transformation**: III.7. This transformation introduces nested Cartesian products.

Having determined the structuring of the unnesting transformations, we turn to the second element of the unnesting strategy: the splitting component.

3.4 Splitting component

Due to the applicability conditions specified, in some cases it will not be possible to apply unnesting rules directly: for example because of predicates which contain references to variables other than the two potential join variables. In this case, splitting rules are used to isolate the predicates which are amenable to join introduction.

We begin with presentation of the splitting rules. We then analyse the strategy for disjunctive and conjunctive predicates.

3.4.1 Splitting rules

Splitting prepares collect functions and predicates for subsequent join introduction. We list here the rules for splitting predicates (rules for splitting functions are deferred to the appendix):

Transformation III.8: (conjunctive collect predicate)

$$\Gamma[x : f \mid p \wedge q](X) \equiv \Gamma[x : f \mid q](\sigma[x : p](X)) \tag{3-36}$$

Transformation III.9: (disjunctive collect predicate)

$$\Gamma[x : f \mid p \vee q](X) \equiv \Gamma[x : f \mid p](X) \cup \Gamma[x : f \mid q](Y) \tag{3-37}$$

Transformation III.10: (conjunctive quantifier matrix)

$$\exists x \in X \bullet p \wedge q \equiv \exists x \in \sigma[x : p](X) \bullet q \tag{3-38}$$

Transformation III.11: (disjunctive quantifier matrix)

$$\exists x \in X \bullet p \vee q \equiv \exists x \in X \bullet p \vee \exists x \in X \bullet q \tag{3-39}$$

Transformation III.12: (negated disjunctive matrix)

$$\neg\exists x \in X \bullet p \vee q \equiv \neg\exists x \in X \bullet p \wedge \neg\exists x \in X \bullet q \tag{3-40}$$

The splitting transformations III.8 and III.10 can be applied in two different ways: during optimization, the choice has to be made which of the two subexpression *p* and *q* is going to be taken as a selection predicate on the right side. This choice is guided by the splitting strategy, which is presented in the following section.

3.4.2 Splitting alternatives

Unnesting rules cannot be applied indiscriminately to input expressions. Each of the unnesting rules poses applicability conditions on the input query: those conditions guarantee that the resulting join expressions can be efficiently transformed and implemented. In the description of the applicability conditions for the unnesting rule set in Section 3.3.2, we have listed the conditions posed on collect predicates and functions, requiring them to contain both join variables and no base table references.

The splitting rules are used to transform predicates into forms conforming to the applicability conditions of unnesting rules. As such, they are guided by the unnesting rules in their alternative applications. In this section, we describe the way in which splitting is guided in our optimization strategy: both by the unnesting transformations and the form of the predicate to split.

The splitting rules III.8-III.12 are used to split both conjunctive and disjunctive predicates. We first consider disjunctive predicates.

3.4.3 Splitting of disjunctive predicates

The presence of disjunctive predicates causes application of rules III.9, III.11 and III.12, which introduce a **range split**: a duplication of collection X in the right-hand expression (consequently, we name transformations III.9, III.11 and III.12 **range splitting transformations**). Range split is an undesirable effect in query optimization, because eventually both instances of the collection will have to be unnested. This may introduce unneeded nested Cartesian product invocations, and in general causes the resulting expressions to become more complex*. Therefore, in the splitting strategy of disjunctive predicates, we guide ourselves with the following heuristic.

Heuristic III.4:

Range splits should be avoided if possible.

In our splitting approach, we avoid the range split by introduction of a Cartesian product, using product rules III.3 and III.5. Before a range split is to be executed, we first attempt to apply any of the product transformations.

Only if no product transformations can be applied, is the range split actually executed. We motivate the heuristic by the result of the evaluation of the strategy options for our set of test queries in Appendix C. During experimentation, we noted that avoiding range splits by Cartesian product introduction improves the quality of resulting expressions. This has been observed for queries C.3, C.4, C.8, C.9, C.14 and C.19 in the appendix. We illustrate the benefits in the following example taken from the test set.

In the example query we use two alternative splitting strategies. One strategy splits the predicates fully, without taking notice of disjunctive predicates, while the other uses the product transformations to avoid range splitting, and returns to unnesting transformations after every split. The second procedure is the one applied in our unnesting strategy.

1. Full splitting strategy.

The example query is as follows (we chose for the generic notation of a dyadic predicate introduced above to make the presentation clear):

$$\alpha[x : \sigma[y : p[x, y] \vee \exists z \in Z \bullet q[x, z] \vee r[y, z]](Y)](X) \tag{3-41}$$

* This observation is supported by the results of experiments with the strategy described in Appendix C.

The expression can be unnested using either join or nestjoin transformations: we analyze whether each of the unnesting rules can be applied to different candidate operand pairs (X,Y), (X,Z) and (Y,Z):

- (X,Y): The select on Y is nested within a map on X: the structure corresponds to the nestjoin transformation III.6. In this case, the predicate $p[x, y] \vee \exists z \in Z \bullet q[x, z] \vee r[y, z]$ would become the nestjoin predicate. However, as the predicate contains reference to base table *Z*, the rule cannot be applied, according to applicability conditions for join introductions.
- (X,Z): Here, rule III.1 cannot be applied, because the existential quantifier on Z is not on an adjacent nesting level with respect to the map on collection X.
- (Y,Z): For the rule III.1 to be applied, the quantifier matrix $q[x, z] \vee r[y, z]$ should contain only references to variables *y* and *z*. As it contains also a reference to variable *x*, the rule cannot be applied.

We now split the expressions using rules III.8 to III.12, to enable application of unnesting rules. Contrary to our strategy, we split the predicate expressions fully, without returning to the unnesting after every successful split. We apply rules III.9, III.11 and again III.9:

$$\begin{aligned}&\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \exists z \in Z \bullet q[x, z] \vee r[y, z]](Y)](X) \equiv \\ &\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \exists z \in Z \bullet q[x, z] \vee \exists z \in Z \bullet r[y, z]](Y)](X) \equiv \\ &\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \exists z \in Z \bullet q[x, z]](Y) \cup \sigma[y : \exists z \in Z \bullet r[y, z]](Y)](X)\end{aligned} \tag{3-42}$$

The map function is now fully split. The unnesting rules III.1-III.7 are now applied. We first unnest the expression using rules III.1 (for a semijoin between *Y* and *Z*) and III.6 (for a nestjoin between *X* and *Y*).

$$\begin{aligned}&\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \exists z \in Z \bullet q[x, z]](Y) \cup (Y \text{ semijoin}(y, z\,;\, r[y, z])\ Z)](X) \equiv \\ &\alpha[w : w \,.\, \text{ys} \cup \sigma[y : \exists z \in Z \bullet q[x \to w_X, z]](Y) \cup (Y \text{ semijoin}(y, z\,;\, r[y, z])\ Z)](X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, ys}\ Y)\end{aligned} \tag{3-43}$$

We then introduce a Cartesian product between *Y* and *Z* using rule III.3.

$$\begin{aligned}&\alpha[w : \\ &w \,.\, \text{ys} \cup \Gamma[v : v_Y \mid q[x \to w_X, z \to v_Z]](Y \times Z) \dot{\cup} (Y \text{ semijoin}(y, z\,;\, r[y, z])\ Z) \\ &](X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, ys}\ Y)\end{aligned} \tag{3-44}$$

Then, the nestjoin result and the Cartesian product can be nestjoined using rule III.6:

$$\alpha[w : \quad (3\text{-}45)$$
$$w \,.\, \text{ys} \cup w \,.\, \text{y2s} \cup (Y \text{ semijoin}(y, z\,;\, r[y, z])\ Z)$$
$$]((X \ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, ys}\ Y)\ \Delta_{w, v\,;\, v_Y \mid q[x \to w_X,\, z \to v_Z]\,;\, \text{y2s}}\ (Y \times Z))$$

The final unnesting which can take place is the introduction of the nested Cartesian product between the nestjoin result and the semijoin, using rule III.7. As the result of the optimization, we get:

$$\alpha[w : w \,.\, \text{ys} \cup w \,.\, \text{y2s} \cup w \,.\, \text{y3s}] \quad (3\text{-}46)$$
$$(((X \ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, ys}\ Y)\ \Delta_{w, v\,;\, v_Y \mid q[x \to w_X,\, z \to v_Z]\,;\, \text{y2s}}\ (Y \times Z))\ \text{❖}_{y3s}\ (Y \text{ semijoin}(y, z\,;\, r[y, z])\ Z))$$

The result of the optimization is a complex algebra expression containing a Cartesian product and a nested Cartesian product.

2. Alternative strategy: unnesting after every split.

For comparison, we optimize the input expression (3-41) using an alternative strategy, in which we attempt to avoid range splits and introduce joins as soon as possible by returning to unnesting after every split. This is the strategy applied in our optimizer.
In this case, the input expression is first split using rule III.9 into

$$\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \exists z \in Z \bullet q[x, z] \vee r[y, z]](Y)](X) \quad (3\text{-}47)$$

We now return to unnesting rules, which allows us to introduce a nestjoin between X and Y using rule III.6:

$$\alpha[w : w \,.\, \text{ys} \cup \sigma[y : \exists z \in Z \bullet q[x \to w_X, z] \vee r[y, z]](Y)](X \ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, ys}\ Y) \quad (3\text{-}48)$$

No other join or nestjoin transformations can be introduced. We attempt further splitting. The only applicable rule is the range splitting rule III.9: conforming to our policy of avoiding range splits if possible, we apply product transformation III.3 to collections Y and Z. Because we transform into a Cartesian product, the predicate $q[w_X, z] \vee r[y, z]$ does not have to contain y and z as the only variables. We get

$$\alpha[w : w \,.\, \text{ys} \cup \Gamma[v : v_Y \mid q[x \to w_X, z \to v_Z] \vee r[y \to v_Y, z \to v_Z]](Y \times Z)] \quad (3\text{-}49)$$
$$(X \ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, ys}\ Y)$$

Having performed the unnesting, we cannot proceed with splitting, because the range split would be still introduced. As specified in the unnesting algorithm, we return to unnesting, and apply rule III.6 to achieve:

$$\alpha[w : w \,.\, \mathrm{ys} \cup w \,.\, \mathrm{y2s}] \qquad (3\text{-}50)$$
$$((X \;\Delta_{x,y\,;\,y\,|\,p[x,y]\,;\,ys}\; Y) \;\Delta_{w,v\,;\,v_Y\,|\,q[x \to w_X,\, z \to v_Z] \vee r[y \to v_Y,\, z \to v_Z]\,;\,y2s}\; (Y \times Z))$$

The result expression is less complex than expression (3-46) achieved with “full split” strategy. It does not contain the nested Cartesian product introduced because of the range split.

Similar benefits have been observed in other queries with disjunctive quantifier matrices (queries C.3, C.4, C.8, C.9, C.14 and C.19 in Appendix C*). If the product transformations are not applied, quantifier ranges are split. This situation could be beneficial, if it would lead to introduction of profitable joins later in the optimization in place of Cartesian products. However, in the example queries we noted that even if some joins are introduced, the Cartesian products still appear in the result. This is caused by the fact that range splits only isolate the matrix elements which can be transformed to joins, but at the same time the elements which lead to products are still left in the expression. Range splits allow joins to be introduced, but they do not improve the quality of the resulting expression as a whole: we therefore attempt to avoid them if possible.

3.4.4 Splitting of conjunctive predicates

We now turn to splitting of **conjunctive** collect predicates and quantifier matrices in rules III.8:

$$\Gamma[x : f \mid p \wedge q](X) \equiv \Gamma[x : f \mid q](\sigma[x : p](X)) \qquad (3\text{-}51)$$

and III.10:

$$\exists x \in X \bullet p \wedge q \equiv \exists x \in \sigma[x : p](X) \bullet q \qquad (3\text{-}52)$$

When applying the transformations, a decision has to be made which of the predicates *p* and *q* should be placed in the selection predicate on the right hand-side. In our splitting strategy, we guide the splitting choices by the possibility of join and nestjoin introduction, which is realized by rules III.1

* In the appendix, for each of those queries we present the alternative strategies and benefits achieved.

$$\Gamma[x : f \mid (\exists y \in Y \bullet p[x, y])](X) \equiv \alpha[x : f](X \text{ semijoin}(x, y ; p)\ Y) \tag{3-53}$$

and III.2

$$\begin{aligned}&\Gamma[x : f \mid (\exists z \in \sigma[y : p[x, y]](Y) \bullet q)](X)\\ &\equiv \Gamma[v : f[x \to v_x] \mid q[x \to v_x, z \to v_y]](X \text{ join}(x, y ; p)\ Y)\end{aligned} \tag{3-54}$$

which both detect an existential quantifier within the collect predicate.

In our splitting strategy for conjunctive predicates, we evaluate which of the two splitting choices will lead to introduction of joins. To choose whether *p* or *q* should be put in the outer collect, we use the following procedure.

If our input expression is of the form

$$\Gamma[x : f \mid p \wedge q](X) \tag{3-55}$$

we check whether one of predicates *p, q* in the expression is an existential quantifier of the form

$$\exists y \in Y \bullet r[x, y] \tag{3-56}$$

where *x* is the outer collect variable. If such a predicate is present as *p* or *q*, we choose it for the inner select predicate. In the following unnesting step, this will lead to direct introduction of a closed flat join according to transformation III.1. In essence, we perform the following transformation

$$\Gamma[x : f \mid \exists y \in Y \bullet r[x, y] \wedge q](X) \equiv \Gamma[x : f \mid q](\sigma[x : \exists y \in Y \bullet r[x, y]](X)) \tag{3-57}$$

Similar considerations apply to the second splitting rule III.10. Here, the input expression is of the form:

$$\exists x \in X \bullet p \wedge q \tag{3-58}$$

Also here, we base our choice for the predicate to place in the select on its suitability for subsequent join introduction, based on rule III.2. However, this time we examine the expression surrounding the existential quantifier. For rule III.2 to be applied, the surrounding expression has to be a collect expression with iteration variable *y*:

$$\Gamma[y : f \mid (\exists x \in X \bullet p \wedge q)](Y) \tag{3-59}$$

We now examine expressions p and q and choose the one which is a closed dyadic predicate of (x,y). We essentially perform then the following transformation:

$$\Gamma[y : f \mid (\exists x \in X \bullet p[x, y] \wedge q)](Y) \equiv \Gamma[y : f \mid (\exists x \in \sigma[x : p[x, y]](X) \bullet q)](Y) \tag{3-60}$$

This allows us to introduce a join in the following unnesting step.

The above two techniques allow us to introduce joins in the following unnesting step. However, if that is not possible, we still want to guide the choice of the conjunct to split. This can be achieved by transforming the predicate in such a way that in the next unnesting step it will be transformed not into a join, but into a nestjoin, using transformation III.6:

$$\begin{array}{l}\Gamma[x : (g \mid q)[x, \Gamma[y : f \mid p[x, y]](Y)]](X) \equiv \\ \Gamma[v : (g \mid q)[x \rightarrow v_x, \Gamma[y : f \mid p](Y) \rightarrow v \,.\, ys]](X \;\Delta_{x,\, y\,;\, f \mid p\,;\, ys}\; Y)\end{array} \tag{3-61}$$

Taking expression

$$\exists x \in X \bullet p[x, y] \wedge q[x, z] \tag{3-62}$$

as example, we proceed by analyzing which of the conjuncts p and q is a better candidate for splitting. We assume that both variables y and z are iteration variables of collect expressions surrounding the quantifier. In our strategy, we choose this conjunct which allows for a nestjoin which is as close to the top level as possible. To illustrate, consider the expression

$$\alpha[z : \sigma[y : \exists x \in X \bullet p[x, y] \wedge q[x, z]](Y)](Z) \tag{3-63}$$

The two possible splitting choices lead to expressions

$$\alpha[z : \sigma[y : \exists x \in \sigma[v_0 : p[x \rightarrow v_0, y]](X) \bullet q[x, z]](Y)](Z) \tag{3-64}$$

$$\alpha[z : \sigma[y : \exists x \in \sigma[v_0 : q[x \rightarrow v_0, z]](X) \bullet p[x, y]](Y)](Z) \tag{3-65}$$

If transformation III.6 were to be applied to the alternative expressions, it would result in the following two expressions

$$\alpha[z : \sigma[v_1 : \exists x \in v_1 \,.\, xs \bullet q[x, z]](Y \;\Delta_{y,\, v_0\,;\, v_0 \mid p[x \rightarrow v_0,\, y]\,;\, xs}\; X)](Z) \tag{3-66}$$

$$\alpha[v_1 : \sigma[y : \exists x \in v_1 \,.\, xs \bullet p[x, y]](Y)](Z \;\Delta_{z,\, v_0\,;\, v_0 \mid q[x \rightarrow v_0,\, z]\,;\, xs}\; X) \tag{3-67}$$

The expression (3-67) is better than (3-66), because it brings the collection *X* to the top level: the other option creates a nested nestjoin invocation, which still has to be brought to the top level. In choosing the conjunct to place in the selection, we therefore choose the one which allows us to introduce nestjoins with “higher-placed” collections.

3.4.5 Splitting strategy: remarks

Our splitting strategy provides good results throughout the whole spectrum of example queries. For completeness, we note that there are cases in which the approach chosen is not the most beneficial: in the appendix we present an alternative optimization of the query (C.15) which achieves better results than our strategy. However, the result achieved in this query requires a planning approach, in which we determine the sequence of splitting steps required to achieve a given join. For each of the splitting steps in the sequence, the choices for expression to put in the selection predicate would have to be made. The planning approach would have not only to guide itself by the possibility of local join introduction, but also by the influence of splitting decisions on the quality of the whole expression. As such, it is much more complicated to implement and expensive to execute than the “greedy” approach presented here, which attempts to introduce joins as fast as possible by analyzing local environments of expressions.

In the last sections, we have introduced two elements of the unnesting strategy: the **ordering and structuring of unnesting transformations**, and the **splitting strategy** used to transform predicates into forms suitable for join introduction. Those two elements can now be integrated in the overall unnesting strategy, which describes the algorithm used to apply unnesting and splitting transformations.

3.5 Overall unnesting algorithm

The unnesting strategy is applied in loops. One unnesting loop consists of the following stages:

1. Top-down unnesting of the query using join transformations III.1, III.2, III.4. Those rules introduce semijoins, joins, antijoins where possible. After every unnest introduction, push expressions to join operands if possible and return to begin of step 1.
2. Top-down unnesting of the query using nestjoin transformation III.6. After every unnest, push expressions to join operands if possible and return to step 1.

3. Top-down splitting of expressions using splitting rules III.8, III.9, III.10, III.11, III.12. If range splitting rule is to be applied, try introduction of product transformations III.3, III.5 and return to step 1. Guide splitting of conjunctive expressions by possible join and nestjoin introductions. Return to step 1 if a split has been performed. If no split has been performed, proceed to step 3.

4. Eliminate remaining table expressions by applying product transformations (III.3, III.5) and nested product transformations (III.7). Both the product and the nested product applications are introduced in separate top-down expression traversals, so that all possible products are introduced before nested products are introduced.

The unnesting strategy is applied in one unnesting loop: we repeat the loop until no transformation is applied. In this way, we are able to join set-valued attributes introduced in previous loops. The expressions are traversed in a top-down manner: from the outer to inner "layers" of the expression. For join introductions, first the outermost base table will be considered, and join transformations will be attempted with all inner table expressions, in the order of nesting level. This allows the joins to be introduced at the top level first. For splitting rules, top-down application allows to split expressions from the outside of the predicate. After every split, we return to the join transformations, so that they are applied as soon as the predicates are brought to suitable form.

In our strategy, we achieve preferable introduction of more efficient operators before less efficient ones according to the heuristics specified: we first choose to introduce semi-, anti- and plain joins, then nestjoins, Cartesian products, and finally nested Cartesian products. Because we return to unnesting after every successful step, our splitting strategy is "greedy": it is guided by the possible introduction of joins in the next unnesting step. We therefore do not employ algorithms which would plan a sequence of splitting steps as to achieve a specified join. This is motivated by the fact that, as shown in our examples, join introduction by itself is not a suitable goal for such a planning system; we have seen that introducing a join (or rather splitting the expression to allow for a join) in one part of the expression may still lead to unprofitable product subexpressions in another part. It is therefore not beneficial to use a planning system with only join introduction as a goal: a proper planning system would have to analyze the consequence of every split on the whole expression. Such a global analysis can be very

complex and very expensive in terms of performance: we have chosen for a simple planning system which guides itself by the following heuristic principles:

- introduce joins as soon as possible by returning to unnesting after every guided split
- do not split ranges unnecessarily

Our strategy is the result of experiments with the implemented optimizer. We have presented the test query set in Appendix C, together with evaluation of different strategy options.

3.5.1 Unnesting algorithm: example

We illustrate the operation of the strategy with an extended example. The example is based on the Query Optimizer Bakeoff [Bake96], which provides a set of queries for testing of different aspects of optimizer operation.

We assume the following schema for our database (using the ODMG notation of [Catt96])*:

```
interface Client: Person
( extent CLIENTS ) {
  attribute Long    bill;
};

interface Person
( extent PERSONS ) {
  attribute String  first_name;
};

interface Hotel
( extent HOTELS ) {
  attribute String  name;
  attribute Address address;
};

interface Address {
  attribute City    city;
  attribute String  street;
  attribute Long    number;
};

interface City
( extent CITIES ) {
  attribute String  name;
};
```

```
interface Tour
( extent TOURS ){
  attribute Long    price;
};

interface Place_to_go
( extent PLACES ) {
  attribute String  name;
};

interface Artist : Person
( extent ARTISTS ) {
  attribute Long    birthyear;
  attribute City    city;
};

interface Theater
( extent THEATERS ) {
  attribute String  name;
  attribute City    city;
  attribute Person  manager;
};
```

Figure 3-1 Schema of the Bakeoff database

* We only list attributes relevant to our examples.

The schema is visible through a number of extents (base tables): CLIENTS, HOTELS, PLACES, CITIES, THEATERS, TOURS, ARTISTS, each of which contains sets of instances of the listed types.

As an example, we use the query “for every hotel, list clients which prefer this hotel or are managers of local theaters”:

$$\alpha[h : \sigma[c : (c\,.\,\text{preferred} = h) \vee \exists t \in \text{THEATERS} \bullet (t\,.\,\text{manager} = c) \wedge (t\,.\,\text{city} = h\,.\,\text{city})](\text{CLIENTS})](\text{HOTELS}) \tag{3-68}$$

Following the unnesting algorithm, we attempt to apply join transformations. Unfortunately, none of the join or nestjoin transformations can be applied due to the variable dependencies in the predicates. Consider for example the pair *(HOTELS, CLIENTS)* and nestjoin transformation III.6. In terms of structure, the transformation is applicable: the select is nested within the map function. However, the selection predicate

$$(c\,.\,\text{preferred} = h) \vee \exists t \in \text{THEATERS} \bullet (t\,.\,\text{manager} = c) \wedge (t\,.\,\text{city} = h\,.\,\text{city}) \tag{3-69}$$

contains a reference to base table *THEATERS*: if transformation III.6 would be applied, this reference would appear in the nestjoin predicate. This would prevent the *THEATERS* collection to take place in subsequent join introductions, and that’s why transformation III.6 cannot be applied to our input expression. Similar analysis is performed for other join candidate pairs and transformations, as specified in the unnesting algorithm.

The consequence of the analysis is that to allow join introduction, we have to split the predicate of the selection. The only splitting rule applicable is the range splitting rule III.9. This splitting rule, if executed, would introduce duplicate table *CLIENTS* in the expression. Therefore, before splitting is executed, we attempt to introduce products using rules III.3, III.5. However, none of the product transformations can be applied, because the quantifier is not adjacent to the selection of the *CLIENTS* collection. We therefore eventually have to split the expression using rule III.9:

$$\alpha[h : \sigma[c : (c\,.\,\text{preferred} = h)](\text{CLIENTS}) \cup \sigma[c : \exists t \in \text{THEATERS} \bullet (t\,.\,\text{manager} = c) \wedge (t\,.\,\text{city} = h\,.\,\text{city})](\text{CLIENTS})](\text{HOTELS}) \tag{3-70}$$

According to the algorithm, we return to join and nestjoin transformations to apply them as soon as possible. Join transformations still cannot be applied, but a nestjoin is possible between *HOTELS* and the first reference to the *CLIENTS* table, using rule III.6. This results in

$$\alpha[v : (v\,.\,\mathrm{cs} \cup \sigma[c : \exists t \in \mathrm{THEATERS} \bullet (t\,.\,\mathrm{manager} = c) \wedge (t\,.\,\mathrm{city} = v\,.\,\mathrm{city})](\mathrm{CLIENTS}))$$
$$](\mathrm{HOTELS}\ \Delta_{h,\,c\,;\,c\,|\,(c\,.\,\mathrm{preferred} = h)\,;\,\mathrm{cs}}\ \mathrm{CLIENTS}) \tag{3-71}$$

No further join and nestjoin transformations can be applied, and we proceed with the splitting of the conjunction using rule III.8. As described above, we have to decide which of the two conjuncts has to be placed in the selection predicate. The proposed splitting strategy chooses the conjunct which can be used for a join in the next unnesting step. In this case, conjunct $(t\,.\,\mathrm{manager} = c)$ is chosen because it references the variable *c* of the surrounding selection over *CLIENTS*. The expression after splitting is

$$\alpha[v :$$
$$v\,.\,\mathrm{cs} \cup \sigma[c : \exists t \in \sigma[z : (z\,.\,\mathrm{manager} = c)](\mathrm{THEATERS}) \bullet (t\,.\,\mathrm{city} = v\,.\,\mathrm{city})](\mathrm{CLIENTS})$$
$$](\mathrm{HOTELS}\ \Delta_{h,\,c\,;\,c\,|\,(c\,.\,\mathrm{preferred} = h)\,;\,\mathrm{cs}}\ \mathrm{CLIENTS}) \tag{3-72}$$

Because a split has been introduced, we return to the unnesting step. Following possible join transformation is the introduction of a join between *CLIENTS* and *THEATERS* using III.2:

$$\alpha[v : v\,.\,\mathrm{cs} \cup$$
$$\Gamma[u : u_{\mathrm{CLIENTS}}\,|\,(u_{\mathrm{THEATERS}}\,.\,\mathrm{city} = v\,.\,\mathrm{city})](\mathrm{CLIENTS}\ \mathrm{join}(c, t\,;\,(t\,.\,\mathrm{manager} = c))\ \mathrm{THEATERS})$$
$$](\mathrm{HOTELS}\ \Delta_{h,\,c\,;\,c\,|\,(c\,.\,\mathrm{preferred} = h)\,;\,\mathrm{cs}}\ \mathrm{CLIENTS}) \tag{3-73}$$

The introduction of the local join between *CLIENTS* and *THEATERS* allows us to nestjoin it with the top level nestjoin using rule III.6:

$$\alpha[w : w\,.\,\mathrm{cs} \cup w\,.\,\mathrm{vs}]$$
$$((\mathrm{HOTELS}\ \Delta_{h,\,c\,;\,c\,|\,(c\,.\,\mathrm{preferred} = h)\,;\,\mathrm{cs}}\ \mathrm{CLIENTS})$$
$$\Delta_{v,\,u\,;\,u_{\mathrm{CLIENTS}}\,|\,(u_{\mathrm{THEATERS}}\,.\,\mathrm{city} = v\,.\,\mathrm{city})\,;\,\mathrm{vs}}\ (\mathrm{CLIENTS}\ \mathrm{join}(c, t\,;\,(t\,.\,\mathrm{manager} = c))\ \mathrm{THEATERS})) \tag{3-74}$$

The resulting expression is then subjected to the product and nested product transformations, which, however, are not applied for the example query. The resulting expression contains all base table references at the top level and no quantifiers.

The goal of our strategy, as formulated in the beginning of the chapter, included removal of base tables from the nested levels of the expression, and replacement of nested quantifier and

collect subqueries with efficient join operands. Our strategy attempts to realize these goals by providing an ordering of transformations based on heuristic assessment of relative performance, and an unnesting algorithm, which introduces joins in the order of their benefit in the resulting expression.

The unnesting step is by far the most complex step in our overall optimization strategy. In the following section, we place unnesting within the context of other optimizations steps, in which we follow standard approaches known in literature.

3.6 Other optimization stages

Before the query can be subjected to unnesting phase, it has to be transformed during a number of other phases. These initial calculus-level phases in the optimization process have as goal the preparation of the query for subsequent unnesting. The initial optimization phases are based on standard approaches [Jar84, Bry89]:

- The input query is subjected to the **standardization phase**, which removes ambiguities introduced by the user in the input query.
- Predicates in the query are transformed into a normal form (**normalization phase**).
- The query is subjected to the actual **translation phase**, during which the query is translated into a representation containing TMa operators. Unnesting forms the element of the translation stage.

We briefly outline each of the phases, citing the representative transformation rules where applicable. The list of transformation rules used in the actual implementation is given in Appendix B.

3.6.1 Standardization

Standardization and normalization phases prepare the query for later introduction of join operators. As such, they do not introduce algebraic operators, but rather transform queries into forms which can be easily matched by the transformations within the translation phase. Transformations within the standardization phase have two goals: reduction of the repertoire of expressions (from those allowable in the query language to those optimizable by the transformations used later), and removal of ambiguities introduced by the user.

During standardization, redundant formulations in input queries are combined using the transformations such as the following one:

Transformation III.13: (existential quantifier composition)

$$\exists y \in \Gamma[x{:}\, f \mid p](X) \bullet q \equiv \exists x \in X \bullet p \wedge q[y \rightarrow f] \tag{3-75}$$

The applicability conditions for the composition transformations involve detection of nested collect expressions and collects nested within quantifier ranges.

Transformation III.13 first replaces occurrences of variable *y* within *q* by subexpression *f*. Then, a new predicate is constructed as a conjunction of *p* and "new" *q*. The actions of replacement of subexpressions by other subexpressions are quite common in transformations in our strategy. A fairly common situation in our optimization architecture is one in which the check for applicability condition establishes information which is later used by the transformation itself. We discuss the issues surrounding the implementation of transformation rules in Chapter V.

3.6.2 Normalization

In this phase of the strategy, predicates are transformed into a normal form. Traditionally, predicates have been rewritten into Prenex Normal Form (PNF), in which the quantifiers were brought to the front of the predicate, and the matrix is brought to either disjunctive or conjunctive form. In [Ste95], predicates are translated finally into Miniscope Normal Form of [Bry89]. Miniscope Normal Form is preferred over Prenex Normal Form, which brings expressions into quantifier scopes unnecessarily.

Both PNF and MNF transformations form substages within the normalization stage. They are applied one after another: predicate in PNF, potentially with expressions brought into quantifier scopes unnecessarily, is subjected to transformation to MNF, which removes those expressions out of quantifier scopes.

As an example, consider the query

$$\exists x \in \text{STUDENTS} \bullet (\exists y \in \text{LECTURE} \bullet (y \in x.\text{attends}) \wedge (x.\text{age} = 23)) \tag{3-76}$$

which is in PNF. The expression $x.\text{age} = 23$, is brought into the scope of *y* unnecessarily, therefore we can transform it to MNF:

$$\exists x \in \text{STUDENTS} \bullet ((x.\text{age} = 23) \wedge \exists y \in \text{LECTURE} \bullet (y \in x.\text{attends})) \quad (3\text{-}77)$$

The transformation used in the example is:

Transformation III.14: (descoping of conjunctive matrix)

$$\exists x \in X \bullet (p \wedge q[x]) \equiv p \wedge \exists x \in X \bullet q \quad (3\text{-}78)$$

For this particular transformation, the matrix of the predicate in PNF has to be analyzed for conjuncts which do not contain references* to variable *x*. That leads to the requirement of management of variable environments for quantifiers.

The description of initial optimization stages concludes the introduction of our optimization architecture. In the following section, we evaluate the unnesting strategy in the light of the goals we posed ourselves and other optimization frameworks known in current research.

4. EVALUATION OF THE STRATEGY

We evaluate our optimization strategy from two viewpoints. We analyze the effectiveness of the overall strategy, based on example queries. Then, we analyze the strategy in comparison with other optimizer architectures known in the literature: in this way we attempt to illustrate its benefits and shortcomings.

The primary set of queries used for evaluation of the strategy is the set of queries in Appendix C. The set is designed to test diverse variable dependencies and relationships between nested queries. The queries from the test set have already been used above to illustrate different aspects of the strategy. In the appendix, we present the results of optimization of all example queries and motivate the strategy choices made.

The other set of queries used for testing the strategy has been the Bakeoff set [Bake96]. The Bakeoff queries are designed to test various aspects of query optimizer effectiveness. Although the Bakeoff set has not been specifically designed to test the effectiveness of unnesting, it is interesting to evaluate our strategy with a number of its queries. We evaluate the operation of our optimizer for a choice of Bakeoff queries containing nested subqueries and base table references.

* This condition cannot be expressed in our notation *e[f]*, which denotes one or more occurrence of *f* in *e*.

4.1 Nested quantifiers

Two of the simple Bakeoff queries involve evaluation of a nested quantifier within the collect predicate. The following expression represents the query “find names of all hotels whose names are client names”:

$$\Gamma[h : (h \,.\, \text{name}) \mid \exists c \in \text{CLIENTS} \bullet (h \,.\, \text{name} = c \,.\, \text{name})](\text{HOTELS}) \tag{3-79}$$

Another of the Bakeoff queries finds the names of clients who paid the price for the entire TOUR:

$$\Gamma[c : (c \,.\, \text{name}) \mid \exists t \in \text{TOURS} \bullet (t \,.\, \text{price} = c \,.\, \text{bill})](\text{CLIENTS}) \tag{3-80}$$

Both of the queries represent simple examples of adjacent nesting of quantifiers. Application of our unnesting strategy yields in both cases a semijoin between the two base tables, applied as a result of transformation III.1. No splitting transformations have to be applied, because the predicates fulfill applicability conditions. The results of the unnesting process are, respectively:

$$\alpha[h : (h \,.\, \text{name})](\text{HOTELS semijoin}(h, c \,;\, (h \,.\, \text{name} = c \,.\, \text{name}))\ \text{CLIENTS}) \tag{3-81}$$

$$\alpha[c : (c \,.\, \text{name})](\text{CLIENTS semijoin}(c, t \,;\, (t \,.\, \text{price} = c \,.\, \text{bill}))\ \text{TOURS}) \tag{3-82}$$

The next query presents a more complicated case: one in which the quantifier is not adjacent to the outer collect. The query “find all hotels in Madison that are also places to visit” is expressed as follows in TMql:

$$\sigma[h : (h \,.\, \text{address} \,.\, \text{city} \,.\, \text{name} = \text{″Madison″}) \wedge \exists p \in \text{PLACES} \bullet (h \,.\, \text{name} = p \,.\, \text{name})](\text{HOTELS}) \tag{3-83}$$

The unnesting algorithm proceeds as follows:
The candidate join operands are base tables *PLACES* and *HOTELS*. However, no semijoin can be introduced, because the structure of the expression does not correspond to the one of the rule III.1. Splitting proceeds from the top level. We follow the splitting strategy for conjunctive predicates as described in Section 3.4.2. Of the two elements of conjunction, we choose the one containing a quantifier whose matrix contains references to both variables *h* and *p*. The

chosen subexpression is placed in the inner select predicate using Transformation III.8, resulting in:

$$\sigma[h : (h\,.\,\text{address}\,.\,\text{city}\,.\,\text{name} = \text{"Madison"})\ ](\sigma[h : \exists p \in \text{PLACES} \bullet (h\,.\,\text{name} = p\,.\,\text{name})](\text{HOTELS})) \tag{3-84}$$

We return to unnesting transformations. Semijoin can now be introduced using Transformation III.1:

$$\sigma[h : (h\,.\,\text{address}\,.\,\text{city}\,.\,\text{name} = \text{"Madison"})\ ](\text{HOTELS semijoin}(h, p\ ;\ (h\,.\,\text{name} = p\,.\,\text{name}))\ \text{PLACES}) \tag{3-85}$$

At this point, the nested quantifier has been removed and all base table references are at the top level. The selection predicate can now be pushed to join operand *HOTELS* using standard pushing transformations listed in Appendix B.

We have observed how the strategy copes with quantifiers nested within the collect predicates. We now turn to nesting within collect functions, using a variation of another Bakeoff example.

4.2 Nested collects

We take the following example query: “for each museum, find names of all local artists born after 1900”. We have the following query

$$\alpha[m : \langle \text{mus} = m,\ \text{l\_artists} = \Gamma[a : (a\,.\,\text{name}) \mid (a\,.\,\text{birthyear} > 1900) \wedge (a\,.\,\text{city} = m\,.\,\text{city})](\text{ARTISTS})\rangle\ ](\text{MUSEUMS}) \tag{3-86}$$

The unnesting algorithm analyses the candidate join operands *ARTISTS* and *MUSEUMS*. The only unnesting transformation which can be applied is the nestjoin transformation III.6. Its application results in

$$\alpha[v : \langle \text{mus} = v_{\text{MUSEUMS}}, \text{l\_artists} = v\,.\,\text{as}\rangle\ ](\text{MUSEUMS}\ \ \Delta_{m,\,a\ ;\ (a\,.\,\text{name})\ \mid\ (a\,.\,\text{city} = m\,.\,\text{city}) \wedge (a\,.\,\text{birthdate} > 1900)\ ;\ \text{as}}\ \ \text{ARTISTS}) \tag{3-87}$$

Thanks to nestjoin transformations, the *ARTISTS* collection can be brought to the top level.

Apart from evaluation of the optimizer effectiveness by evaluation of example nested queries, we compare our unnesting strategy with optimization frameworks known in the literature.

4.3 Comparison with existing approaches

The topic of optimization of nested queries has raised considerable interest in the research community. We compare our approach to well-known unnesting approaches and assess the benefits and shortcomings of our strategy.

4.3.1 Cluet & Moerkotte

In a series of papers [Clu92, Clu94] Cluet and Moerkotte have approached the problem of optimization of nested queries. In the paper [Clu94] a classification of various nested query forms have been presented. For each of the classes, an example has been optimized using a number of proposed transformation rules. Much attention has been paid to nested queries containing aggregate operators, which are lacking in our input language. Some of the transformation rules mentioned in this thesis have also been used in their rule set (e.g. introduction of semi- and antijoins). However, there is no precise structuring of rules, or a strategy for gradual introduction of preferable operators. Also, no attention is paid to the need of splitting of predicate expressions to allow for introduction of joins. To our knowledge, the proposed optimization strategies have not been implemented.

4.3.2 Steenhagen et al.

The optimization rule set of [Ste95] forms the basis of the unnesting and splitting rules in our thesis. The (largely notational) differences between the two rule sets have been described in Appendix A. The main difference between their work and ours is the introduction of the unnesting strategy, based on the heuristic ordering of transformations. We have also added splitting strategy elements, and special treatment for conjunctive and disjunctive predicates. The implementation of the optimizer allowed for the evaluation of the optimizer for a number of test queries. The evaluation led to modifications of the initial strategy, especially in its treatment of splitting.

4.3.3 Mitchell et al.

In the Ph.D. thesis [Mit93], an extensible architecture for the construction of query optimizers has been introduced. The EPOQ architecture allows for modeling of optimizer components as a hierarchy of goal-directed regions. While intended for all forms of optimization, the architec-

ture contains mechanisms for modeling of complex strategy elements such as seen in our architecture. It does not contain, however, ordering and structuring of transformations, and no unnesting strategy. Also, no special attention is paid to the need for splitting of expressions. The thesis contains an object algebra, and a set of rules which can be used for optimization of nested queries. The focus of EPOQ is the extensible representation for query expressions and strategy elements. Although extensibility has not been the focus of our research, the implementation techniques used in our optimizer are similar to those proposed there. We describe the similarities and differences in terms of implementation techniques in Chapter V.

4.3.4 Cherniack et al.

Optimization techniques for nested queries rely on complex applicability checks for transformation rules. The architecture proposed in [Che96] uses a variable-free representation of transformation rules: the representation allows for precise description and semi-automatic reasoning over transformations, as well as easier implementation. The focus of this work is extensibility and ease of optimizer construction, although much attention has in this case been given to the problem of proving correctness of transformation rules. Related research [Che97] focuses on formal description of optimization strategy elements: allowing for flexible rule structuring and multiple traversal options. The approach presented here seems to be the most promising in terms of future optimizer research, also in terms of formalization of complex unnesting strategies such as the one presented in this thesis.
As an example of the expressive power of the variable-free representation, a strategy for unnesting of nested queries has been formulated using COKO notation. The strategy offers a similar manner of top-down unnesting of expressions: however, due to the lack of a nestjoin operator, combinations of nest, unnest and join operators have to be introduced and manipulated. The degree to which our strategy can be expressed in COKO notation requires further research.

In this section, we have presented an evaluation of the proposed unnesting and splitting strategy. We first illustrated the benefits achieved thanks to the splitting strategy applied. We then illustrated the operation of the strategy for the Bakeoff query benchmark and shown that the

strategy allows for introduction of efficient join forms. We then compared our strategy to the approaches known in the literature.

The unnesting strategy is geared towards optimization of nested collect and quantifier queries, and removal of base table references. As such, it is only a small element of the overall query optimizer and can be improved in many ways. In the following section we describe a number of possible extensions of the strategy.

5. EXTENSIONS AND FURTHER RESEARCH

In the treatment of possible extensions, we turn first our attention to the problem of nesting of quantifiers within the **SELECT** clause. While in the current strategy this nesting form has to be solved through the introduction of the nested Cartesian product, an introduction of an additional operator can improve the situation*.

5.1 Handling quantifiers within collect functions

Join transformations in the unnesting strategy have been designed for quantifiers nested within the collect predicates (**WHERE** clause). However, quantifiers can also occur within collect functions, as in the example query

$$\alpha[x : \sigma[y : p[x, y] \wedge \exists z \in Z \bullet q[x, z]](Y)](X) \tag{3-88}$$

In the current strategy, no unnesting rules can be applied directly and the selection predicate has to be split:

$$\alpha[x : \sigma[y : \exists z \in Z \bullet q[x, z]](\sigma[y : p[x, y]](Y))](X) \tag{3-89}$$

Now, unnesting rule III.6 can be applied to introduce a nestjoin between X and Y:

$$\alpha[v : \sigma[y : \exists z \in Z \bullet q[x \rightarrow v_X, z]](v \,.\, \mathrm{ys})](X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, ys}\; Y) \tag{3-90}$$

This, however, concludes the unnesting step, and the only possible transformations in the initial strategy are product and nested product transformations, leading to

* We do not include this aspect in the unnesting strategy because it has not yet been implemented. As such, it is suitable for an “extensions” section: it shows promise, but its full impact has not yet been tested.

$$\alpha[v : \Gamma[w : w_{YS} \mid q[x \to v_X, z \to w_Z]](v \,.\, \text{ys} \times Z)](X \;\; \Delta_{x,y\,;\,y \mid p[x,y]\,;\,ys}\; Y) \tag{3-91}$$

Base table *Z* can still be removed to the top level, but the Cartesian product will remain in the map function.

The relatively inefficient optimization result is caused by a missing transformation allowing for unnesting of quantifiers within map functions. The proposed transformation rule is similar to rule III.6: however, it unnests a quantifier rather than a collect within a collect. To introduce the rule, we introduce another operator in the TMa algebra: **markjoin**.

Definition III.5:

Markjoin marks matched and unmatched left operand tuples with values true and false.

Let m be a label, then:

$$T_m = \{\langle m = \text{true}\rangle\} \qquad \text{and} \qquad F_m = \{\langle m = \text{false}\rangle\} \tag{3-92}$$

$$\begin{aligned} &X \;\; \perp_{x,y\,;\,p\,;\,m} \;\; Y = \\ &((X \;\; \text{semijoin}(x, y\,;\,p) \;\; Y) \times T_m) \cup ((X \;\; \text{antijoin}(x, y\,;\,p) \;\; Y) \times F_m) \end{aligned} \tag{3-93}$$

Intuitively, markjoin extends left operand tuples with a boolean-valued attribute *m*, value of which is determined by checking whether there exists a tuple in the right operand fulfilling condition *p*. Markjoin has been introduced in [Ste95], as a derivation of constrained outerjoin of [Bry89].

We can now introduce two new transformation rules, which allow for unnesting of existential quantifier within a collect function. The new transformation rules complement rules III.6 and III.7, as they allow for unnesting of arbitrarily nested quantifiers.

Transformation III.15: (markjoin introduction)

$$\alpha[x : f[\exists y \in Y \bullet p[x, y]]](X) \equiv \alpha[x : f[\exists y \in Y \bullet p \to x \,.\, m]](X \;\; \perp_{x,y\,;\,p[x,y]\,;\,m} \;\; Y) \tag{3-94}$$

Transformation III.16: (negated markjoin introduction)

$$\alpha[x : f[\neg\exists y \in Y \bullet p[x, y]]](X) \equiv \alpha[x : f[\neg\exists y \in Y \bullet p \to \neg x \,.\, m]](X \;\; \perp_{x,y\,;\,p[x,y]\,;\,m} \;\; Y) \tag{3-95}$$

In the new transformation rules, we replace the existential quantifier in the map function by the reference to the markjoin attribute *m*.

To integrate the new rules in the improved strategy, we have to determine the ordering of new transformations in terms of their performance along the lines of Section 3.3.3. As before, we determine the sets of alternative transformations:

1. For queries with existential quantifiers within collect functions, the alternative transformations are III.15 (introducing a markjoin) and III.7 (introducing a nested Cartesian product).
2. For queries with negated existential quantifier within collect functions, the alternative transformations are III.16 (introducing a markjoin) and III.7 (introducing a nested Cartesian product).

To determine the ordering, we use the following heuristic:

Heuristic III.6:
Markjoin has the same or better performance as the nested Cartesian product.

Assuming nested-loop processing, nested Cartesian product will traverse *R*S* tuples, while the markjoin will traverse *R*S* tuples only in the worst situation (if none of the right-hand tuples match). Also, in case of a markjoin, the result tuple will be extended with a boolean attribute, while in a nested Cartesian product it will be extended with a set-valued attribute. This will always make nested Cartesian product more expensive, as its execution will involve *R+1* temporary collection creations in comparison to *1* for the markjoin.

The heuristic analysis leads to the following ordering of transformations:

For both cases (nested existential and nested negated existential quantifier) we have to choose between markjoin application (III.15, III.16) and a nested Cartesian product (III.7). Heuristic III.6 prescribes that markjoin application will never be worse than nested product: the choice for markjoin transformations is motivated also by the fact that both of them remove the existential quantifier from the collect function. The alternative rule III.7 will leave the quantifier in the collect function, and will only replace the quantifier range with the reference to nested product attribute. We choose therefore for markjoin transformations: the orderings are **<III.15, III.7>** and **<III.16, III.7>**. The markjoin rules can be placed in the join transformation set, together with semijoin, antijoin and join introductions.

If the strategy is extended as described, the example query will be transformed differently than before. Unnesting rule III.15 will be applied directly for collections *X* and *Z*, resulting in

$$\alpha[x : \sigma[y : p[x, y] \wedge x\,.\,m](Y)](X \ \perp_{x, z\,;\, q[x, z]\,;\, m} \ Y) \tag{3-96}$$

Now, the selection predicate does not even have to be split, and nestjoin can be introduced directly:

$$\alpha[v : (v \,.\, \text{ys})]((X \;\; \perp_{x,\, z\, ;\, q[x,\, z]\, ;\, m} \;\; Z) \;\; \Delta_{x,\, y\, ;\, y \,|\, p[x,\, y] \,\wedge\, x \,.\, m\, ;\, ys} \; Y) \tag{3-97}$$

This result is much better than the one achieved without markjoin transformations: there are no expensive operator invocations in the map function.

The addition of markjoin transformations improves the optimization results for quantifiers nested within map functions. In Appendix C, we have further illustrated the benefits of markjoin processing in the example query set.

In the following section, we address other possible extensions of the strategy: handling of quantifiers on set-valued attributes, as well as additional predicate transformations.

5.2 Remaining problems and extensions

The quality of our strategy depends not only on the removal of base table references, but also on the level to which it can remove calculus-level expression forms from collect parameters. This quality depends on the splitting strategy used, as it is through splitting that expressions are prepared for removal of nested quantifiers and subqueries. In the standardization phase, we attempt to rewrite predicate expressions to forms which can be handled in the unnesting and splitting phase. This involves transformation of certain predicate forms to equivalent quantifier expressions. This allows us to work with a compact set of splitting rules. Consider the example query, which finds names of all hotels which have highly-skilled chefs:

$$\Gamma[h : (h \,.\, \text{name} \mid h \,.\, \text{chef} \in \text{GOOD\_CHEFS})](\text{HOTELS}) \tag{3-98}$$

If the query would be left as is, the only possible transformation would be introduction of the nested product between *HOTELS* and *GOOD_CHEFS*. The set membership predicate can, however be rewritten into an existential quantifier, producing:

$$\Gamma[h : (h \,.\, \text{name} \mid (\exists v \in \text{GOOD\_CHEFS} \bullet (v = h \,.\, \text{chef})))](\text{HOTELS}) \tag{3-99}$$

The transformation used is

$$x \in X \equiv \exists v \in X \bullet (v = x) \tag{3-100}$$

It is important to note that the above transformation is applied only if X is a table expression, as only in this situation it is beneficial to introduce quantifiers in the query. Predicates which cannot lead to unnesting will be left in their original form, which in many cases allows for efficient implementation in the physical algebra. The transformation allows us to transform the query to a semijoin:

$$\alpha[h : (h\ .\ name)](\text{HOTELS semijoin}(h, v\ ;\ (v = h\ .\ \text{chef}))\ \text{GOOD\_CHEFS}) \tag{3-101}$$

The example clearly illustrates the benefits of predicate transformations. We applied the predicate transformation rules in the standardization phase, so that the newly introduced quantifiers can be normalized into PNF and MNF.

Another possible problem with the strategy is the presence of correlated expressions in the query, leading to quantifiers in the result. An example is the following query, which for every museum retrieves its name and checks whether all artists displayed in this museum are born in or after 1900:

$$\alpha[m : \langle \text{m\_name} = m\ .\ \text{name}, \text{m\_artists} = \neg\exists a \in m\ .\ \text{artists} \bullet a\ .\ \text{b\_year} < 1900\rangle](\text{MUSEUMS}) \tag{3-102}$$

None of the unnesting rules in our strategy can be applied to this query, because the inner quantifier range is a set-valued attribute of the outer iteration variable: the inner quantifier and outer collect are **correlated**. This does not mean that the strategy cannot unnest set-valued attributes: an example of such a transformation is (3-91). However, the two collections in (3-91) are not correlated, which allows us to apply one of the product transformations.

The above example shows a situation in which it is impossible for our strategy to remove the quantifier from the collect parameters. However, our strategy guarantees that no base tables will occur in the (remaining) quantifier ranges. This will happen either because some of the join transformations will be executed, or because a nested Cartesian product will be introduced when everything else fails. Base tables will always be removed from parameter expressions because the nested Cartesian product transformation III.7 is applicable for arbitrarily nested expression, and because it is always applied at the end of the unnesting loop. The nested product transformation effectively removes all nested base table expressions which remained after unnesting and splitting steps.

6. CONCLUSIONS

In this chapter, we have presented the optimization framework used within the Joquer query optimizer. We first defined the goals of the strategy. Taking the rule set of [Ste95] as a starting point, we defined the set of transformation rules used for unnesting and splitting of expressions. We have proposed a structuring of unnesting transformation rules suitable for implementation: one which allows for planned application of transformations to expression parts. We based the structuring on ordering of transformations, based on performance and structural properties of alternative transformation results. We have presented a splitting strategy, which allows for preparation of expressions for join introductions: the strategy is designed to cope both with conjunctive and disjunctive predicate forms. The optimizer strategy has been implemented and tested on a variety of input queries of diverse complexity.

The flexibility in definition of all aspects of optimization strategy, needed in implementation of complex strategies, was one of the requirements, and has become one of the characteristics of the realized implementation architecture. The other requirement for the implementation concerns our goal of creation of a **distributed** and **scalable** query optimizer: one which would be able to operate efficiently in varying network environments. In the following chapter, we analyze the requirements of current and future networks to derive recommendations for implementation of query optimizers in a distributed and scalable way. Those two requirements: flexibility and distribution, form the input of the implementation architecture of the Joquer optimizer, presented in Chapter V. We evaluate the performance of the optimizer in Chapter VI.

IV. Scalable distributed optimization

1. INTRODUCTION

Modern database systems (and information systems in general) are confronted not only with increasing complexity of application domains and data models, but also with the problems of scale and variability. A proposal for query optimization architecture cannot be complete without paying attention to those problems, as every architecture will be inevitably confronted with those problems in reality. This calls for an inclusion of scalability considerations in design decisions for all information systems, and database systems in particular.

In this chapter, we present an approach to cope with scalability problems in client-server database systems through allocation of components on the client side. We derive our proposal from the environmental analysis of future usage of database applications. We identify the environments and usage patterns for those applications in the intranet and Internet setting. Consequently, we propose two component allocation approaches suitable for both the intranet and Internet situations: a centralized and a distributed approach. In case of the centralized approach, we approach the scalability problems through analytical modeling and measurement, results of which are presented in Chapter VI. Within the distributed approach, suitable for autonomous environments such as the Internet, we describe the market-based computational economy guiding the allocation process.

2. REQUIREMENTS FOR DATABASE APPLICATIONS ON INTERNET AND INTRANET

Our approach to client-server query optimization is motivated by the problems of scalability, with which current and future database systems are and will be confronted in heterogeneous network environments such as the Internet. Those environments will pose different requirements on future database systems.

We define the **intranet environment** as the one in which communication between the clients and the server takes place via the internal network within an organization. The **Internet environment** describes all situations in which communication between the clients and the server takes place outside of the internal network.

In both environments, database systems will be confronted with two possible usage patterns: the **on-line** pattern, with stringent response time requirements, and **off-line** pattern**,** in which response time requirements are not of primary importance. On-line patterns involve interactive processing of queries posed by the human users; off-line patterns occur in batch-style processing, in which data is periodically retrieved from the database by automatic tools. The usage patterns both on the intranet and Internet will involve a mixture of on-line and off-line operation. However, the usage patterns in the intranet case can be expected to be less variable (in terms of the characteristics of the queries and the sizes of query batches) than in the Internet case, due to geographical and administrative locality of client population on intranets. The two environments differ also in the aspect of **availability of performance parameters** of its participants. Those performance parameters include both host and network **state information**, such as the processing power of the client and network transfer speed, and the **workload parameters**, such as the usage pattern parameters mentioned above. In the context of the intranet, the server will be able to make assumptions as to the client and network state and workload, as these parameters can be expected to be more constant on the intranet than on the Internet. More importantly, the intranet has better means of centralized gathering and management of client and network state and workload information, either through direct reporting, or by storing configurations centrally.

In the context of the Internet, clients accessing the database server will operate a greater variety of computing platforms and configurations. Also, the database server will have no means of

timely and accurate management of state information, as it has no guarantee that the clients' reported performance parameters are not distorted and current.
The analysis of the two environments leads to the following conclusions.

2.1 Environmental analysis: conclusions

The first conclusion drawn from the analysis of the environments is that it is wrong to assume that the only goal of the system is the processing of the query in the fastest time possible. This may be true only in on-line usage patterns; in off-line patterns, response time may be of secondary importance. In many situations, and especially from the standpoint of the server, the goal of reliability of the service and scalability to changing workloads will be more important.
The second conclusion is that both on the intranet and Internet we will be confronted with varying **state parameters** (numbers of clients, clients' processing power, and network quality) and **workload parameters** (size and complexity of query batches). The variations on the Internet can be expected to be larger than on the intranet: however, more important is the difference in **availability** and **timeliness** of those parameters: possible on the intranet, lacking on the Internet.
How can the requirement of scalability be realized in both environments? One possibility calls for investments in the server infrastructure as a way of coping with variable workloads. However, those server-side-only solutions, such as use of parallel processing and hardware scaleup, are not always suitable due to prohibitive cost. Also, the system may be confronted both with very high and very low load: this makes investments on the server side less feasible.
In this thesis, we call for an alternative approach: As the main source of variability is the client population (its size and distribution), a better solution is to **include the client side in the allocation policy for the query optimization tasks**, which means distributing parts of the query optimization process to the clients, and executing them on the client side. The cornerstone of our approach to building scalable query optimizers is therefore allocation of needed computational resources both on the client and the server. Our allocation policy is dynamic: it is determined for a specific combination of parameters: concerning the environment both on the client and the server, the quality of the connection and the query processing workload. The allocation concerns not only the choice whether to distribute processing or not, but more importantly, how many of the processing components to distribute. In that sense, we use the term of **compo-**

nent allocation to hosts, rather than traditional allocation of host resources to processes. The two problems are, however, related: our approach draws therefore from the body of available resource allocation research, outlined in Chapter II.

The differences in the availability of state and workload information in the two environments lead to the two-pronged presentation of our component allocation architecture in the following section.

3. COMPONENT ALLOCATION MODELS

Resource allocation has long been recognized as a way to solve scalability problems in distributed systems. Not surprisingly, it can also be applied to the problem of allocation of components to processing hosts.

In our approach, the solution to the scalability problem can be achieved by a suitable **partition** of the query optimization process between the client and the server: one in which initial parts of the process are executed on the client, the intermediate results are transferred to the server, where the optimization process is finalized. Our goal in this chapter and Chapter VI is the identification of situations in which such a partition would be beneficial, and determination of optimal partition for a given situation.

In Chapter II, we have identified two broad approaches applied to the resource allocation problems in current research: the **centralized** (hierarchic) and **distributed** (autonomous) approach. We now remind the principles of the two approaches (already presented in Chapter II), and identify their applicability to the situations identified in our case analysis.

3.1 Centralized allocation of optimizer components

In centralized approach, the placement decisions are guided by one system-wide notion of "better performance", such as the average response time over the whole client population.

In centralized allocation model, there exists a controlling instance (host), which determines the placement of optimizer components based on performance parameters gathered from processing hosts (clients and server). The controlling instance establishes the placement by constructing a **performance model** based on the provided parameters, and solving it in such a way as to achieve the optimal value of some system-wide performance metric (e.g. response time).

Because the controlling host distributes components in real time based on the results of the performance analysis, it is important that the performance model is accurate. The quality of the estimation is based on the accuracy of the provided model parameters, which requires cooperation and coordination among hosts.

We deem this requirement to be difficult to fulfill on the Internet, where hosts are autonomous and coordination is hindered by network delays. On the intranet, however, centralized component management can be realized because of better means of control and coordination.

Assuming that a performance model of client-server query optimization is constructed (and we propose such a model in Chapter VI), how could it be applied in practice for a query optimization system on an intranet configuration? We assume a topology of one database server (which also manages the optimizer components*) and many clients of known configurations, connected through an internally-managed network. As we are considering the intranet environment with possible on-line operation, we attempt to allocate components in as little time as possible.

To reduce response time, we have to minimize the amount of message exchanges needed to determine the allocation of components; preferably, the allocation should be determined before the query is posed. In the centralized component allocation model, the allocation is determined by the controlling instance based on the state parameters of the hosts (clients and server) involved, the parameters of the network and the workload (the queries). Those parameters have to be gathered by the controlling instance to solve the performance model.

To avoid message exchanges, all state parameters for on-line operation should be based on **continuous monitoring** rather than on-demand transmission. In case of host information, a possible mechanism is the management of a **state repository** on the controlling instance, in which the information about the clients' and network state is stored. The information is in part based on configuration databases maintained in many company-wide networks, and in part gathered by monitoring of the hosts and the network. The parameters in the state repository are used as the input to the performance model, which guides the actual allocation of components. The model is continuously updated to allow for direct allocation when a client begins a connection. However, for the model to be calculated, we may require not only the host parameters,

* It is possible that the components are managed on a host distinct from the database server: we analyze such a situation in Chapter VI.

but also the workload parameters, which in the on-line situations may not be known until the query is posed. In the intranet context, we want to avoid such late determination of allocation, and the analysis of the environment points us to a possible solution.

In the intranet setting we are not only able to monitor the host and network state information, but also **workload information**. The queries posed by the clients in the past can be used to characterize the workload in a statistical way, deriving parameters such as average frequency of query submission, the number of collections accessed and average sizes of result sets. In the intranet setting, it is less likely that these parameters will change significantly in time: that's why they can be used to calculate the preferable allocation, even before the queries are actually posed.

We have described how centralized means of allocation can be used to distribute query optimizer components to the client population in an intranet setting. We continue the description of the centralized approach in Chapter VI, where we analyze the mechanisms underlying client-side distribution of components. This analysis allows us to identify factors influencing performance of the system, and situations, in which client-side distribution is a feasible solution to the scalability problems.

On first sight, also on the Internet the imbalance between the client and server would dictate the use of centralized (hierarchical) means of component allocation, in which a controlling host would determine the allocation of the query optimizer. However, we propose an alternative solution for the Internet case.

As mentioned above, the accuracy of the centralized allocation is based on the accuracy of the underlying performance model. On the Internet, the variability of the host and network parameters will be much higher than on the intranet: hosts may have different memory and process architectures, and varying protocols may be used for communication between the hosts. In this case, the hosts and the network will have to be characterized using more parameters, which will make the performance model more complicated, and more difficult to solve. The performance model in the Internet case will not only be more complicated than in the case of intranet: its accuracy problems will also be amplified; as the communication of parameters would have to be performed over the Internet itself, the parameters could be inaccurate and outdated when arriving.

In the absence of authoritative and timely means of performance measurement and control, as well as unreliable communication and increased complexity of the performance model, the centralized approach cannot establish global system-wide measures to guide allocation. It is therefore clear that the usability of hierarchic models of resource allocation on the Internet is restricted. For a possible solution, we turn to distributed allocation models: ones in which the allocation is determined with reduced transmission of state information over the network.

3.2 Distributed allocation

In Chapter II we have presented economic models of resource allocation, which form an alternative to centralized allocation models. In contrast to centralized models, economic models are specifically suited for situations in which the management of a performance model is impossible due to state exchange costs and varying goals of participating agents[*]. Rather than trying to achieve globally optimal allocations through creation of a (global) performance model, economic models attempt to achieve solutions satisfactory to all agents (a **pareto-optimal** allocation). A pareto-optimal allocation is achieved by reconciliation of (locally-preferred) allocations, which are **determined for each of the agents independently**. Thanks to the local determination of the preferred allocation, the amount of state information exchanged can be reduced. Local allocations allow also the agents to express their differing goals independently.
Once preferred allocations for the agents are determined, they have to be reconciled to achieve a global allocation. This reconciliation can be achieved either on a cooperative or competitive basis. In a price-based market system, the reconciliation is based on the mechanisms of supply/demand and price-based exchange of goods. In contrast to the centralized model, the interactions in a market are based on exchange of currency, rather than state information. Also the role of the controlling instance changes: from the management of the performance model to the administration of the market processes. In comparison with the centralized model, the allocation process is less dependent on the completeness and timeliness of model parameters, which can be a problem on the Internet.
The above considerations make economic models good candidates to guide component allocation in the Internet case. Thanks to them, we hope to reduce the increased control and coordi-

[*] In the Internet case, the server may aim at servicing as many queries with as little cost as possible, while a client may be interested in either minimizing query response time, or minimizing the usage of its local resources.

nation problem, avoid more complex performance modeling, and allow for better autonomy. Note that we do not exclude the use of economic models for the intranet case: they are a viable alternative if the problems of coordination arise also there.

It is important to note that we apply the economic allocation model to one connection (association of one client and one server) only: the same model will be constructed for all connections in the system. There are therefore only three agents in our economy: the client, the database server and the server managing the components. This makes the model relatively simple, and provides separation of connections, which prevents the components to be allocated to clients that do not want them. The only components that are allocated to the client are those which will be used to optimize the client's "own" queries: the refusal to accept components will influence only the execution of "his" queries, and not the queries posed by other clients. The allocation strategy of the server may have other goals than those of the clients. If, for example, the clients have to pay for each connection, the server will be interested in servicing as many connections as possible. Therefore, it will be interested in accepting as many connections as possible, and completing them as fast as possible. This will lead the server to accept the components when it has little connections, and refuse them if there are many connections. We stress that agents in our economy are programs, distributed to the hosts; they have to conform to the rules of the market, but they can express local preferences through different bidding strategies.

In the following section, we model the problem of component allocation as a price-based computational economy. We first introduce the principal elements of the economy: suppliers, consumers, products, and prices. Then, we propose an exchange model based on hybrid auctions.

3.2.1 Principles of query optimizer economy

The first choice which has to be made in a computational economy is the choice of the product, as this choice leads to the choice of market agents and the exchange mechanism. In contrast to some computational economies present in the literature [Fer95], we have not chosen the CPU power, and memory on the clients and the server to constitute the product of our economy. This is partly due to the heterogeneous character of the Internet, where it is difficult to establish one comparable measure of those parameters for multiple possible host platforms. Another reason for our choice is that the goal of the query processing economy as a whole is not a particular allocation of components on client and the server. The goal of the economy (in the Internet

case) is a **full** allocation (partition) of query optimizer components between the client and the server: an allocation, in which the **left** partition of the component sequence is placed on the client, and the **right** partition on the server. In this way, component ordering is maintained, and there is no component which is not allocated. Only this guarantees the execution of the query, while any host-related goal may still lead to partial allocation (e.g. when a client fails to cooperate). Full allocation of components is better achieved in a competitive setting: when the optimizer components themselves are the products, and the clients and the server are the consumers.

In the query optimizer economy, the product is the sequence of optimizer components. We define the **component broker** as the supplier in our economy. The component broker is the host which manages the components: it leads the allocation process, and once the allocation is determined, it transfers the components to the **consumers**: the client and the server. Frequently, the component broker will be collocated with the server on the same host.

The query optimization market requires an exchange mechanism, which will lead to full allocation of components as fast as possible. To that end, we have decided to base the mechanism on an **auction**, in which the two consumers compete to buy goods from the supplier. Our auction system is a hybrid Dutch-English auction system [Agor95] allowing for fast allocation of component sequences. The auctioning process is initiated by the auctioneer (the component broker), which transmits a **request for bids** to the client and the server. The client and the server calculate the set of possible component allocations which they can afford (**budget set**), choose the one they prefer, and transmit it as a **bid** to the auctioneer. In our economy, the consumers may determine their preferences independently of each other: a consumer may base its preferences on a local performance model, or it may include predictions of global parameters in consideration. The auctioneer determines the partition by analysis of submitted bids.

The basic principles of the query optimizer economy presented above need more formal and complete definition, with answers to the following questions:

- how are the initial request for bids determined?
- how does the consumer determine the relative utility of the alternative allocations?
- how are initial bids determined?
- how does the bidding process proceed?

We proceed with the exposition in the following section, introducing the market agents and their activities, adopting the notational conventions from [Gries93].

3.2.2 Optimizer economy: definitions

In an instance of an optimizer market, there are two component **consumers** and one **component broker**. One of the consumers is the client and the other is the server: they compete to obtain the components from the broker. The consumers are connected via a network. A component broker owns a sequence of components $C = \langle c_1, \ldots, c_n \rangle$. In case of the Joquer query optimizer, the sequence of components consists of the parser, the type checker, preprocessor, and logical optimizer.

The goal of the component broker is to achieve full and segmented component allocation, which we call a **partition**. Full component allocation is achieved if all components are allocated. Segmented allocation is achieved if both allocated subsequences C_c, C_s are segments of *C*, i.e. both of them appear in C as subsequences of adjacent elements. We denote the length of sequence *x* by *#x*. We denote the concatenation of sequences *x* and *y* as *x^y*. We further define a number of relationships for sequences:

$$\begin{aligned} isprefix(x, y) &\equiv \{\exists z | (x \hat{} z = y)\} \\ isseg(x, y) &\equiv \{\exists w, z | (w \hat{} x \hat{} z = y)\} \\ ispostfix(x, y) &\equiv \{\exists z | (z \hat{} x = y)\} \end{aligned} \tag{4-1}$$

Thus, an allocation C_c, C_s is a partition if $(C_c \hat{} C_s = C)$. In this case, the segment C_c is named the **left partition**, and C_s the **right partition**.

In the optimizer economy, the first action is the allocation of funds to consumers: these funds are later used in the auction to obtain components.

3.2.3 Funding and pricing

We first consider the problem of consumer funding: assignment of wealth to consumers for subsequent market activities. For the market to operate fairly, the wealth should be allocated by an agent independent of the consumers themselves, in our case the component broker. If wealth was to be allocated by the consumers themselves, they could allow themselves funds disproportionate of the component prices, leading to skewed market interactions. We assume that the client and the server have achieved funding and it is represented by the wealth parameter *w* (both consumers get equal funding as in [Fer88]).

Before market interaction can proceed, the component broker must establish the initial bidding price for the component sequence *C* and announce it on the market. Pricing can either be based on simple measures of performance such as the components' transfer size, or may involve elaborate estimations of component relative performance. A possible performance-related pricing discipline is as follows: for each of the components *i*, the component broker determines or estimates their relative processing time t_{pi} for an example query. The price vector $P = \langle p_1, \ldots, p_n \rangle$, containing prices for each of the components, is set proportionally to the processing time values t_{pi}.

The price vector is transmitted to consumers in a **request for bids (RFB)**. Besides the price information, the RFB may contain **state parameters**, which the consumers can use to determine the relative utility (a measure of consumer preference) of different component subsequences. The state parameters include the component processing times t_{pi} and their transfer sizes b_{ci}. The RFB contains also information about the workload: the size of the query batch (which in our model is known or estimated by the component broker) and the estimation of the processing requirements of the input query in relation to the query used to calculate t_{pi}. The information in RFB allows each of the consumers to calculate a local performance model of the workload, and choose the component subsequence based on its results.

Request for bids can be time-stamped at the component broker side, to enable crude estimation of network delay once an RFB arrives at the clients (if the component broker will be coupled with the server, network delay in case of the server can be discarded). The network delay can also be transferred as a parameter within the RFB itself.

Once the RFBs have been transferred to the consumers (the client and the server), the consumers proceed with the determination of their respective bids.

3.2.4 Consumer bidding

The actions performed by the consumers in the optimizer economy are the following:

1. Calculation of the budget set: the set of allocations affordable within the consumer budget.
2. Computation of utility functions (preferences) for the elements of the budget set.
3. Bid determination, based on the result of preference calculations.

We consider the actions in following paragraphs.

The budget set is the set of feasible allocations within the budget constraint [Fer95]. In the optimizer economy, the consumers allocate a percentage of their wealth to the purchase of components. This percentage constitutes the **budget** of the consumer: it allows the consumer to allocate part of its wealth for following auctions in case the first bid does not produce a partition.

The consumers compare prices of allocations (sequences of components) to their budget. Thus, the budget set is defined as the set of these allocations *x*, for which the sum of prices of all of its components is smaller than the budget of the consumer. A budget set forms the collection of possible allocations which are affordable to the consumer (we use the notation *p(x)* to denote the sum of prices of components in a subsequence *x* of component sequence *C*):

$$B = \{x \mid p(x) \leq c \times w\} \quad (1)$$

where *c* is the ratio of the wealth that the agent is willing to use in a bid.

After establishing the budget set (the allocations it can afford), the consumer chooses the most preferable allocation. A consumer determines whether an allocation *x* is more preferable than allocation *y* (denoted by $x \gg y$) by evaluating the **utility functions** for both of them (then $U(x) > U(y)$). In the literature, the utility functions reflected various preferences of a given agent: for cheaper jobs (price), faster serviceable jobs (service time), or weighted preferences for both of them. Once the preferred allocation is found, the consumer allocates bid prices to each of them: the bid price consists of the component price as in the RFB and the **markup** (the amount of currency which the consumer is prepared to pay above the component price).

Preferences allow the consumer to determine on which of the affordable allocations to bid. In our economy, we allow the preference determination to be independent for each client and server. We propose two categories of utility calculations: model-based and heuristic.

In **model-based utility determination**, the consumer uses the state information transmitted in the request for bids to construct a **local performance model** of the alternative component allocations. The local performance model can be constructed along the lines of global performance model presented in Chapter VI, to calculate the local performance metric for different affordable allocations. A local performance metric is the consumer's own performance criterion (utility function). Possible local metrics for both the client and the server include:

- The **consumer processing time** for the allocation (local service time), calculated from the

t_{pi} parameters transferred within the RFB, and the consumer's own estimation of its relative processing power in comparison with the component broker. The consumer will choose those allocations, for which the client-side processing time does not exceed a specified maximum value. An example of such preference is: "allow only allocations for which the processing time on the client machine does not exceed 20 seconds".

- The **total component transfer time** of the allocation, based on the transfer sizes of the components and measured or estimated network transfer sizes between the component broker and the consumer. This allows for formulation of preferences such as: "allow only allocations which can be transferred to the client in less than 30 seconds". Note that this criterion is independent of the query workload.
- The **part of local resources** (CPU, memory) used by the allocated components. The client may use an upper level for this metric to restrict the influence the query optimizer has on its normal operation, as in "allow only allocations which take 30% of the available CPU time".

Heuristic determination of utility can be used if it is impossible to construct a local performance model, either because state parameters are unavailable, or if they cannot be transferred with suitable performance. In this case, the client bases its preferences on non-performance criteria:

- The order of components should be retained after allocation (the logical optimizer should not be allocated to the client if the parser is allocated to the server). This translates into consumer preferences for the start and direction of allocation. The client prefers subsequence allocations starting from component *1*, while the server prefers subsequence allocations starting from component *n*.
- It is preferable to achieve partition as fast as possible. This can be achieved if the consumers bid on **long allocations**: allocations which contain as much components, as the consumers can afford. This form of bidding increases the chance that all components in the component sequence will be bid on in the first bidding round. This will allow the component broker to allocate them in the first bidding round. In our economy, the consumers choose as long allocations as possible within their model-based and budgetary constraints[*].

[*] We note that other bidding strategies are possible: the consumer may prefer to bid on **short allocations,** in which the consumer bids on small amount of components, but allocates large markups for them. Such a strategy may, however, lead to multiple bidding rounds, which we try to avoid.

In our economy, the consumers express their willingness to obtain components by adding markup to the preferred components' prices. The higher the markup, the greater the chance that the component will be allocated to the consumer. As the markup must remain within the limits of the budget, we express it as the percentage of the remaining (after component prices are subtracted) budget. The percentages decrease with the preferred direction of the allocation: a client may have the following **markup vector**: <0.5, 0.3, 0.2>, which defines the markup for the first, second and third component to be respectively 50%, 30% and 20% of the remaining budget. We use the markup to express both of the heuristic preferences described above.

The determination of the budget set and the preferred allocations leads to the choice of the allocation on which the consumer is willing to bid. As described above, in our economy the chosen allocation (**demand**) is the longest allocation conforming to the consumer's model-based and/or heuristic preferences within its budget. To formulate the bid, the consumer must then determine the bid prices on each of the components. To determine the bid prices, the consumer subtracts the chosen component prices from its budget, and distributes the remaining budget according to the markup vector.

For illustration, let's consider bid computation on the client for the component sequence $\langle c_1, c_2, c_3, c_4, c_5 \rangle$ with the following parameters (w denotes the wealth of the client, c its budget percentage, P the price vector transmitted in the RFB):

$c \times w = 15$, $P = \langle 1, 3, 3, 1, 5 \rangle$

Within the RFB, the component broker transmitted also the transfer sizes of the components, which are respectively (in bytes):

$b_{c1} = 50,000$, $b_{c2} = 60,000$, $b_{c1} = 80,000$, $b_{c1} = 90,000$, $b_{c1} = 60,000$

The transfer speed between the component broker and the client has been estimated at 10,000 bytes/second.

Each of the consumers determines the set of allocations affordable under the budget constraint. The price of the whole component sequence is 14, which allows the client to buy the whole sequence. The budget set for the client is therefore:

$B = \{\langle c_1 \rangle, \langle c_1, c_2 \rangle, \langle c_1, c_2, c_3 \rangle, \langle c_1, c_2, c_3, c_4 \rangle, \langle c_1, c_2, c_3, c_4, c_5 \rangle\}$

Once the budget set has been determined, the client chooses the preferred allocation using model-based and/or heuristic utility determination. For model-based utility determination, the client uses state information transmitted together with the RFB. Let's assume that the client

uses the total component transfer time metric as its criterion, with the limit of 30 seconds for the total transfer time of allocated components. Based on the estimated network transfer speed, the client determines that the transfer of all components would take too long for its self-imposed limit. The client also determines that the transfer of components 1-4 will be allowed under the transfer limit: it will take 28 seconds. The client therefore rules out the last element of the budget set. According to the heuristic preference for the longest allocation possible, it chooses allocation $\langle c_1, c_2, c_3, c_4 \rangle$ as the preferred allocation.

The client now has to formulate a bid for the preferred allocation by calculating bid prices for its components. As described above, the client first subtracts the component prices from its budget (reducing it from 15 to 7), and adds markups according to its markup vector. The markup vector in case of our consumer (client) is $M = \langle 0.4, 0.3, 0.2, 0.1 \rangle$, which leads to following bid determination (using the notation *price+markup* for all of its components).

$\langle 1 + 0.4 \times 7, 3 + 0.3 \times 7, 3 + 0.2 \times 7, 1 + 0.1 \times 7 \rangle$

The server determines its bid in an analogical way, taking into account its local utility function (which for example can take into account the connections already active). Both bids are now transmitted to the component broker for the determination of the final partition. We describe the partition determination in the following section.

3.2.5 Component broker: auctioning component sequences

The main function of the component broker is the process of auctioning of the component sequence. After a partition is achieved, the broker transfers the components to the consumers. We discuss both of the functions in the following paragraphs.

After sending out RFBs, the component broker awaits the arrival of bids (possibly up to a time-out period). As was mentioned above, the RFB contained the price vector $P = \langle p_1, \ldots, p_n \rangle$. The bids contain demand allocation vectors, consisting of bid prices and markups for allocations wanted by a consumer. For the client, the vector is $C = \langle c_1, \ldots, c_i \rangle$, while for the server it is $S = \langle s_j, \ldots, s_n \rangle$, where c_i and s_j denote bid prices for components i, j offered by respectively the client and the server. Bid analysis proceeds by determining the relationships between i and j, and identifying market situations corresponding to those situations:

- **deal**: $i + 1 = j$ and a partition is achieved directly
- **reluctant consumers**: $i + 1 < j$, there exist components which are unwanted:

$U = \langle p_{i+1}, \ldots, p_{j-1} \rangle$

- **eager consumers**: $i + 1 > j$, there exist components which are wanted by both consumers: $W = \langle \langle c_j, s_j \rangle, \ldots, \langle c_i, s_i \rangle \rangle$, where c_i and s_i represent bids offered by both consumers.

In case of a deal, the component broker can proceed directly with component allocation.

3.2.6 Reluctant consumers

When consumers are **reluctant**, the component broker has to proceed with the allocation of components in *U*. A broker may proceed with this task using two available strategies: either allocate the components in *U* arbitrarily, or auction them (the choice is analogical to the expensive bid protocol and the purchase order protocol in MARIPOSA [Ston96]). If the broker allocates, it may either choose arbitrary consumer, or guide itself by the markups offered by the consumers for the components already auctioned (for $k \leq i \vee k \geq j$). It may, for example, choose the consumer which provided larger markups on the components.

In the query processing economy, we want to achieve a partition as fast as possible. To achieve that, we encourage bidding on long component sequences through appropriate consumer heuristics. Even if this does not prevent the reluctant consumers situation to occur, we attempt to avoid secondary auctions by broker allocation as described above. However, there may be situations when we can afford increased network traffic for the sake of fairness of allocation. In this case, the broker will initiate a secondary auction of the components in *U*. If the broker decides to proceed with the auction of components in *U*, it decrements their price, and sends a modified RFB. Clients respond with new bids, and the process repeats itself until a deal is reached. This variation corresponds to the Dutch auction system known in the literature [Agor95].

3.2.7 Eager consumers

In case of **eager consumers** (which is a more likely situation thanks to our bid allocation discipline), *W* contains entries representing conflicting bids. Bid resolution proceeds then by determining the largest prefix and postfix with higher bids from analogically client- or server-side. We characterize largest **client prefix** using characterization of prefix and segment relations shown above in (4-1) (*b* denotes a bid pair on a single component, *x* and *y* denote bid sequences):

$$
\begin{aligned}
&isofclient(b) \equiv (b = \langle c, s\rangle) \wedge (c > s)\\
&isclientprefix(x, y) \equiv isprefix(x, y) \wedge (\forall t|((t \in x) \Rightarrow isofclient(t)))\\
&ismaxclientprefix(x, y) \equiv isclientprefix(x, y) \wedge (\neg\exists z|(isclientprefix(z, y) \wedge (\#z > \#x)))
\end{aligned}
\tag{4-2}
$$

Intuitively, largest client prefix is the largest prefix subsequence of *W* for which the client bids higher than the server. Largest server postfix is characterized analogically.

For example, if $W = \langle\langle 10, 7\rangle, \langle 12, 4\rangle, \langle 10, 11\rangle, \langle 13, 10\rangle, \langle 10, 9\rangle, \langle 10, 11\rangle\rangle$, the largest client prefix is $\langle\langle 10, 7\rangle, \langle 12, 4\rangle\rangle$, and largest server postfix is $\langle\langle 10, 11\rangle\rangle$. The segment $\langle\langle 10, 11\rangle, \langle 13, 10\rangle, \langle 10, 9\rangle\rangle$ is the conflict sequence. Similarly as in the case of reluctant consumers, the component broker can either allocate the conflict sequence directly, or begin a secondary auction. As the goal of our auction system is fast allocation of components, we choose the first option.

There can be different ways in which the component broker can allocate the conflict sequence. Naturally, the allocation should not be completely arbitrary, but rather it should draw on the information provided by the consumers in their bids. We propose the following heuristic algorithm for conflict sequence allocation:

The component broker sums the bid prices for all components in the conflict sequence for both consumers. For the conflict sequence $\langle\langle 10, 11\rangle, \langle 13, 10\rangle, \langle 10, 9\rangle\rangle$, the two bid sums s_c, s_s will be 33 and 30. Then it divides the conflict sequence between the two consumers in proportion to the bid sums. The consumer which has bid larger on the conflict sequence will receive proportionally larger part of the sequence. Assuming the length of the conflict sequence to be n_c, the number of components in the conflict sequence allocated to the client will be

$$
\left\lceil \frac{s_c}{s_c + s_s} \times n_c \right\rceil \tag{4-3}
$$

In our example, the client would receive 2 components and the server 1 remaining component. The allocation algorithm allows us to resolve the conflict in concert with the markup allocation policy of the consumers. Our algorithm guarantees that the consumers allocating high markups to the components are more likely to get the components than the ones which allocate low markups.

3.2.8 Deformations in consumer behavior

We deem the market allocation model to be more suitable for the Internet case: however, this means that the allocation policy should include mechanisms which cope with possible deformations in consumer behavior. In particular, this concerns a situation in which a consumer provides an empty bid.

We first consider empty client bids. If in this case the server bids for the whole sequence, then the whole sequence will be allocated to the server: the server has declared itself able to accept all components. However, if the server bids only on a subsequence of the component sequence, the market system should not allocate all components to the server. Such a behavior would reward client "cheating", in which clients would not bid at all, knowing that the components will eventually be allocated to the server. Instead, if such a situation occurs, the component broker aborts the auction process: the components are not allocated and the query is not processed. In this way, the failure of the client to bid harms only himself, and the client is discouraged to submit empty bids.

In our economy, the server will often be collocated with the component broker. We therefore trust it to participate in the market fairly. If the server provides an empty bid, we assume that this is caused by its current load caused by other clients*. We therefore do not abort the auction, but allow it to be allocated fully on the client, either through a secondary auction, or arbitrary allocation. In this way, the client can choose to drop the query if it does not want to commit its resources, or execute the query at increased expense of **its own** resources.

3.2.9 Partition allocation

The algorithms for eager and reluctant consumers allow the component broker to determine partition. Once this is done, the component broker allocates the components to the consumers according to the partition, and query processing can proceed. After components finish their activities on the consumers, they can be removed. In case of the server, this also means freeing the resources assigned to the connection, as they can be used by other connections.

3.3 Comparison with current approaches

Although computational economies have not been used to our knowledge for allocation of components in a client-server environment, our approach to economic component allocation can be compared to a number of existing proposals.

Contrary to many computational economies presented in the literature, we only have three agents in our economy: the client, the server, and the component broker. This restriction

* This means that the server cannot participate in query optimization due to other tasks it has to fulfill: for example because it still has to execute the optimized queries.

allowed us to avoid situations in which components would be allocated to non-related clients: a condition which is very important in the Internet environment.

The foremost difference with MARIPOSA [Ston96] lies in the fact that we do not consider distribution between multiple sites on the server side, but only partitions between one client and one server. This restriction allowed us to use a much simpler market system, which however, contains elements present also in the MARIPOSA economy (the hybrid auction system). We have chosen a slightly different notion of product, and allowed the local utilities to be guided by a local performance model. Another difference with the MARIPOSA economy is the possibility to subcontract processing between hosts, which is not present in our model. The difference with MARIPOSA lies also in the fact that their system distributes query execution, and not query optimization activities. Our component allocation model can also be applied to the activities of query execution: we do not exclude the possibility of distribution of query execution activities to the client. However, most of the query execution activities involve operations on data, which is usually stored on the database server. That's why we have restricted ourselves to the optimization activities, which we, pragmatically, deem to be more amenable to client-side processing than the retrieval of actual data from the database system. Further research is needed to determine the extent to which execution activities can be allocated to the clients, and estimate the additional network costs imposed by such allocations. Interesting possibilities may open, for example, if parts of the database are cached on the client side: then, data retrieval operations can be executed on the client.

Our auctioning system is a variation of the Hybrid auctioning system known in the literature [Agor95]. Our contribution lies in the introduction of model-based bid determination, based on the local performance model. In this way, we integrate aspects of performance modeling and computational economies in one framework.

4. CONCLUSIONS

In Chapter III, we have proposed a staged strategy for optimization of nested queries. In the strategy, our goal was not the efficiency of the optimization, but rather its effectiveness: the degree to which we were able to eliminate inefficient constructs from the query. In our opinion, any optimization strategy is incomplete without considering whether the approach taken is

scalable: whether it is efficient if faced with high variability of system parameters: number of clients, network connection quality and the query complexity. In this chapter, we propose to cope with the scalability problems by using the client processing power for partial optimization of queries.

Based on the analysis of the possible environments, we propose two mechanisms for allocation of query optimization components to the client and the server. The two mechanisms correspond to hierarchic and economic approaches to resource allocation, and are suited to the two network configurations in which queries are likely to be processed: the Internet and the intranet. In the centralized model, allocation of optimizer components is guided by an analytical performance model, which is constructed from the parameters of the system elements: the client and the server. We claim that such a model is feasible only in situations where its parameters can be gathered in a timely manner.

For the Internet, we propose an allocation discipline based on market-based reconciliation of local preferences of clients and servers. This allows us to restrict the model's complexity and make it more stable in the face of authonomy of hosts.

The main contribution of this chapter lies in the concept of partial allocation of optimizer components to the client as a way to battle scalability problems. While server-side resource allocation models have been proposed in literature, using both model-based [Hag86] and market-based models [Ston96], we are not aware of architectures which extend component allocation to the client side. Although the context is new, we apply well-known mechanisms: performance modeling [Jai90] in the intranet case, and a computational economy [Fer95] in the Internet case. In the computational economy, we had to choose the product and the currency as to exploit the unique autonomous character of the global Internet: this led us to the choice of the components as the product of the economy (in contrast to the host-related resources known in literature). Our economy contains elements of performance modeling: however, the models can be constructed locally by each of the consumers.

Due to the lack of time, we were not able to evaluate the market architecture through experiments; such experiments could be used, for example, to determine whether a fair partition is achieved in the presence of consumer behavior deformations described above. The experimental evaluation of the stability of the market system is, however, rather difficult because the sizes

of client populations for which the system is intended are difficult to emulate in experimental setting.

In Chapter VI, we present an analytical model of the component allocation problem, and validate it by experiments in an intranet setting. The model and the experiments are the continuation of centralized approach for component allocation presented in this chapter. While a sufficiently accurate performance model could be used to allocate components in the intranet setting as described above, this is not the primary reason why it was included in this thesis. The creation of the model allows us to identify the factors which influence component placement decisions, and quantify their influence. In other words, the model helps us to understand the mechanisms which come to play when component distribution is attempted. Only the understanding of the underlying principles and identification of factors affecting performance can lead to construction of systems, which deliver the performance benefits they promise.

V. Architecture for distributed query optimization

In this chapter, we present the implementation architecture of the realized Joquer query optimizer. Our architecture has two goals: it should allow for flexible implementation of complex optimization strategies such as those in Chapter III, and it should allow for distribution of its components according to the lines of Chapter IV.

We first present the requirements for the architecture, drawn from two previous chapters. We then describe the architectural aspects of the optimizer, paying special attention to its potential distribution. Lastly, we evaluate the architecture in comparison with well-known approaches and our own requirements.

1. REQUIREMENTS

On one hand, we aim at an architecture which allows for implementation of complex optimization strategies, such as those shown in Chapter III. The complexity of the query language leads to complex optimization strategies: such strategies are not easily implementable using existing query optimization frameworks, which often enforce predefined query traversal and transformation strategies. The requirement for our architecture was therefore the ability to express complex optimization strategies**.**

Considering the processing phases of nested queries, logical optimization is the one which poses the most challenges in terms of research goals: our requirement was therefore to put most accent on this phase. We applied well-known parsing and query verification methods in our research; however, we still implemented them as to smoothly integrate them within the

overall query processing. The physical optimization component and the execution engine has been developed in the scope of MAGNUM project [Mag96]. In our research, we did not treat the optimization phase only: we wanted to see this phase as a part of the overall query processing methodology. This translates itself into a requirement for integration of the query optimizer within the query processing: an integration not only in terms of suitable placement within the strategy, but also in terms of reuse of representations and algorithms.

It was not our goal to propose an architecture which allows us to cope with different rule sets and data models. We worked in the context of the object-oriented data model TM [BaBZ93] and the ADL rule set [Ste95]. We therefore did not aim at the realization of a query optimizer generator, such as Volcano or Exodus [Gra86, Gra93b]: in our architecture the transformations are compiled within the optimizer.

Although the problem of logical optimization of nested queries is independent of the fact whether the optimizer is distributed or not, the implementation of the logical optimizer has to take that fact into account. We therefore attempt to design an implementation architecture which is **easily**, **flexibly** and **efficiently** distributable:

We required an implementation platform which would make the action of the distribution of the optimizer as easy as possible: one in which the user would not be forced to perform extended installation actions to distribute components. We aimed at a platform in which the extent of the distributed components could be flexibly set; this requirement was needed to allow for resource management techniques described in Chapter IV. Finally, we wanted both the distribution of the components, as well as the transport of data between distributed components to proceed in an efficient way.

The requirements posed before the implementation architecture led to the following structuring of the architecture. We first turn to the representation of the query expression itself, to design it in the light of the requirements. Then, we turn to the implementation of other strategy elements: transformation rules and their collections, to see how they should be implemented conforming our requirements. In both cases, our implementation choices are applications of well-known solutions known in the literature. Then, we describe the distribution mechanisms, which influence the performance characteristics of the architecture, and form the primary contribution of this chapter. The distribution aspects concern the choice of the implementation

platform, the techniques used for transport of intermediate optimizer results, and the distribution of the schema information.

2. REPRESENTATION OF QUERY EXPRESSION

We have chosen for a common query representation used in all query optimization phases: the choice is motivated by current optimization architectures and distribution aspects:

In many query optimization frameworks (EXODUS and Volcano are the primary examples) both physical (cost-based) as well as logical optimizations are executed in the same framework. This calls for a common management of the query representation for the logical and physical optimization phases. Our query representation caters for the initial phases of logical optimization in the framework described in Figure 2-1 in Section II, as these phases are the focus of our architecture; we deem the representation to be easily extendable to the phase of physical optimization.

In the context of efficient distribution, we would like to reduce the size of the components transferred. Management of separate representation libraries in components could lead to their transfer for each of the components transferred. In many cases the functionalities of these libraries are quite similar: often they include various tree navigation and transformation primitives. The coupling of these functionalities in a common query representation library is therefore possible, and can bring about a decrease in the size of the components transferred.

Our query representation is illustrated in the following figure.

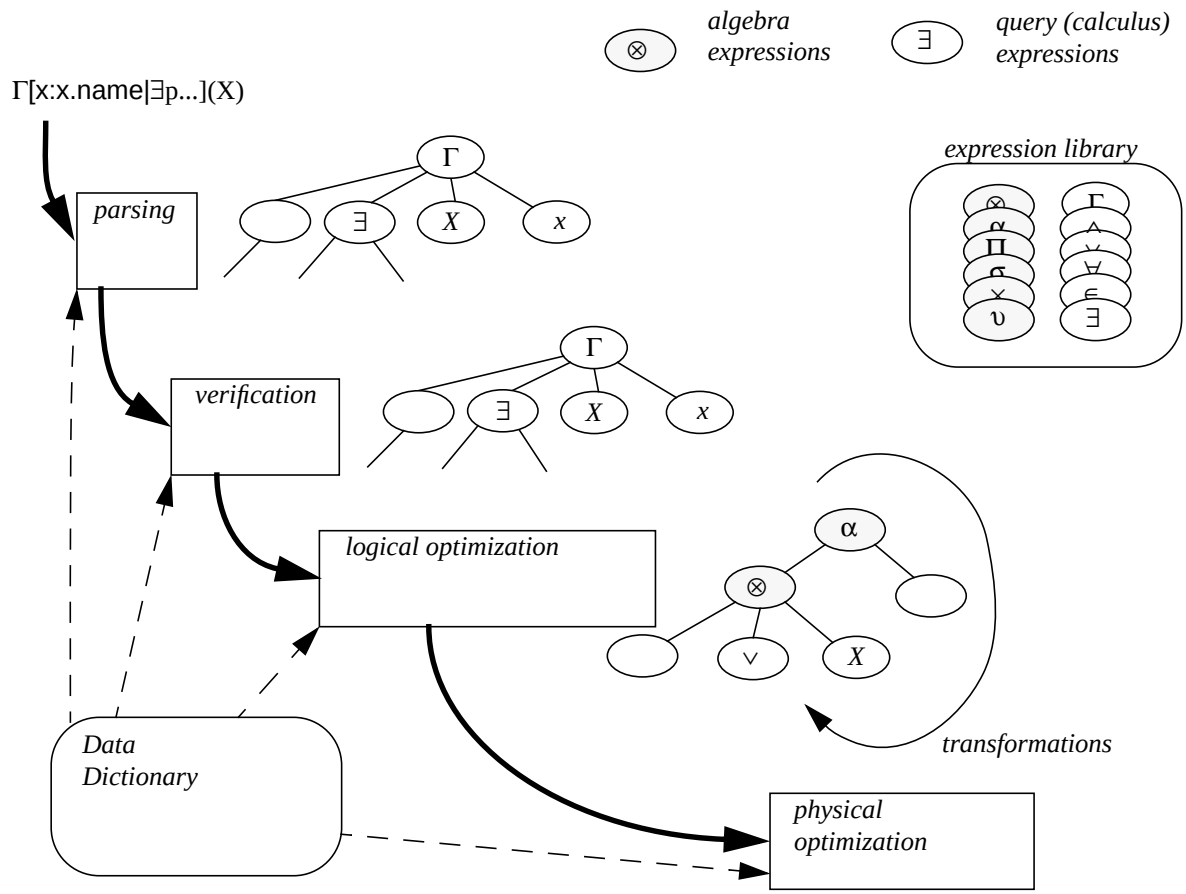

Figure 5-1 Transformations of query representation

Our common query tree is constructed during parsing from the library of expression classes, representing different elements of the query. During different processing phases, the query representation is transformed in different ways: some of these phases involve introduction of new elements in the query: algebra operators. In our representation, algebra and query language elements coexist in the same representation.

An expression tree is annotated with additional information. This information includes the link to the type of the expression (derived using the Data Dictionary) and information on variable usage. Similarly to EAT representation of [Mit93], we track variable usage in subexpressions by annotating the nodes of the expression tree with collections containing variables defined (*Def*) in the expression (and all its subexpressions), used in the expression (*Use*), and variables made available (*Avail*) to this expression from its parent expressions. Variable usage annotations are used within applicability checks of transformations rules.

The query representation is one of the building blocks of our architecture: another is the representation of the transformations (rewrite rules) and their collections (rulesets).

3. REPRESENTATION OF TRANSFORMATIONS AND RULESETS

The logical optimization strategy introduced in Chapter III requires application of transformation sets on the query. We describe the implementation aspects of single transformations and rule sets in the following sections.

3.1 Applicability checks and transformation functions

We adopt an object-oriented modeling paradigm for our architecture: transformations are represented as instances of classes structured in the transformation class hierarchy. Transformation classes implement applicability checks and actual expression rewrites as methods.
An example of a transformation is one of the unnesting transformations presented above:

Transformation V.1:

$$\Gamma[x : (g \mid q)[x, \Gamma[y : (f \mid p)[x, y]](Y)]](X) \equiv \\ \Gamma[v : (g \mid q)[x \rightarrow v_x, \Gamma[y : f \mid p](Y) \rightarrow v \,.\, ys]](X \;\; \Delta_{x,\, y\,;\, f \mid p\,;\, ys}\; Y) \tag{5-1}$$

The applicability check determines first if the parameter expression is a collect expression and X is a table expression. More importantly, it also involves the traversal of the collect function g and predicate q in search of a nested collect expression with the following properties (described already in Section 3.3.2 in Chapter III):

1. The expression Y is a table expression: a base table, a set-valued attribute, a set expression with base table/set-valued attribute operands or an algebra expression.
2. The function f and predicate p do not contain base table references
3. The predicate p consists of atomic terms only (comparisons of attribute values and constants)

We model the different constraints of this applicability check as classes: this allows us to reuse them in multiple transformations. Each of these classes may involve its own traversal of the query tree and may call other applicability checks on the encountered subexpressions.

3.2 Representation of rule sets

Transformations are elements of rulesets. Within the concept of ruleset, we define the way in which the expression is traversed, and the way in which the next transformation to apply is found. The ruleset concept is implemented as a class in our architecture, which allows us to adapt the expression traversal and pending transformation aspects in different optimization phases. Within a ruleset, we can call apply other rulesets, which lets us realize complex optimization scenarios such as the one introduced in Chapter III.

For the most part, we could base the implementation techniques used in the construction of the query optimizer on techniques used in existing research prototypes. Unfortunately, we did not have that benefit when distribution aspects are concerned, as optimization architectures have not involved client-side distribution before. Those distribution aspects form an important part of the architecture, and are described in the following section.

4. DISTRIBUTION ASPECTS

The distribution aspects of our implementation architecture result from the requirements posed: the ease of distribution, flexibility in terms of amount of components and data, and efficiency. Those requirements led us to certain choices made during the implementation. These choices concern the selection of the implementation platform for the optimizer itself and the determination of the transmission method for the (intermediate) optimized queries. As some of the query processing phases require schema information, typically stored in a data dictionary, we describe a system allowing for transparent distribution of the data dictionary to the client side. We then integrate the distribution elements in a scenario, in which we illustrate the role of different elements of the Joquer query optimizer.

4.1 Implementation platform

The requirement for the distribution of components led to the selection of Java as the implementation platform for our optimizer. We first describe the distributed computing platforms known in the literature, and describe the rationale for our choice.

4.1.1 Distributed computing platforms

Similarly to data, computational components (modules, procedures) can be distributed (remotely executed) in a **distributed computing system**. Various proposals have been suggested, allowing for partitioning of computing entities over multiple sites. Within the area of distributed operating systems, process allocation policies are used to allocate processes to processors as to balance the load between a set of shared computers. Process allocation can take place before they are executed, or during their execution, leading to the concept of **process migration**. In the Amoeba distributed operating system [Tan90, Tan92], a process descriptor can be created for an existing process; the descriptor can be used to recreate the running process at another host.

The distribution of computation can also be considered on the program and module level. In the Obliq language [Car95], procedures and methods can be freely transmitted over the network in the form of **closures** which, besides program text, contain references to objects and locations needed for its execution. Different hosts are visible on the network as execution engines: delegation of computation is realized by importing an **execution engine** and providing a closure to be executed remotely.

Process migration techniques were traditionally used on platforms which provided the same architecture on the distributed hosts. However, this assumption cannot be realized on the Internet, where architectures of hosts can be heterogeneous. In the TUI system [SmHu96], heterogeneous process migration has been realized by translation of the memory image of a program between four common architectures (m68000, SPARC, i486 and PowerPC). A program is compiled for all four architectures: once it is selected for migration, an image of the program is created and stored. On the destination host, the memory image and the suitable compiled version of the program are retrieved to create a migrated process. The shortcoming of the TUI system is its requirement for the destination platforms to be known in advance.

The heterogeneity of the computing platforms has been addressed in the design of the Java architecture [Gos94]. Java offers **program migration** (not process migration) via the notion of platform-independent bytecode. On the source host, the source program text is translated into bytecode: a program for the Java virtual machine. The destination hosts all possess a copy of the virtual machine (often implemented within a WWW browser), which allows execution of bytecode programs retrieved via the network from the source host. As Java was designed to

operate on the global Internet, mechanisms have been provided to verify the security and authenticity of the received bytecode, and restrict its influence on the receiving host architecture.

Java is not the only computing platform based on the notion of platform-independent bytecode. Within the Inferno project at Bell Labs (R&D arm of Lucent Technologies) [Infer97], similar design principles have been applied. However, in contrast to Java, Inferno is a network operating system, providing uniform access to system services through a hierarchical file system. The services, both local an remote, are accessed using the Styx communication protocol. The operations on remote resources in the Inferno file system are transformed into remote procedure calls. Applications are written in the Limbo programming language; Limbo programs are translated into a bytecode format for the internal Dis virtual machine. Limbo programs are structured into modules, which can be loaded and unloaded dynamically.

The difference between Java and Inferno lies primarily in their scope. While Java provides a small set of elements (Java programming language and bytecode definition), Inferno contains all parts of an operating system: the kernel, programming language, communication protocols, libraries, security and authentication mechanisms. Not all of those elements are included in the Java environment, but some of them can be implemented as sets of libraries on top of the basic Java toolkit. This difference makes it easier for developers to design new versions of graphic, security or network toolkits in Java. Thanks to its smaller “footprint”, Java can be deployed easier than Inferno. In terms of suitability for distributed computing, Inferno contains more of the needed mechanisms built-in, such as remote method invocation. However, most of this functionality is provided also by Java in the form of libraries external to the language platform. This difference may lead to better performance on the Inferno platform, but better extensibility on the Java platform, as libraries can be easily designed. In this way, performance efforts can be concentrated on the implementation of the Java virtual machine. The public Java bytecode format allows multiple virtual machine implementations to appear, which has led to fast improvements in performance.

After presenting the available distributed computing solutions, we evaluate their suitability for client-side query optimization.

4.1.2 Rationale for optimizer implementation platform

Of the available distributed computing platforms, the Java platform is the easiest to use: Java programs are transparently distributed over the network without the need for installation on the client side. The only requirement for execution of transmitted programs is the presence of the virtual machine implementation on the client side. As virtual machines are frequently included within the browser implementations, they are much more popular than any solutions based on migration programs such as in the TUI system. Concerning the use of development, the choice is less clear: due to the lack of suitable libraries, much of the communication structures (protocols, server programs) have to be implemented directly in Java[*]. A process migration platform such as Obliq offers a much higher level of abstraction than Java.

The TUI system displays another serious shortcoming in terms of flexibility. The migrated program must be compiled for all platforms within the distributed system. This seriously restricts its suitability in the Internet environment, in which the platforms are not known beforehand. The TUI system can, however, be used in the intranet context, where the platforms are maintained and managed centrally. Our requirement for flexibility has also another dimension: that of variable **extent** of components transferred. This translates in the need for partitioning of the query optimizer components between the client and the database server, according to one of the two allocation models described in Chapter IV. This partitioning is achieved in the Java platform by structuring of the implementation: once a decision is made that a given component is to be executed on the client, all needed classes will be retrieved by the client-side virtual machine when needed. There is no need for external migration step, as is the case in the TUI system.

We note that Java platform is a **program** migration system, and not a **process** migration platform, such as Obliq or Inferno. This means that there is no built-in way to partition query optimizers while they are running; the allocation has to be determined before the execution. This forms a certain restriction of the current implementation, although dynamic allocation can be achieved by designing customized class loaders. In certain situations, it may be beneficial to allow for reallocation of components while a query batch is processed. This would allow for the system to be even more dynamic, but may introduce additional reallocation costs.

[*] We note that this is a temporary problem, as new versions of suitable libraries appear.

The Inferno system, as described in Chapter II, is a solution for management of network resources, and as such is much more than a system for the management of client-server applications. Although it provides more built-in elements needed for the implementation of a distributed computing system, its size and complexity make it also less flexible and more difficult to deploy.

The above evaluation of platforms led us to the choice of Java as the implementation platform for the Joquer distributed query optimizer. The choice allowed us to implement a prototype which offered the needed flexibility, but required development of all elements of our architecture: the query optimizer with its rule system, and the communication infrastructure. In this second area, it provided us with the needed computation distribution primitives, but it did not supply us with built-in means to distribute data. In our architecture, data distribution concerns two aspects: the transport of (partially) optimized queries from the client to the server, and the transport of schema information from the server to the client. We consider those two aspects in the following sections.

4.2 Transport of query representation

As described above, we have bundled expression classes in a library, which allows it to be efficiently distributable. However, we also require the constructed query tree to be distributable, in situations when the intermediate optimization results have to be transferred from the client to the server. This led us to examine options for transporting query trees over the network.

One of those options is the **parsing/unparsing** solution: one in which a text representation is created (unparsed) from the in-memory representation on the sender side, sent over the network and parsed on the receiver side to re-create the in-memory representation. This option has two important drawbacks. As we have noted in Section 2, query tree in our architecture contains not only the structure of the query, but also annotations describing the variable usage and typing information. During transport, they could be included in the text representation of the query, increasing parsing/unparsing costs, or recreated on the receiver side from the query itself, which would also pose a burden on the performance. Also, for this solution to work, the query representation library must be available both on the sender and receiver side.

We have therefore chosen for a **serialization** solution: one in which we use a generic serialization library. The library provides methods for writing an arbitrary object to a stream and read-

ing objects from a stream (which can be associated with a network connection). Both of the methods are designed to cope with complex (recursive) object graphs. In the Joquer implementation, the partially optimized query object is serialized into a byte stream, which is then encapsulated in an HTTP message [Fiel97] and transmitted to the server. At the server side, the byte stream is handed over to a server side function, which calls the deserialization method to retrieve the complex query tree. We already remark here that serialization speed is an important factor in the overall system's performance: we present the results of performance measurements in Chapter VI.

Another data-related aspect of the distribution architecture concerns the need to access schema information on the client side, in the form of a client-side data dictionary.

4.3 Client-side data dictionary

As is shown on Figure 5-1, many of the query processing phases make use of the schema information contained in the Data Dictionary. In a related M.Sc. project [Cap97], we have examined the issues involved in distributing the schema information to the client side. The requirements for a client-side data dictionary system are derived from the perceived uses of a data dictionary system. Those possible uses can be partitioned in following dimensions (we do not consider client-side updates to the data dictionary):

- A data dictionary system can be subjected to (server-side) updates. In a **dynamic** client-side data dictionary system (CDDS), those updates will have to be propagated to the client side. In a **static** system, we assume that there are no changes on the server side which have to be propagated.
- As in Chapter IV, we can consider the CDDS in an **Internet** and **intranet** environment. In the context of a data dictionary system, this amounts to the choice between open networks and networks which can offer quality-of-service guarantees.

In situations when the schema information is not changing on the server side during the optimization (static CDDS), schema information can be transferred once and in full in the beginning of interaction and does not have to be updated on the clients. This is the situation with the use of a client data dictionary within the Joquer query optimizer.

In case when schema changes on the server do take place (dynamic CDDS), client-side data dictionaries can be updated in various manners, depending on the network configuration avail-

able. On reliable networks with possible quality-of-service guarantees (intranets), we propose to **broadcast** incremental updates to client-side data dictionaries. On unreliable networks with possible client failures (Internet), we propose **client-initiated** incremental or full updates. Both solutions are based on the use of **timestamps**, and operate as follows:

When the data dictionary is first transported to the client, a timestamp is created. In the intranet context, the server will maintain the timestamps of all distributed client data dictionaries. When a server-side data dictionary is updated, the comparison of the update time and the client timestamps allows the server to determine the differences between the server data dictionary and each of the client-side data dictionaries. This allows the server to broadcast only the changed elements to the clients' data dictionaries. In the Internet context, the update of the client data dictionary is client-initiated: the timestamp is managed by the client and sent to the server every time an update is requested. The server is then able to respond with only those portions of the data dictionary which have to be changed.

The incremental character of updates of client-side data dictionaries poses requirements on the protocol used to transfer the data dictionary information. The protocol used in the prototype of the client data dictionary system is the DDT protocol, which has been used in the integration of a suite of database design tools [FlKeS94, KeSk96]. In the DDT protocol, schema changes are expressed as schema modification messages.

A client-side data dictionary system such as described above can be easily integrated in our distributed query optimization architecture. However, we have to determine its allocation policy, as was the case with other optimizer components. To achieve that, we have to determine the place of a data dictionary within the overall optimizer component sequence. In principle, the data dictionary can be used by different optimizer components: the type checker will use it to assess the type consistency of the query, while the logical and physical optimizer may use it to determine cost parameters. As we consider only sequential component structures, the pragmatic solution is the transmission of the data dictionary with the first component which uses it: the type checker (query verifier). It will be then available to all subsequent components if they are allocated on the client. Allocation of non-sequential component sequences, one of which can be the data dictionary, lies outside the scope of our thesis: in Chapter VI we mention possible ways of attacking this problem.

In previous sections, we have described the elements of the distributed architecture: the techniques used for distribution of computation and data (queries and schema information). Now we are in the position to illustrate the roles of those elements by presenting the operation of the Joquer distributed query optimizer.

4.4 Operation of the Joquer query optimizer

We present the operation of the distributed query optimizer based on the following figure, which illustrates the role of different elements of the system.

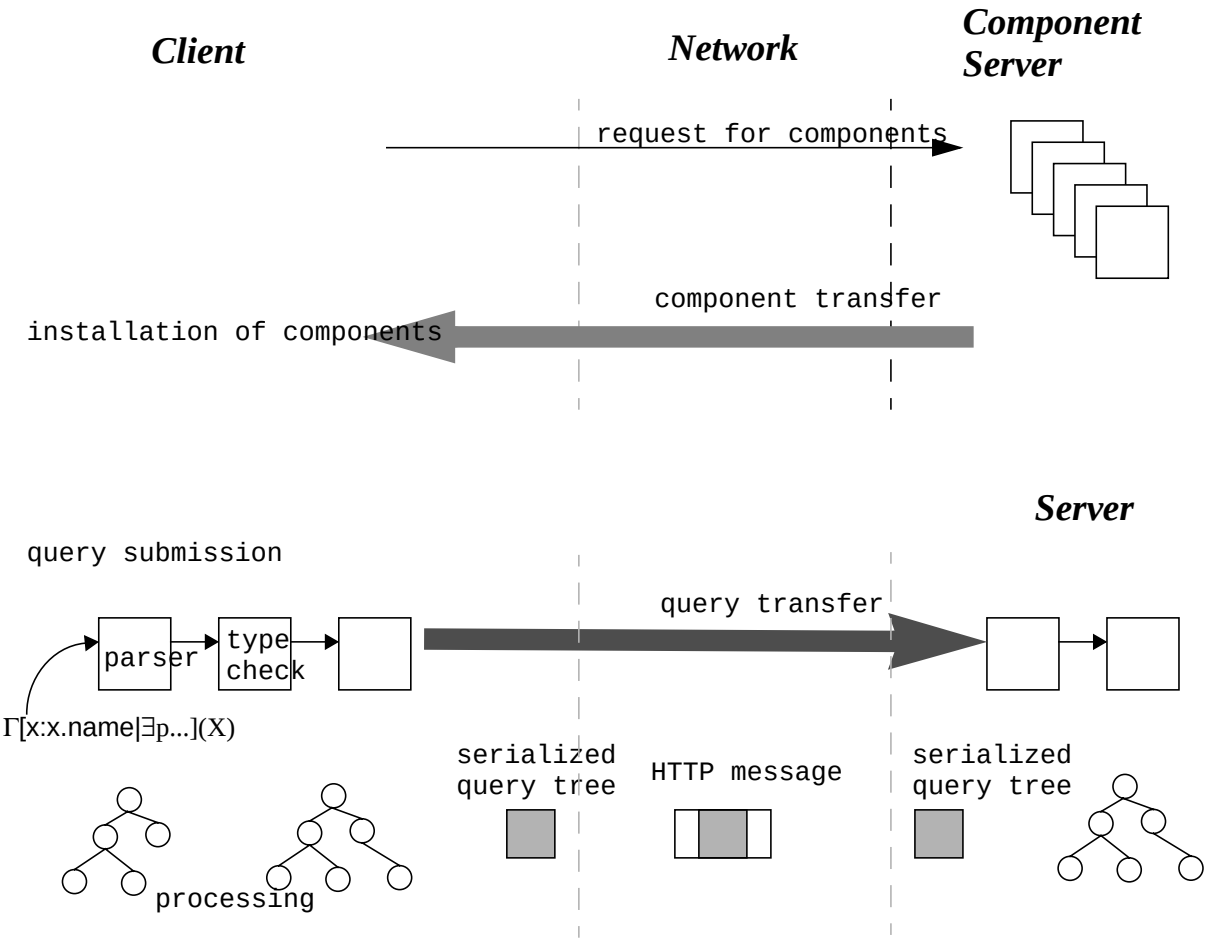

Figure 5-2 Distributed operation of the Joquer optimizer

The scenario for distributed query optimization proceeds as follows:

The client transmits a request for components to the component server. Depending on the allocation model used, the request contains client information needed by the component server to determine the partition. In case of centralized allocation, the client will transmit its host param-

eters and the parameters of the workload. The component server will then solve the analytical model for the situation, based on the parameters provided by the client and gathered from the database server. In case of economic allocation, the client will not transmit its state information, but only the workload information. The component server will respond by sending request for bids to the client and the database server, containing the prices for the component sequence and the information needed for construction of local performance model (performance characteristics of the workload and the component sequence, as described in Chapter IV). In both allocation models, component partition will then be determined: in centralized allocation by the component server through solving of the performance model, and in economic model by auctioning of the component sequence. The outcome of the allocation process will be the component partition, which is transmitted to the client. In the Joquer implementation, the agreed-upon component partition is requested by the client by formulating a HTTP message with the component partition as a parameter. The HTTP request is sent to the WWW server, which contains the components in the form of Java class files, structured in packages. The WWW server transmits the Java class files to the client. Besides the query optimizer components, the WWW server transmits also the distribution packages, used for serialization and transmission of query trees. At this point, the system is ready to accept queries.

The client may have received *0* to *n* components during allocation. In case of *0* components, the only action performed by the client is transmission of the input query string to the server. If 1 or more components are transmitted to the client, the query is transformed into a query tree, and it is this tree which is transmitted to the database server. Both query strings and query trees (**query objects**), are subjected to serialization before transmission over the network. After transmission, the serialized query object is deserialized to achieve its initial form. In this way, the problem of transmission of complex objects over the network is reduced to the problem of transmission of byte sequences. Possible options in this case include creation of dedicated network connections between the client and the database server, using a proprietary protocol to transmit the serialized query string. In our implementation, we have chosen to implement the database server as an external module (**servlet**) of a WWW server. In this way, the database server can be accessed by any WWW browser, because WWW protocols are used for the communication between the client and the database server. In the Joquer implementation, the database servlet is bound to a Uniform Resource Locator (URL) on the WWW server. The

transmission of the serialized query tree proceeds then as follows. The client formulates a HTTP **POST** request with the following parameters:

- The Uniform Resource Locator bound to the database servlet.
- The byte sequence constituting the serialized query object.
- The partition parameter.

The **POST** request is then sent to the WWW server. The WWW server identifies the URL as the one bound with the database servlet and calls it, making available the query object in the form of a stream of bytes. The database servlet uses deserialization to recreate the query object in memory on the server side, and invokes the remaining components to process it. The partition parameter is used by the database servlet to determine which components have already been invoked on the client, and which yet have to be invoked on the server. After finishing the optimization, the database server responds to the client with a status message, containing the query result.

The optimization strategies and their implementation have been the focus of a number of research projects. In the following chapter, we evaluate our strategy and its implementation in the light of those results.

5. EVALUATION OF IMPLEMENTATION

We evaluate the implementation architecture from two standpoints. We first compare our approach with other optimizer architectures and assess their differences and similarities. We then evaluate our implementation from the standpoint of requirements which we posed ourselves in the beginning of the chapter.

5.1 Comparison to existing approaches

In Chapter II, we presented existing approaches to the construction of query optimizers. Although in some cases the goals of the architectures developed were different than ours, we can place our architecture in the evolution of those systems.

We did not design an architecture for **query optimizer generation**, of which Volcano and EXODUS are prime examples. The goal of those architectures was to cater for diverse query

and data models, each of them introducing different transformations. This made it feasible to introduce a rule description language, which is used for the generation of an optimizer. The language, although allowing for a high-level definition of transformations, required applicability checks to be represented as code fragments.

In our case, most of the transformation rules were already set: we also worked in the context of the single data model TM [BaBZ93]. Furthermore, our optimization strategy is difficult to express in Volcano or EXODUS due to its mixed calculus-algebra character. Both of the systems were designed more for the integration of physical and logical optimization steps, than for the representation of calculus-level transformations with their complex variable checks. As we could observe above, the strategy requires also a high level of control on the query and transformation traversal process: Volcano and EXODUS put more accent on extending the rule set, providing the default traversal algorithms. Finally, the distribution requirements of the architecture made the application of both of the optimizer generators impossible: both of them generate C programing language code, which would have to be compiled on the client before it could be executed. This gravely restricts the ease-of-use of the optimizer if it were to be distributed.

The need for a more flexible modeling of strategy has been the focus of a number of efforts. In EPOQ [Mit93], extensibility of the strategy has been approached through the definition of optimization **regions**, each of which attempts to realize its localized goals. Many of the aspects of EPOQ can be found in our architecture: we adopted the structuring principles presented there and extended them to other phases of query processing. We also used the techniques of variable usage tracking introduced there. We did not adopt the high-level strategy description language used in EPOQ due to the lack of suitable implementation.

One of the biggest problems in the implementation of an optimizer is the realization of complex applicability checks required by the interactions of variables within the query. This formed the biggest obstacle for applying high-level rule language to calculus-level transformations. Our approach was not to use a high-level rule language, but implement flexible framework which would ease the realization of variable checks. Similar approach has been chosen in the Query Rewrite phase in the Starburst system [Pir92]. The approach allows for variation of the control scheme applied during application of transformations: as in Starburst, it allows for implementation of totally procedural and totally randomized regime.

Another promising approach has been recently introduced in COKO-KOLA [Che97], where transformations are represented using combinators in a variable-free manner. Many of rules from [Ste95] can be expressed in KOLA, which allows not only for their precise description, but also for their formalization: KOLA rules can be (semi-automatically) proved as theorems. Another interesting aspect of COKO-KOLA is its strategy modeling language, which can express many complex optimization strategies. We expect that, besides the transformations, also parts of our unnesting strategy could be expressible in COKO as well. This forms a promising area of further research. However, similarly to EPOQ, COKO-KOLA cannot be easily used in a distributed context.

All of the above-mentioned efforts aimed at a high-level description of rules and strategies. However, in our opinion, definition of rules and strategies in a high-level language has to fulfill three important properties to be useful in practice:

- it should allow to express complex rules and strategies
- it should allow to validate rules and strategies as being provably correct
- it should allow for (preferably provably) correct implementation of query optimizers

While we feel that the first and second conditions have been partly fulfilled in the above mentioned studies, we still see the implementation issues as being gravely underestimated. In Volcano, although rules can be expressed on a high level, applicability conditions still have to be added in programming language code. No correctness checks are (or can be) performed. In EPOQ, a structuring language for strategies is proposed, but again it cannot be subjected to correctness checks. The COKO-KOLA approach provides the ability to prove rules and analyze strategies. However, all of those high-level descriptions are eventually translated into programming language code. This means that even if the high-level representation is provably correct, no assumptions can be made on the correctness of its implementation, as this correctness depends on the implementation techniques used (which are usually not formally described) and the presumed correctness of programming language code (which cannot be proved for many program languages). This is a well-known problem in the area of implementation of functional languages, and has been approached in the area of program correctness [Gries81].

We do not claim that the architecture described in this thesis fulfills the three requirements mentioned above. However, our aim was more directed towards facilitating the implementation

of query processors through proper modularisation and structuring. We believe that these insights can be useful for efforts towards provably correct implementations.

After placing our architecture in the context of current research we need to examine how it fulfills our own requirements posed in the beginning of the chapter. Following the general line of the chapter, we first look at the optimization strategy itself, and then examine its distribution aspects.

5.2 Optimizer architecture evaluation

Although we did not adopt a high-level formulation for rules or strategies, we attempted to structure our implementation in such a way as to make modifications relatively painless. We use object-oriented modeling of many architecture elements: transformations, rule sets, query and transformation traversal strategies. New strategy elements can be introduced either through subclassing or method overriding, making use of considerable library of variable and expression manipulation methods. New rules can be added by adding subclasses, and redefinition of applicability and transformation functions. Applicability functions can make use of the available set of complex expression checks. New phases can be constructed by drawing transformations from the available transformation library, using new or existing query traversal strategies and ruleset traversal strategies,

5.3 Distribution aspects

In the distribution requirements posed, we stressed the ease of distribution, flexibility in distribution extent and transport efficiency. For all requirements, we have to consider the distribution of both computation (the query processor itself) and data (queries and schema information).

The implementation of the query processor itself should be easy to distribute: this led us to adoption of Java as a platform for the query processor. Although Java enabled us to distribute components without recompilation, it also forced us to implement all elements of the query processor, including a parser for the query language (using Java-based parser generator) and a rewrite rule system. However, basing the query processor on one computing platform gave us insights in its construction and allowed to develop a common query representation. Once developed and deployed, a query processor developed in Java is extremely **easy to distribute**:

all needed computation elements are retrieved form the server by the Java runtime system, and no recompilation on the client side is needed. The extent of the components to be distributed can be varied flexibly: we examine the performance issues involved with determining optimal component partition in Chapter VI. Within a component, classes can be loaded on demand, which means that only those parts of the expression and transformation classes will be loaded which are needed for the query at hand. The efficiency of the computation transport depends on the quality of the network connection and the implementation of the Java runtime system. Multiple such implementations are available and their performance is being constantly improved, for example by sending multiple classes in one network connection.

In the implementation, we considered the distribution of query processing phases involved in the optimization of the query. Although it is not included in our framework, we deem the physical optimization to be amenable to distribution as well. While it is claimed that physical optimization is much more closely integrated with the database content than logical optimization, we do not think that this fact rules out its distribution. In physical optimization, costs of different access plans are calculated based on performance parameters, and optimal plans are chosen. The physical parameters involved in the computations often involve table cardinalities and attribute selectivities. These parameters can easily be integrated in the client-side data dictionary and can therefore be available to a client-side physical optimization phase. The only problem arises when those client-side parameters become invalidated, for example when updates take place on the server side. As was discussed in the case of the data dictionary, the solution to that problem depends on the network configuration, and may involve server-initiated broadcasts in cases when network is reliable.

Considering the last phase of query processing: query execution, one would assume that this component cannot be distributed at all, because the data remains on the server side. However, in some situations even this is possible: the server may transfer a (sub)copy of the database (replica) to the client. Then, the queries posed on the client will be executed on the client, which requires distribution of query execution. We see the client-side distribution of query execution as a topic of future research.

The chosen common query representation contributed to the ease, efficiency and flexibility of distribution. Only one query representation library has to be transferred for all components, reducing the size of the transfer. The chosen serialization library allows for queries, in text

form or annotated query/operator trees of arbitrary complexity, to be transferred uniformly. The performance of the serialization algorithms in relation to query processing speed has been analyzed qualitatively and quantitatively in Chapter VI using analytical modeling and measurements.

6. SUMMARY

In this chapter, we have introduced the implementation architecture for query processing. We described the requirements for the architecture arising from the complexity of the strategy and the distributed context of our research, and we described an object-oriented modeling approach to representation of the transformations and the strategy itself. We then evaluated our architecture and strategy against current research proposals and our own requirements.

In this and previous chapters, we have described the architecture for distributed optimization of query languages. The goal of the distribution is better scalability. In the following chapter, we provide a qualitative and quantitative analysis of the resource allocation models proposed in Chapter IV, involving the distributed prototype of the query processor implemented as described above.

VI. Performance analysis of the Joquer distributed query optimizer

In Chapter IV, we have presented the distributed architecture for query processing. One of the main predicted benefits of the architecture was its ability to scale better to varying client loads. This chapter quantifies the benefits of the architecture. We develop an analytical performance model of client-server component sequences. As model validation, we perform measurements on the prototype of the Joquer query optimizer. We compare the results of the experiments to the model and we use the model to extrapolate for non-measured parameter ranges.

1. APPROACH

Intuitively speaking, the concept of the distribution of components to the clients should provide us with increased scalability and better performance in comparison to server-side-only scaleup solutions. As not only the server's, but also the clients' computational resources are available for placement of components, the pool of available resources scales gracefully with the variations in client population (both upwards and downwards). However, the overhead of client-side deployment and execution brings additional delays, which impact the performance and may render client-side distribution infeasible. The overhead includes the transfer costs of the components themselves, and the transfer and marshalling costs of inter-component communication. In real-life situations, it is important to establish the means to estimate the system parameters (in terms of number of clients, network speeds etc.) at which client-side distribution becomes beneficial.

An analytical model of the situation in question is a step in that direction. In an analytical model, we attempt to capture the performance of the system in a set of parameters and formulas. As such, those formulas are always an approximation of the real-life situation. We therefore test the accuracy of the analytical model through measurements. This allows us to identify the restrictions of the model and calibrate it as to make it more accurate for the given parameter scopes.

Even when validated with measurements, the model may lose its accuracy for parameter values outside of measured scopes. For those parameter scopes, we see therefore the model primarily as a means of gaining insight in the factors and mechanisms affecting performance. When used for determining component partitions, the model will always have to be accompanied and calibrated by measurements in the environments for which it is intended. In Chapter IV, we have identified two environments in which client-side distribution of components may be feasible. In case of the intranet, the environment parameters will be more likely to be available and accurate: in this case the partition can be determined by the calculation in the performance model. In case of the Internet, a market-based solution is more feasible, in which each of the parties (clients and server) determines the desired partition based on a local measure of utility. The final partition is then determined in an auction. This does not mean that performance modeling cannot be used on the Internet: in Chapter IV we have shown how a client and server may use a local performance model to determine the relative utility of allocation options.

In our performance analysis, we follow the following approach. We first describe the performance problem in an **intuitive, diagrammatic way** (Section 2). This allows us to illustrate the activities involved in the process of client-server query processing, and find the possible parameters which influence its performance. We then attempt to quantify our intuition by capturing it in an **analytical model**: a set of formulas which specify the relationships between the model parameters and metrics (Section 3). Model metric is the criterion used to evaluate the performance of the system and to choose between different component allocation options: in case of our model it is the **elapsed batch optimization time** (including both the deployment of the components and the optimization of the query batch).

Analytical modeling is one of the three techniques used in performance analysis [Jai90]: the other two are simulation and measurement. As we were in position to experiment with the implemented Joquer query optimizer, we chose **measurement** as the second method in our

performance analysis. The results obtained in the measurements were used to modify and calibrate the model. They provided us with realistic levels of model parameters; they also allowed us to include model parameters previously omitted into model formulas.

We present the measurement setting and results (Section 4) after the introduction of the analytical model formulas. In the presentation of the measurement results, we analyze the observed phenomena and attempt to explain them, using concepts introduced in the intuitive introduction. In this way, by observing we gain insight in the operation of the system in question. The second element of our performance analysis: the analytical model, is a way to capture this insight in mathematical formulas, which can be used by programs to assess the relative utility of alternative component allocations. We therefore compare the results obtained in the measurements to those calculated in the analytical model in Section 5. The comparison allows us both to assess the accuracy of the model, and to identify the shortcomings of our modeling and measurement methodology. Once the model is found to be accurate, it can be used to analyze the performance problem for parameter values which could not be measured: for example, for large numbers of clients found in the Internet case described in Chapter IV. This and other extrapolations of the model is found in Section 6 and Appendix E. We summarize the shortcomings of the model and the measurements, and point to areas of future research in Section 7. We finalize the chapter with the conclusions drawn from the analysis.

2. INTUITIVE DESCRIPTION OF THE PERFORMANCE PROBLEM

Before analytical model formulas are introduced, it is useful to establish the intuition behind the performance problem. In this section, we use **activity diagrams** to provide that intuition: diagrams which schematically show the influence of the parameters of the model on its metrics. The diagrams allow us to gain better understanding of the performance problem and serve as a basis for the analytical model further in the chapter.

We begin the description by introducing the participants (agents) in the client-server query optimization process. We then describe the activities in which they are involved using the activity diagrams.

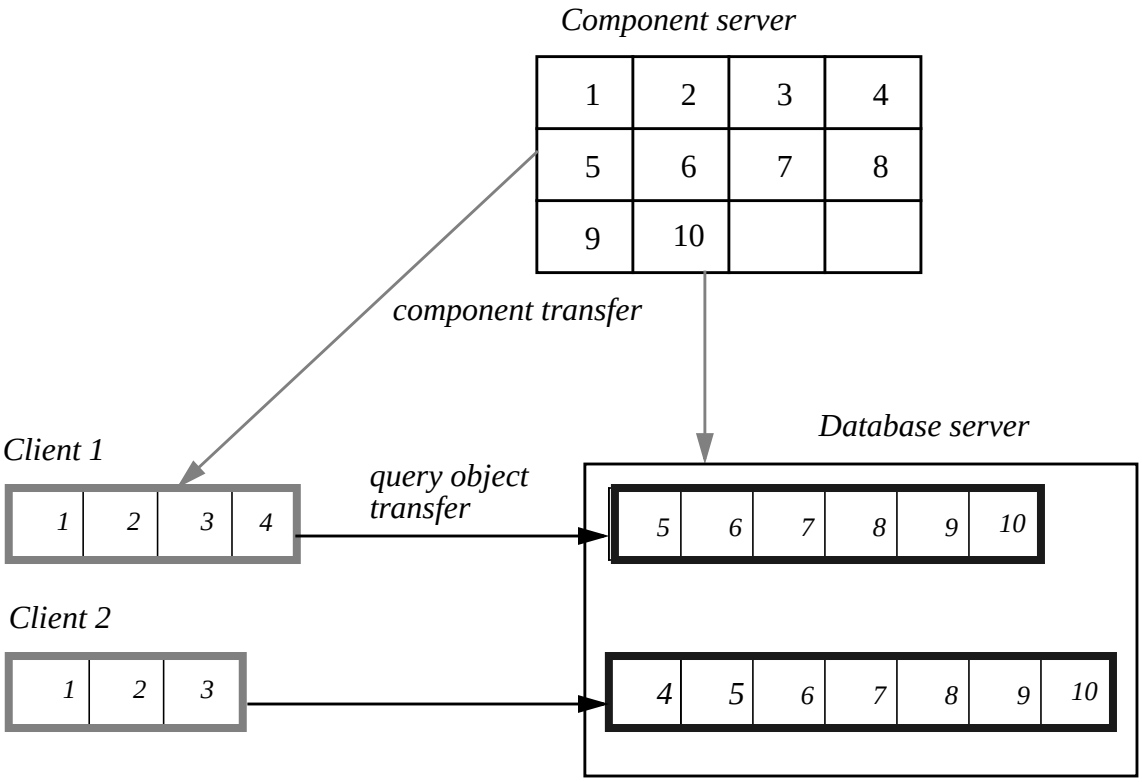

Figure 6-1 Agents in the client-server query processing architecture.

We distinguish three kinds of agents in our architecture:

- *clients*, which pose queries to the database server
- *database server*, which is the recipient of the queries
- *component server*, where components are located initially

Before query processing can take place, the component server distributes components over the client and the server. The extent to which the components are transferred to the client determines the **component partition**, which can be determined either through performance modeling, or via auctioning as described in Chapter IV. As can be seen on Figure 6-1, component partitions may differ for different clients, due to their varying processing speeds, and network connections. Once components are deployed, the queries are input by the user on the client and processed by a number of components. The result of the last client-side component (query object) is transferred over the network and processed further on the server.

On the above figure, the components are deployed to the client and the database server from a distinct component server. In many situations, the database server itself will serve as the com-

ponent server: this was the situation in case of our experimental setup. In such a situations, deployment delays from the component server to the database server do not exist. The analytical model presented below describes this special case (**collocated database and component servers**): the generalization for a distinct component server is not complicated and is presented in Appendix D.

The initial situation in our architecture is one in which no components are deployed to the client. First, the components are transferred to the client during the **deployment phase**[*], and then the queries (grouped in a query batch) are optimized during the **optimization phase**. The metric of our performance analysis is the **elapsed batch optimization time**: consisting both of deployment of components and optimization of the batch.

Both during deployment of components, and during the optimization of the query batches, the agents and the network connecting them are involved in a number of activities:

- **Processing**: the actual optimization of the query performed in a sequence of stages, taking place either on the client or the server.
- **Marshalling**: preparation of components (during deployment) and data (during optimization) for their transport over the network. This activity is performed by the sending hosts and results in serialized representation of either the query object or the software component. The receiving host obtains the in-memory representation from the transported data through the activity of **unmarshalling**.
- **Transfer** of query objects and components over the network.

During deployment, the components are marshalled on the database server, transported over the network, and unmarshalled on the client. During optimization, the query is processed and marshalled on the client, transferred over the network, and unmarshalled and processed further on the database server.

The activities of client-server query optimization form the basis of the analytical model: the model parameters influence the relative participation of each of the activities in the elapsed optimization time. This influence is expressed quantitatively in analytical model formulas provided below: however, we first attempt to express this influence intuitively using the **activity diagrams**.

[*] In intranet situations some components may be pre-allocated to the clients, and then there is no deployment

The following activity diagram represents the optimization (after components have been deployed) of one query.

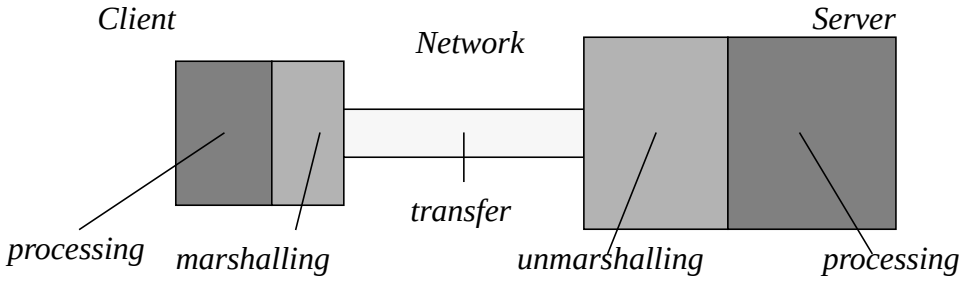

Figure 6-2 Activity diagram for client-side query optimization.

Each of the blocks on the diagrams represents one of the three activities executed during the optimization. The relationship between model parameters and model metrics is captured in the geometric relationships between the height, width and area of activity blocks*.

The **width** of the activity block is the time a given activity takes: therefore, the total width of the combined blocks on Figure 6-2 represents the elapsed optimization time of 1 query. The changes in model parameters lead to changes in block area and height, which in turn lead to changes in its width.

The **area** of every block represents the amount of work which has to be executed within every activity for given model parameters (the query batch size, network speed, client-server relationship, wrapping delays). The **height** of the block represents the capacity with which hosts and the network can execute work (process queries or transfer query objects). Both for processing and marshalling, the height of the blocks represents the processing speed of the host. For the network, the height of the block represents the data transfer rate.

It is important to note that for given system and workload parameters, each of the three activities will participate to a different level in the overall time. In many situations, one of the activi-

* This suggests that these relationships are linear. However, this is not always the case in the analytical model. The activity diagrams are therefore, as mentioned before, only an intuitive manner of description.

ties will form a **bottleneck activity**: one which forms a large percentage of the overall time in relation to the others. This is the case on the following activity diagram:

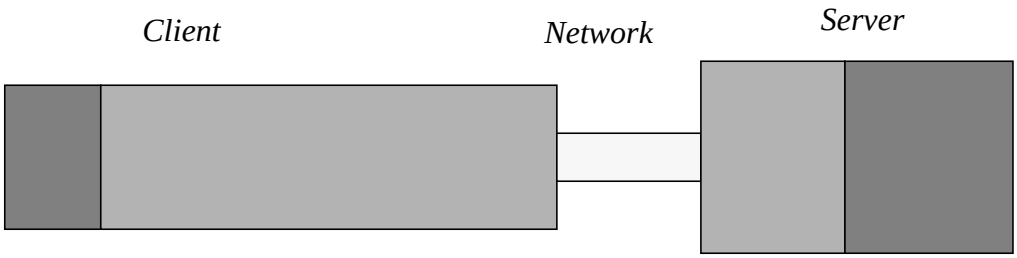

Figure 6-3 Activity diagram for client-side query optimization: marshalling as bottleneck.

Here, marshalling times form a large percentage of the overall elapsed optimization time.

The identification of bottleneck activities is paramount to performance improvement and one of the benefits of performance modeling. If bottleneck activity exists and is not identified, then changes to model parameters will be inconsequential if they do not influence this activity. On the above figure, increasing network speed will therefore have much less effect than improving the marshalling algorithm used. Only when the relative importance of the activities is established for given system and workload parameters, is it possible to influence the proper activities by changing the proper parameters.

Activity diagrams allow us to illustrate the influence of model parameters on the elapsed optimization time. We now use the diagrams to establish the intuitive reasoning behind the concept of client-side optimization. We do this by showing the negative effects of client population size (and corresponding changes in query load) on elapsed time, and how these effects can be countered through manipulation of the component partition.

2.1 Client population as a factor

The number of clients is an important parameter in our model. Its influence is biggest on the server, which for a larger client population will have to divide its resources among multiple connections. We illustrate this effect on the following figure:

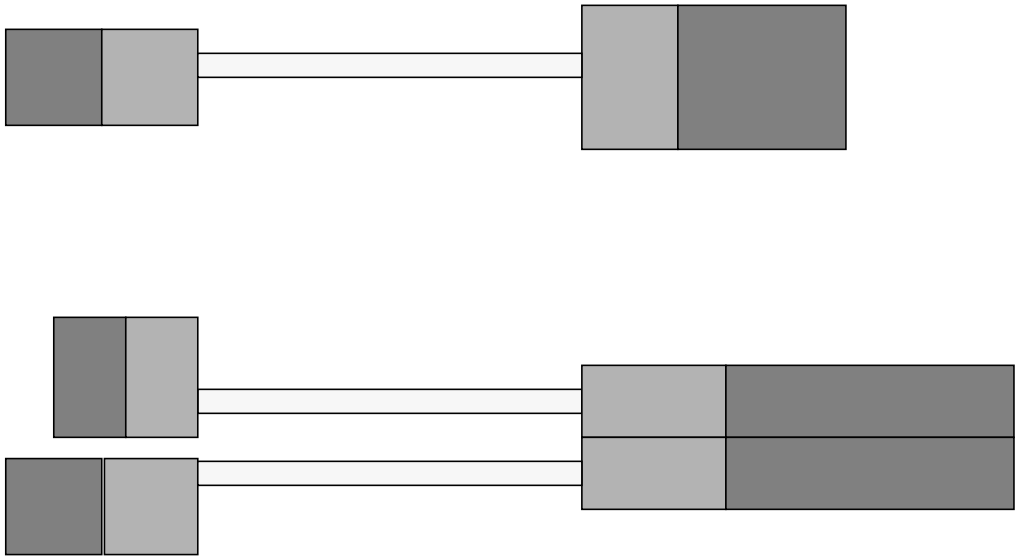

Figure 6-4 Activity model for multiple clients: no change in component partition.

In the top figure, there is only one client, and the components are divided between the server and the client. Once another (more powerful) client is added in the bottom figure, the server divides its available resources over the two connections. Each of the connections on the server side has now smaller height (less processing resources on the server side). As the amount of work needed to be executed (the area of the activity blocks) does not change, the elapsed time (the width of the blocks) increases in the lower figure.

There are a number of ways for the system to cope with the increased load without changing the distribution of components. We consider those measures in turn: first considering the improvements on the server side, then on the network and finally on the client side.

Through investment in the **server** host processing power, the total height of the server block can be increased, leading to the decreased width of the connection blocks (smaller server marshalling and processing times). Improvements in the process architecture can lead to the same result, as well as better algorithms for marshalling and processing (which will both decrease

the area of the blocks). In this way, the server will be able to cope with the increase in client population without increasing the elapsed server processing time.

The **network** transfer time can be decreased either by the use of faster network infrastructure or by decreasing the amount of transported data (the area of the network transfer block), which can be influenced by better marshalling algorithms and use of compression.

Each of the **clients** can improve the total elapsed processing time by allocating more of its resources to the optimization task: this will increase the height of the client blocks, and decrease their width.

The above solutions do not take into account the possibility of varying the amount of work that a host has to perform. This becomes possible if the amount of components transferred to the client becomes a model **factor** (model parameter varied in the performance study), a factor with which we can counter the negative effects of changes in other model parameters. The component partition is a factor which is easier to change than host processing powers and algorithms: as such, it can be set differently for each of the clients, depending on the dynamic relationships of model parameters.

Assuming that component partition is a factor in our model, we use the activity diagrams to describe their influence on model metrics.

2.2 Component partition as a factor

In the activity diagrams, the component partition influences the activity block areas, as it changes the amount of work which has to be executed: some components are removed from the server and placed on the client. On the figure below, we reproduce the bottom drawing of Figure 6-4 and add a diagram for an alternative component partition.

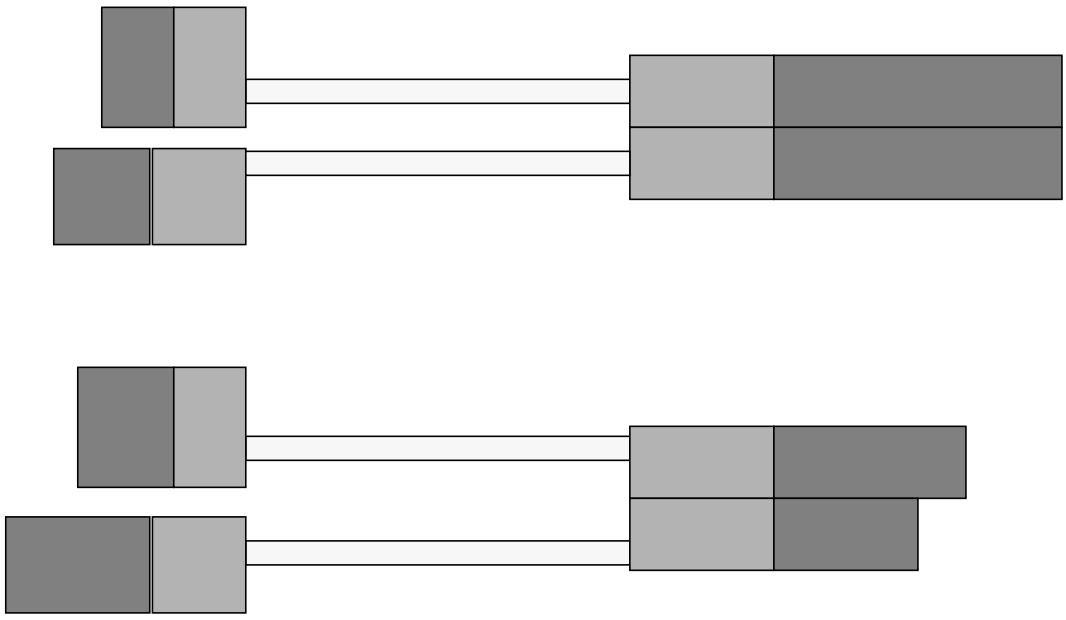

Figure 6-5 Activity model for multiple clients: change in component partition.

In this situation, some part of the server-side processing activities has been transferred to the clients (as the clients were not equal, each of them could accept varying additional load). This led to two effects: less work to be executed on the server (decrease in the area of server activity blocks) leading to decreased server processing time (decrease in width, varying per connection[*]), and more work to be executed on the client (increase in the area on the client), leading to increased client processing time (increase in width). The benefit of client-side optimization lies in the fact that the **increase in the client-side elapsed time can be smaller than the decrease in the server-side elapsed time**, leading to overall decreased elapsed time.

We note that we consider here only the optimization of the query: similar considerations can be applied to the deployment phase, and both considerations contribute to the metric of our model: the elapsed time to deploy and optimize a batch of queries. In fact, the first critique which can be posed to our approach is the influence of component partition on deployment time. If more components are to be transferred to the client, this will increase the component deployment time. Even if optimization time would decrease, the increased deployment time

[*] As each of the clients accepted different number of components, different numbers of components have to be processed on the server. This leads to the varying widths of processing blocks on the server.

may still lead to increased elapsed time (as it consists both of deployment and optimization). However, this observation does not take into account the fact that we optimize **batches of queries**, and that deployment only has to take place for the first query in the batch. The deployed components reside on the client for the duration of the query batch processing. For large query batches, the deployment time will have relatively little influence on the elapsed batch optimization time: even if the deployment time increases, the decreased optimization time for each of the batch queries will make the elapsed batch optimization time smaller. The query batch size can be expected to have large influence on the elapsed batch optimization time. The nature of this influence can be expected to be near to linear, as this is the way in which it impacts the model activities: the network transfer time, the processing, marshalling and unmarshalling costs. Client population size, on the other hand, influences the activities in much more complex manner.

After illustrating the possible effects of client-side component distribution on the system performance using the activity diagrams, we move to a more formal treatment: the introduction of the analytical model. The analytical model and the measurements have the following goal:

> ***To assess the ability of client-side optimization to counter the negative effects of "environment" parameters (such as the number of clients) on the model metric: the elapsed time to deploy components and to optimize a query batch.***

3. ANALYTICAL MODEL

We now proceed with the introduction of the analytical model; one in which the component server and the database server are collocated at the same host.

3.1 Terminology

We follow the performance modeling terminology of [Jai90]. We define the measured **system** as the client-server query optimizer, consisting of 4 components: the parser, syntax checker, preprocessor and logical optimizer. We define the **workload** as the requests (queries) made by the users of the system. Both the system and the workload have characteristics that affect the performance of the system. Those characteristics are called **parameters**. The server host processing speed is an example of system parameter, while the number of users is an example of a

workload parameter. The parameters which are varied in the performance analysis are called **factors** (e.g. number of clients). A **metric** is a criterion used to evaluate the performance of the system. An example metric is the deployment and processing time for a batch of queries.

We define the **connection** as an association of 1 client, 1 server and the network between them. Thus, alternatively to saying that *c* clients are connected to the database server, we can say that there are *c* connections active at a given moment. The qualities of network connections between the client and the server are represented as capacity parameters: number of bytes (octets) transferred per unit of time. For the network between the client and the server, the parameter is C_n.

In our model, the **query optimizer** is a sequence of *n* communicating components. Initially, all components are located on the server. When a client intends to pose queries to the server, the server determines which components are to be placed on the client, and transfers them. The sequence is partitioned between the client and the server: components 1 up to *k* are allocated to the client, while the components *k+1* up to *n* are allocated to the server. Allocation of all components to the client or server are represented by respectively *k=0* and *k=n*. The parameter *k* is the **component partition**: it is this parameter which is used by our optimization architecture to minimize the elapsed processing time. The *k* components are transferred to the client: this constitutes the deployment phase of the modeled activity.

During actual optimization, the client receives a sequence of queries (**query batch**) from the user. The queries in the batch, in our model consisting of *q* identical queries, are optimized by *k* components on the client and transferred to the server, where the optimization proceeds in the remaining *n-k* components. We note that it can be a query object and not the query string which is transferred: a string query posed initially to component 1 (parser) may be transformed into an in-memory tree-like structure, and it is this structure which is transferred to the server.

Each of the components *i=1,...,n* is characterized by the processing time t_{pi} needed for optimization of the query at the server within this component. The processing time represents the situation in which there is only 1 connection: it does not take into account any effects of multiple simultaneous connections. The component is also characterized by its size b_{ci} as well as the amount of data b_k (the size of the query object) it needs to transfer over the network, if the component is the last component on the client. In case when all components are on the server, the query object transferred has size b_0.

A client is characterized by factor f_c, which describes its relative power in comparison to the server. Therefore, the processing time for component i on the client is t_{pi}/f_c.
We summarize the model parameters in the following table

Table 6-1: Model parameters

Parameter	Description
c	Size of client population
n	Number of components in component sequence
k	Component partition: number of components located on the client ($k = 0, \ldots, n$)
q	Number of queries in query batch
C_n	Network transfer speed
t_{pi}	Processing time for component i on the server ($i = 1, \ldots, n$)
b_{ci}	Transfer size of component i
b_k	Transfer size of query object after component k ($k = 1, \ldots, n$). b_0 is the size of the query object before component 1.
f_c	Client processing power ratio in relation to the server.
t_{wc}	Time to (un)marshal 1 byte of component
t_{wq}	Time to (un)marshal 1 byte of query object

3.2 The model formulas

We first consider the optimization of queries, after components have been transferred to hosts. The total optimization time on the client is the sum of processing times for all components located on the client

$$\frac{1}{f_c} \sum_{i=1}^{k} t_{pi} \tag{6-2}$$

In case of the server, we also have to take into account the effects of multiple connections being active at the same time. In our model, we assume total separation between server-side connection components: the server does not assume or know that some components have already been loaded in other connections. The server essentially “divides” its processing power between all connections. The server processing power available to a connection will therefore decrease once more connections are active. We use a simple modeling of multiple server con-

nections by relating the effective server processing time to the number of clients raised to the power *r*. The component processing time on the server with *c* active connections is therefore

$$t_{pi}c^{r} \qquad r \geq 1 \tag{6-3}$$

For *r=1*, the processing times are increasing linearly with the number of connections; any larger *r* values represent the increasing effects of connection management. The processing time on the server is the sum of component processing times for all components located on the server

$$c^{r} \sum_{i=k+1}^{n} t_{pi} \tag{6-4}$$

During deployment, the components have to be marshalled on the server side, transferred over the network to the clients, and unmarshalled on the clients. During optimization, the query object has to be marshalled on the client, transferred over the network and unmarshalled on the server. We characterize marshalling and transfer activities for query objects and components by parameters ξ_q, and ξ_c, respectively (all defined as time for marshalling and transfer of 1 byte). Time ξ_q represents the marshalling of query objects on the client, their transfer and unmarshalling on the server. Time ξ_c represents marshalling of components on the server, their transfer from the server to the client, and their unmarshalling on the client. Similarly to processing time, we represent the marshalling activities by the parameters t_{wc} and t_{wq}, which denote the times spent on the server while marshalling (or unmarshalling) one byte of component or query object. These parameters, which can be obtained via measurements, allow us to formulate the times ξ_q and ξ_c:

$$\begin{aligned} \xi_q &= t_{wq}\left(\frac{1}{f_c} + c^{r}\right) + \frac{1}{C_n} \\ \xi_c &= t_{wc}\left(c^{r} + \frac{1}{f_c}\right) + \frac{1}{C_n} \end{aligned} \tag{6-5}$$

In the top formula, the network transfer time $1/C_n$ is added to the client marshalling of the query object t_{wq}/f_c and server-side unmarshalling $t_{wq}c^{r}$.

The metric in our model is the (elapsed) batch optimization time *T* (including the deployment of components and optimization of the batch) for one connection in a given client population *c*.

We split the batch optimization time into the deployment time T_d and optimization time for one query T_o. Then, we have:

$$T = T_d + qT_o \tag{6-6}$$

The deployment time T_d can be expressed as the sum of the component sizes times the marshalling costs per byte:

$$T_d = \xi_c \sum_{i=1}^{k} b_{ci} \tag{6-7}$$

The optimization time T_o consists of an optimization in k components on the client, transfer of the query object of size b_k and further optimization on the server (6-4).

$$T_o = \frac{1}{f_c} \sum_{i=1}^{k} t_{pi} + b_k \xi_q + c^r \sum_{i=k+1}^{n} t_{pi} \tag{6-8}$$

The final formula for the time T is obtained by substituting (6-7) and (6-8) in (6-6):

$$T = \xi_c \sum_{i=1}^{k} b_{ci} + q\left(\frac{1}{f_c} \sum_{i=1}^{k} t_{pi} + c^r \sum_{i=k+1}^{n} t_{pi} + b_k \xi_q\right) \tag{6-9}$$

In our performance analysis, the **parameters** of the model are the sizes of components (b_{ci}), sizes of inter-component communication (b_k), relationship between host speeds (f_c), network transfer speeds, the marshalling times for components and data (t_{wc}, t_{wq}) and the number of components (n). In the measurements, we examine the influence of factors on the **metric**. The **factors** in our measurements are: the number of clients (c), the number of queries in query batch (q) and the component partition (k). We use the model to determine the optimal component partition k for a given client population and query batch size.

This concludes the description of the analytical model. We now turn to the second element of our performance analysis: the measurements.

4. MEASUREMENTS

To estimate the feasibility of the analytical model as a means of determining component partitions, we have executed a number of performance measurements on the Joquer distributed

query optimizer. The measurements have been performed in the intranet setting: one in which a relatively small number of homogeneous clients connects to the server via a fast network. During the test, we have chosen the component partition, the number of clients and the query batch size as factors: parameters which are varied during the test. We have measured the elapsed batch optimization time, as well as processing times on the client and the server, and the sizes of query objects transferred. We present the measurement setting, followed by an analysis of the measurement results and their comparison with the model.

4.1 Measurement setting

The tests were executed on a network of up to 8 identical clients (Sun SparcStation 5 with 32 Mbytes) and 1 server (Sun UltraSparc with 64 Mbytes). To minimize outside influences, the optimizer was the only user job running during the tests. On the server, the optimizer ran as an external module (servlet) of a WWW server [JaSe97]. Both the component server and the database server were located on the same host. The clients were connected to the server within an Ethernet network.

Before components can be deployed on the client, they have to be fetched by the server from the disk storage. The components of the query optimizer have been placed on the local disk of the server to avoid network file system delays during the deployment of the components.

The optimizer consists of 4 components: parser, syntax checker, preprocessor and logical optimizer. The workload used during the experiments consisted of a batch of queries of varying size (1, 10 or 30 queries).

The query posed was a simple query from the bakeoff optimizer benchmark [Bake96]. Our metric: elapsed batch optimization time, is defined as the time between the first query was posed on the client (which then does not have any components yet) and the time the response to the last query (a confirmation of a successful optimization) was received on the client. When no components reside on the client, query objects are strings, while in case when some processing is done on the client, complex query trees are transferred. The sizes of query objects transferred over the network have been found to be very small for query strings (150 bytes) compared to query tree objects (~6K bytes). This is consistent with the fact that after the query is parsed, the complex annotated query tree has to be marshalled and transferred over the network. The communication between the client and the server has been realized using object

serialization libraries: the query object (string or tree) on the client side is serialized into a byte stream, which is transported over the network, and deserialized on the server side, recreating the query tree. The use of serialization made clear that the marshalling costs differ for the query objects and for components. This led to the modification of the initial version of the analytical model, in which both the components and the query objects were marshalled at the same rate.

We performed two series of tests, each of the series consisting of 195 measurements:

- 15 measurements for the 1 client case: a measurement for each of the 3 query batch sizes (1, 10 or 30 queries) and 5 component partitions (0, 1, 2, 3 or 4 components on the client)
- 60 (4*15) measurements for the 4 client case
- 120 (8*15) measurements for the 8 client case

We now proceed with the exposition of the results of the measurements. We first analyze the distribution of processing times within the batch. We also examine the relationship between client-side processing, network and server-side processing for 1 query. Then, we show the measured relationship of the elapsed batch processing time to the size of the batch. We then turn to another factor: client population size and show its relationship to the metric. Throughout the presentation, we compare the measurement results with the intuition established in Section 2.

4.2 Measurement: time differences within the query batch

We observed that the first run of the optimizer is much more costly than the subsequent ones, due to the fact that the optimizer components have to be deployed to the client. This was reflected in all client-side measurements, as illustrated in the following example of the C8CO4Q30 case (8 clients, 4 components on the client, 30 queries in a batch).

Horizontal axis represents the subsequent queries in a batch, vertical axis represents time in milliseconds, Series 1 represents client optimization time (including deployment of components for the first query), Series 2 server optimization time (including unmarshalling of results) and Series 3 the remaining transport time (including both the network transport time, as marshalling of optimization results on the client side).

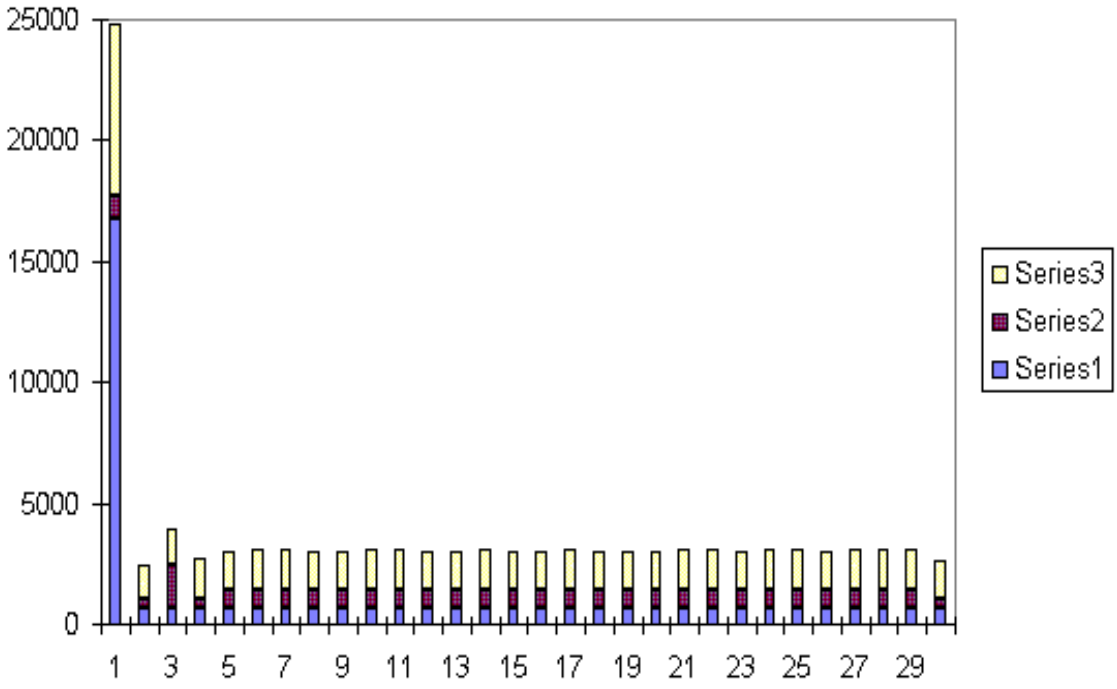

Figure 6-6 Measurement: client, server and transport time throughout the execution of a batch

For all situations, the optimization of the first query includes deployment of optimizer components to the client, which leads to much higher optimization time. This is consistent with the intuition. For query 1, the system transfers the classes constituting the optimizer on demand once they are used by the client: that's why the majority of elapsed time is registered as client time (obviously, this is not only client processing time: the client sends a request for a class to the server, which finds it and sends it back to the client). Once the optimizer is deployed (for the remaining 29 queries in the batch), the client-side optimization is executed in almost constant time. We observe also a relatively stable network transfer time (Series 3), even for 8 clients, the largest client population measured.

After analyzing the distribution of processing times within the query batch, we now turn to the distribution of processing times between the activities in our model: processing, marshalling and transfer. This can be observed by analyzing 1 query and the divisions between client-side processing, network transfer and server-side processing. We also analyze the influence of model factors (client population size and component partition) on the distribution.

4.3 Measurement: elements of query processing

We illustrate the relative costs of query processing activities in the following 2 diagrams: for populations of 4 and 8 clients. X axis represents different component partitions, from no components on the client (0comp) to all components on the client (4comp). The Y axis represents the average elapsed processing time for 1 query. It is obtained by averaging processing times for all queries in a batch besides the first (which contains deployment)

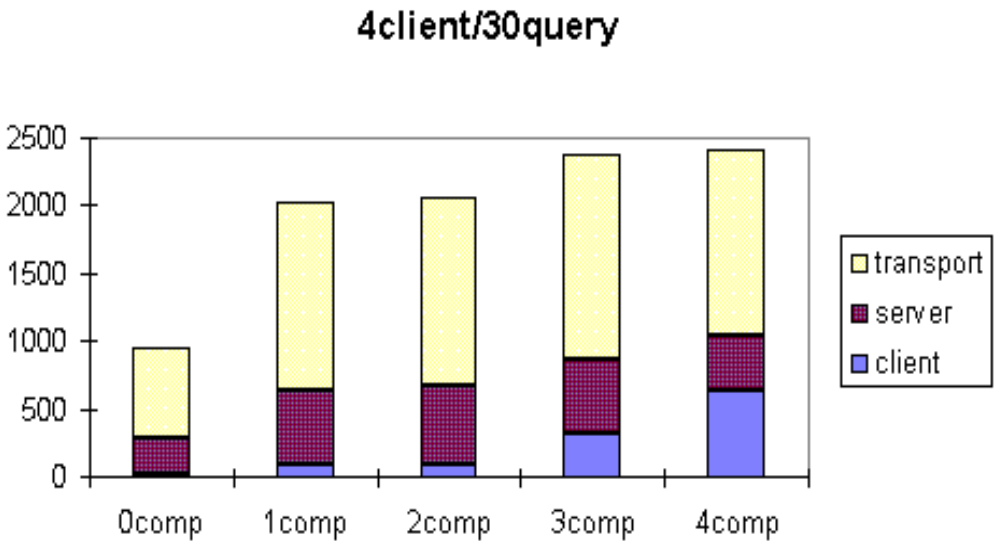

Figure 6-7 Measurement: partition of average optimization time for 1 query: 4 clients.

The first thing to observe is the growth of the query processing time (the total height of the 3 blocks): allocating all components on the server (0comp) is the best situation. We also note that there is no significant decrease in server costs when more components are transferred to the client. It is even the case that the server has more to do when some components are transferred (1comp to 4comp) than when no components are transferred (0comp). This result, although counter-intuitive, can be explained as follows: in case when there are 0 components on the client, the client transfers query strings to the server. For all other situations, the client transfers query trees to the server. The above result shows that the costs of unmarshalling a query tree are much higher than the costs of unmarshalling a query string (~120 ms for query string and ~1200 ms for query objects). By transferring components we off-load processing activities from the server, but increase the server-side unmarshalling costs. The identification of this phe-

nomenon led to introduction of separate marshalling speeds for query strings and query objects in the analytical model.

We now observe the effect of increased client population.

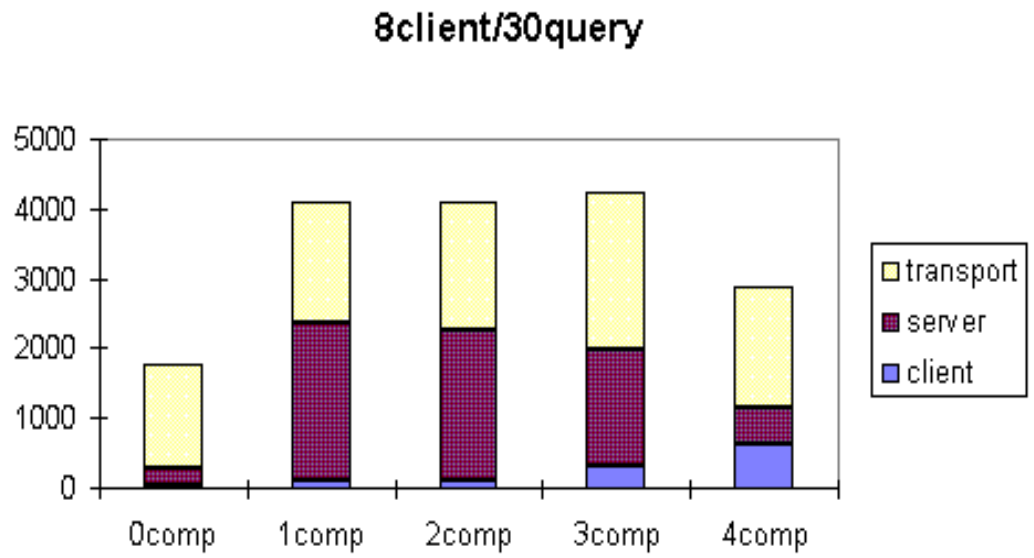

Figure 6-8 Measurement: partition of average optimization time for 1 query: 8 clients.

The client-side processing time displays similar distribution as in case of 4 clients. In case of 8 clients, the level of server time is significantly higher than with 4 clients: the server has to put additional effort managing multiple connections and scheduling the various optimization tasks. As was shown on the activity diagrams, the server divides its resources to the connections, leading to higher server-side processing times.

It is important to note that for 4 components transferred (4comp) we observe the first benefits of client-side query optimization described in Section 2. When 4 components are transferred, the server processing time decreases at a faster rate than the client-side processing grows. This leads to the decrease in total query processing time: a situation when transferring all components is better than only some of them. This effect, observed for 1 query, should propagate itself to the elapsed batch optimization time (however, as mentioned in Section 2, we will have to add the increased deployment time).

The analysis of processing requirements for different partitions allows us to formulate a number of observations over the optimizer components. As was mentioned above, the components of the optimizer pose differing processing requirements to the hosts. The most demanding are

the preprocessor and the logical optimizer (components 3 and 4), both of which traverse the query tree and attempt to apply rules to subexpressions. Because the logical optimizer checks more rules, its processing time is slightly larger than that of the preprocessor. The differences in processing requirements are reflected in the measurements.

After observing the processing times within the batch and elements of the processing, we turn to our primary performance metric: the elapsed time needed to deploy the components and optimize a batch of queries. We can expect that the effect observed for one query will propagate itself to the batch processing time: at some point, the server-side benefits of component transfer will outweigh additional client-side burdens.

4.4 Measurement: elapsed batch optimization time in relation to batch size

The following series of figures presents the relationship of the elapsed time to process a batch of queries (in milliseconds), the number of queries in a batch (1, 10, or 30 queries) and the number of components on the client (0 to 4 components), for different numbers of clients (1, 4, and 8 clients).

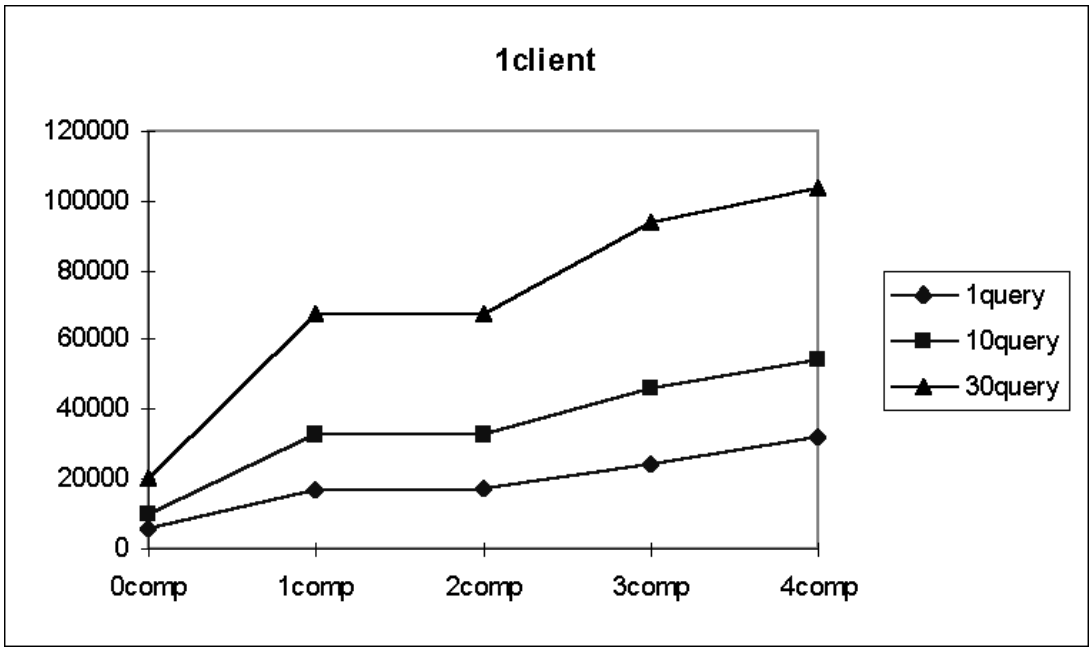

Figure 6-9 Measurement: elapsed time as a function of number of components transferred and query batch size: 1 client

In above case, there is only one client. We observe an increase in the elapsed time with the number of components transferred for all query batch sizes. This means that for all batch sizes it is best to keep all components on the server. The shape of the graphs corresponds to the processing requirements of the components and the relative importance of deployment time. For query batch with 1 query, the deployment time forms the primary component and the differences in processing requirements do not have much influence. For 30 queries, the deployment time forms a small percentage of elapsed batch optimization time: the differences between allocations are caused by processing times, and are therefore much higher: the difference between 0comp and 1comp allocation (and between other allocations as well) is bigger for 30 query batch than for 10 query batch.

In case when there are 4 clients, we observe the following relationships:

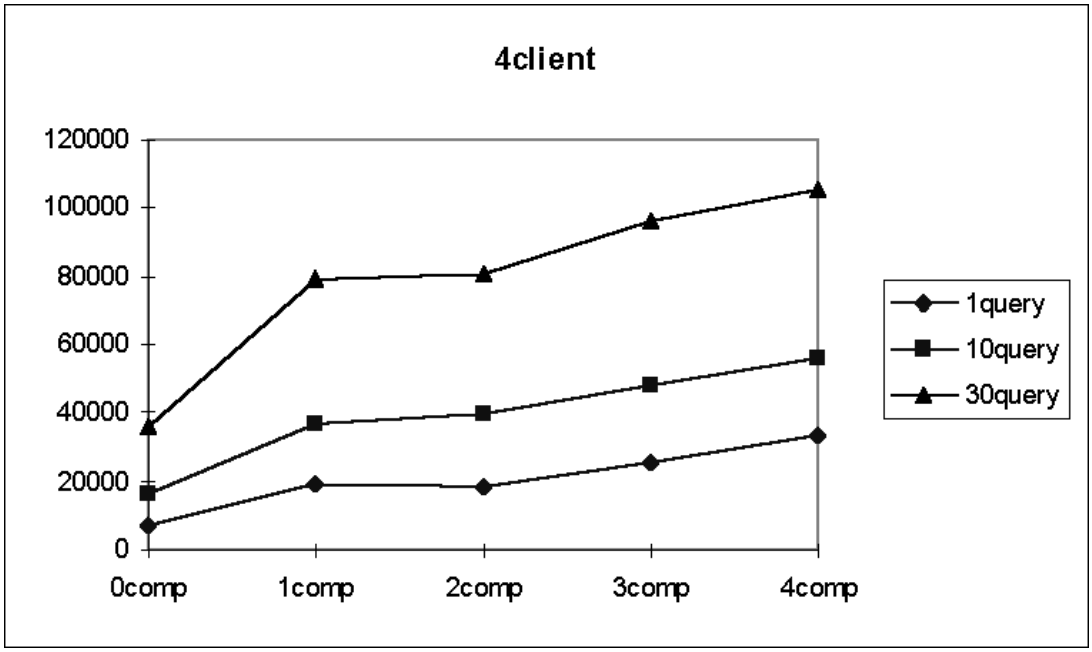

Figure 6-10 Measurement: elapsed batch optimization time as a function of number of components transferred and query batch size: 4 clients

Also in this case the elapsed time grows with the number of components transferred.

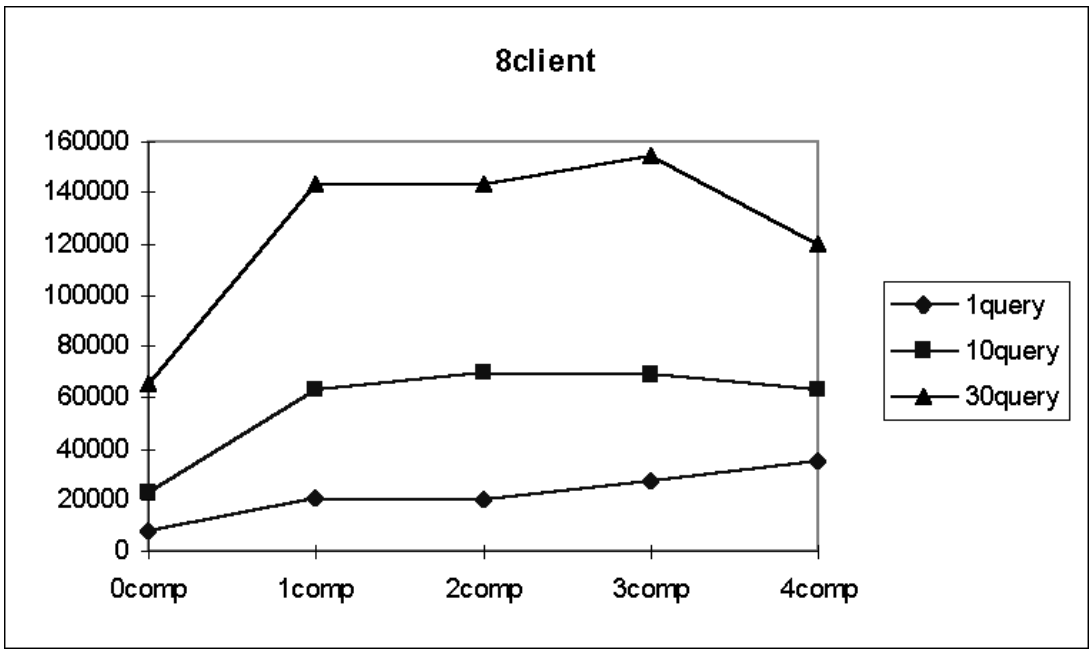

Figure 6-11 Measurement: elapsed batch optimization time as a function of number of components transferred and query batch size: 8 clients

For 8 clients we observe that for larger number of queries in the batch, it becomes more beneficial to transfer components to the client than leave them on the server. The effect observed for 1 query (see Figure 6-8) propagates itself therefore to larger workloads. This can be seen both for batches of size 10 and 30. It is still best to keep all components on the server, but transferring all components is more beneficial than transferring all of them. This is the onset of benefits for client-side distribution of components we observed for one query.

We draw the following conclusions from the above three results: it is much more beneficial to transfer the components for large query batches than for small batches. This is consistent with the intuition. The components have only to be deployed once. For larger batches, the deployment time, which is the primary burden of increased client-side processing, will form a smaller part of the elapsed batch optimization time.

We also observe that it is more beneficial to transfer components for large client populations than for small ones. While for small client population transfer of components causes increase in the elapsed batch optimization time, for larger client populations the management of simultaneous connections becomes an important factor. When some (but not all) components are transferred to the client, the server still processes the query to some extent. It still has to create

data structures associated with the optimizer: its rule set and search engine. This remaining server-side load causes the elapsed time to grow with the number of components transferred. When, however, even this load is shed to the client, the server does not have to create the optimizer data structures. It can then allocate its resources to faster management of connections. The decrease in server time is larger than increase in client time, leading to the decrease in elapsed batch optimization time.

The above three measurement results illustrate the benefit of client-side optimization: we have observed that with sufficient levels of parameters (batch size and number of clients) it becomes beneficial to transfer components to the client. The measured result is consistent with the intuitive treatment in Section 2. There, we predicted that large batch sizes will make client-side optimization more beneficial. We also stated that similar effect will be observed for increasing client populations. We now present the measurement results pertaining to that factor.

4.5 Measurement: elapsed batch optimization time in relation to client population.

The influence of client population time on elapsed batch optimization time could already be observed in the previous section. To make the presentation more clear, we present a graph of measured elapsed batch optimization time for one query batch (30 queries) as a function of number of clients. Again, the Y axis represents the elapsed batch optimization time in milliseconds.

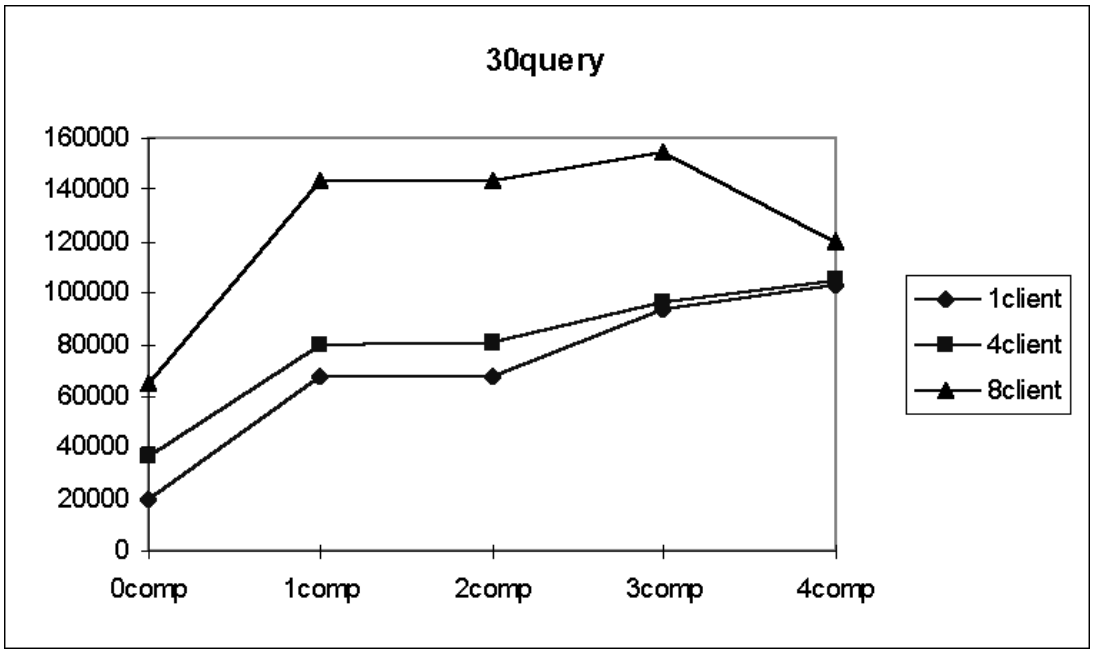

Figure 6-12 Measurement: elapsed batch optimization time as a function of number of clients.

In the above figure, we observe that the increase in client population from 1 to 4 brings about a minimal change in the elapsed time graph. However, when the client population increases to 8, we observe a sharp increase in elapsed time for all component partitions. This phenomenon can be interpreted as follows: for 1 client and 4 client populations, the server operates under its nominal capacity, which allows it to schedule connections without delays. Increase in the number of clients leads to larger probability of requests arriving almost simultaneously, inducing waiting delays. The waiting delays may depend on the architecture of the server: in our experiments the server is a single-processor machine which has to serialize different connections. At some point, the server becomes saturated to such an extent that it is more beneficial to use all of its resources for management of connections and to off-load all the components to the client. The relatively small level of client population at which this effect takes place is caused by high server processing and unmarshalling costs. Introduction of parallelization on the server side would increase the level of client population at which the additional delays occur.

It is important to note that the client population size does not have the same influence on the batch processing time as the batch size. This can be observed by comparing the above figure with Figure 6-11. Increasing the batch size does not affect deployment time: it makes deployment time count less in the batch optimization time. Increasing client population affects both

the deployment costs and the processing costs. As suggested in our intuitive description, the increase in client population influences more query processing activities than the batch size: it may require therefore more accurate modeling than in our simple model.
The measurements provided us with the indication that the concept of client-side query optimization can be beneficial for certain levels of system parameters: the client population and the size of the query batch. Both of those parameters are present in our analytical model: we can therefore attempt to use the model to calculate the model metric and compare the measured and modeled results.

5. MODEL RESULTS

The analytical model presented in Section 3 contains a formula for calculation of elapsed batch optimization time. To calculate it, we have to provide the values of model parameters in Table 6-1. We first describe how these values have been obtained, and then present the model results for the relationship of the three model factors: the query batch size, the client population size and the component partition. We restrict ourselves to the parameter levels at which we observed the benefits of client-side optimization, to see if these benefits can be predicted by the analytical model.

5.1 Model parameter levels

We have assumed the network transfer rate C_n to be 1,000,000 bytes per second. As shown in [Bog88], Ethernet bit rates are close to nominal for small loads, which was the case in our experiments. We therefore assumed the nominal transfer rate within the analytical model.
The processing power of each of the clients has been found to be approximately 2 times smaller than that of the server ($f_c=0.5$), using Java benchmarking [Ben97], in which Java applets are used to measure the performance of remote hosts. We deem this approach to be more suitable for our goals than any industry-wide host processing power benchmarks, because we are more interested in the performance of the Java virtual machine on a host than in the performance of the host itself.
The average transfer size of the components has been determined based on the sizes of the Java class files which are transferred over the network. This is a potential source of inaccuracies in

the model, as it is the Java virtual machine implementation which decides which class files have to be loaded at which moment. We have chosen a conservative approach by assuming that all classes used by the given optimizer phase will be loaded. This caused the modeled values to be larger than the measured ones. The largest component size was 240 Kilobytes. The sizes of the query objects transferred over the network (b_k) have been found to be very small for query strings (150 bytes) compared to query tree objects (~6K bytes).

The measurement results for 1 query and 1 client allowed us to estimate the processing requirements of different components (t_{pi} parameters in the model). We have measured the time spent by a query in the client, in the server and on the network. As we have measured these metrics for different amounts of components on the client, we were able to isolate the processing times for each of them. For example, the time spent in the component 3 on the client was calculated as the difference of the times spent on the client with 3 and 2 transferred components. For the example query, the parsing took ~65 ms, syntax checking ~2 ms, the preprocessing ~220 ms and the logical optimization ~320 ms. The marshalling times were obtained by measuring the server-side processing time for the 4comp partition with 1 client: in this case the server does not process any components, and there is only one connection. The deserialization of the query object has been found to take ~400 ms, which points to marshalling as an activity with large influence on the overall performance of the system. We have assumed the marshalling and unmarshalling speeds to be identical. We assumed $r=1.2$, taking into account non-linear influence of multiple connections on processing speed.

After the values of model parameters have been set, we have calculated the elapsed batch optimization time to estimate the influence of query batch size and the client population size.

5.2 Model: elapsed time as a function of query batch size

The calculation of the elapsed batch optimization time as a function of query batch size for the largest measured client population (8 clients) produced the following result.

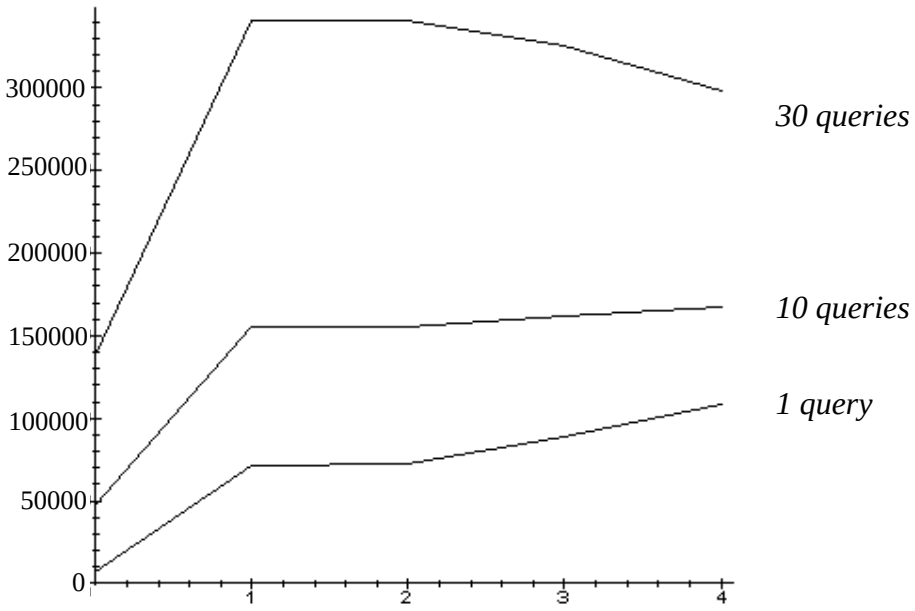

Figure 6-13 Model: elapsed time as a function of query batch size: intranet case, 8 clients.

The figure is an analytic equivalent of the Figure 6-11: the X axis represents the number of components transferred, the Y axis the elapsed batch optimization time in milliseconds, and the three lines represent the measurements for three batch sizes: the lowest for 1 query, the middle for 10 queries, and the top for a 30 query batch (all for the situation when 8 clients are connected).

The discrepancies in the measured metric values are caused by conservative approximations of system parameters, and relatively simple modeling of server process architecture. We note however, that the goal of the analytical model in case of component allocation would not be the accurate prediction of the batch optimization time, but rather the determination of a component partition for which this time is minimal. In all cases, the model accurately predicts the optimal component partition; the one in which all components are placed on the server (*k=0*). This is consistent with the measurement, and caused primarily by low marshalling and transport costs

for the query string. Similar to the measurements, the model shows increasing benefit of client-side execution for components 1 to 4.

Having analyzed the influence of the query batch size on the elapsed batch optimization time, we turn to the size of client population as a factor.

5.3 Model: elapsed batch optimization time in relation to client population size

The following figure shows the relationship of the elapsed batch optimization time to the client population size obtained in the analytical model, for the largest query batch (30 queries). As in Figure 6-12, the lines represent the elapsed times for client populations of 1, 4 and 8.

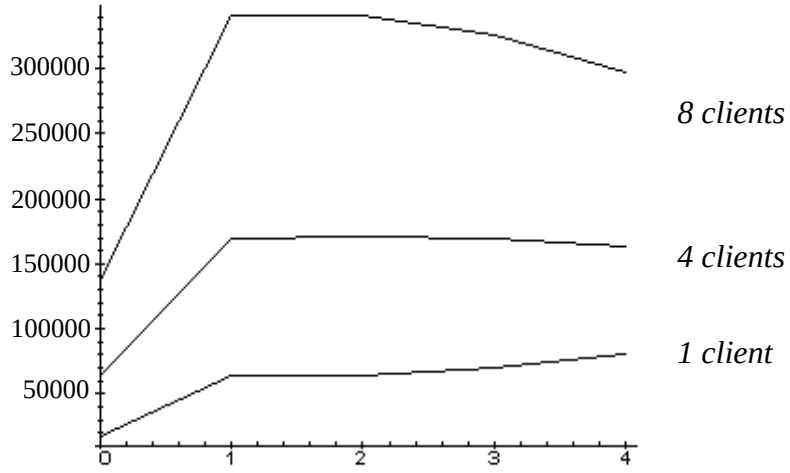

Figure 6-14 Model: elapsed time as a function of client population size: 30 query batch.

As in the measurements, the analytical model predicts increasing feasibility of client-side execution for increasing client populations. However, in the measurements, the difference between 1 client and 4 client population has been less pronounced than indicated by the model. This can be caused by the relatively simple modeling of simultaneous connections in our model: in the model, the influence of multiple connections is represented by the increase in the component processing time by factor c^r (see equation (6-3)). In the measurements we observed the onset of saturation effects: a relatively stable performance for a small number of connections and a sharp increase for large number of connections. These effects are strongly dependent on the process architecture and connection management on the server side, which is provided by the operating system and the underlying network protocol handlers: that's why we have not

included them in the model. This illustrates the need for a component allocation model to accurately represent the process architecture of the server. More accurate modeling of simultaneous connections is therefore one of the possible extensions of our research.

The comparison of the results obtained in the analytical model and the measurements leads us to the following conclusions.

For the two factors, the model illustrates the onset of client-side optimization benefits: both for the measurements and the model, those benefits occurred at the same levels of system parameters. The model is, however, not as accurate in terms of the level of its metric: the elapsed batch processing time. That's why a model such as ours can be used only for determination of partition, and not for prediction of the values of the elapsed optimization time. This can, therefore, be the only role in which such a model will be used in a centralized allocation architecture.

The relative accuracy of the model is achieved thanks to the restricted measurement setting, which was designed to minimize outside influences and variations in system parameters (query complexity, network speed). In real-life situations, the model would have to include much more complicated modeling of server process architecture and take into account variations in network speed. These variations may lead to the introduction of request frequencies in the model; in this case, queueing theory methods can be used [Mit87]. This is one of the possible improvements in the model, more of which are listed in Section 7.

The fact that the model results have been found to be relatively accurate in terms of optimal partition choice, allows us to examine the model predictions for the situations which have not been measured.

6. MODEL EXTRAPOLATION

The measurement setting imposed certain restrictions on the hardware and software configuration. The analytical model allows us to determine the optimal component partitions for other values of system parameters. We first consider a situation in which the marshalling costs are significantly lower than measured. We also extrapolate the situation in which the physical optimization component is added to the 4 existing components. We then turn to other possible changes in model parameters: we first assume high processing costs and we analyze the situation in which no deployment is needed. Finally, we consider the general situation: one in which

the components have to be deployed both to the client and the database server from the separate component server.

6.1 Extrapolation: higher marshalling speed

The measurements have indicated that marshalling costs take a large percentage of the overall processing costs. An interesting application of the analytical model is the calculation of the elapsed time in situation when marshalling does not form a bottleneck. We assumed the marshalling rate to be 5 times higher than the measured one, and calculated the following result:

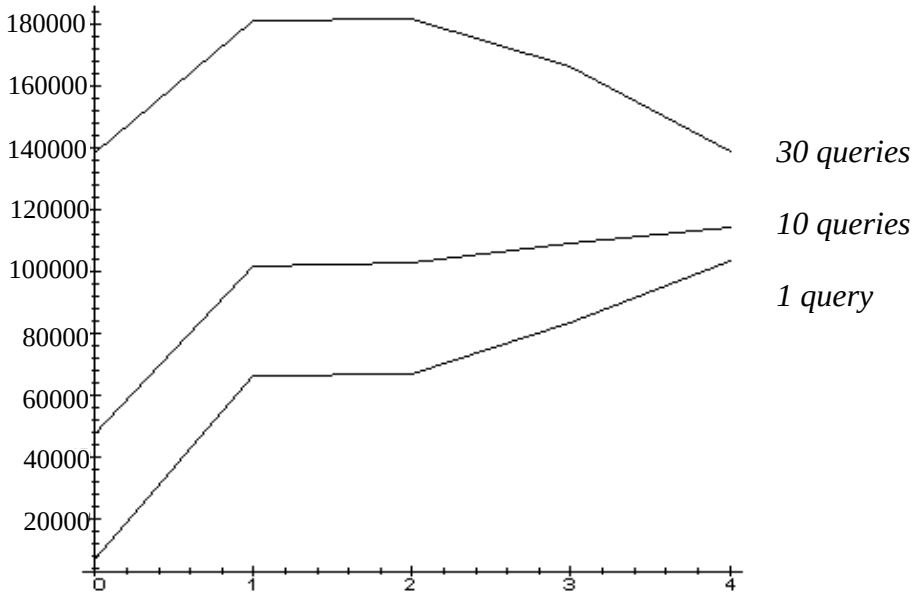

Figure 6-15 Model: elapsed batch optimization time as a function of query batch size, low marshalling costs.

We see that the model estimates the transfer of all components to become beneficial for 30 batch query. The decrease in marshalling costs causes other costs to play more important role: the network transfer costs and the processing costs.

6.2 Extrapolation: addition of physical optimization

In our experiments, the optimizer consisted of four components, last of which was the logical optimizer. Logical optimization can be followed by physical optimization, during which we choose the most beneficial implementation for algebraic operators.

We used our model to analyze the situation in which the physical optimization component can also be executed on the client. In this case, we add the fifth component to our component sequence. We assumed that the physical optimization component has the same size as the logical optimization component, but its processing time for the example query is 3 times higher. We also assumed that the physical optimization component transfers the same amount of data as the logical optimization component. We have then calculated the elapsed batch optimization time as a function of the number of clients.

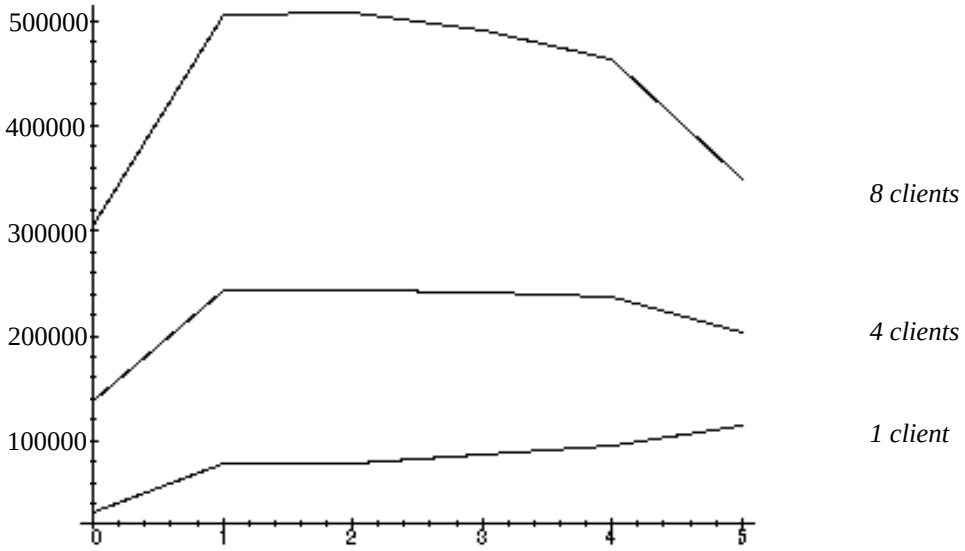

Figure 6-16 Extrapolation: elapsed batch optimization time in relation to client population (5 components)

The figure differs from figure 6-14 by the addition of the right-most component (physical optimization). As we can see, the addition of a component with high processing demands leads to the increasing benefit of component transfer. On figure 6-16, for 8 clients, the transfer of all components to the client becomes equally beneficial to all-server allocation. Also for 4 clients the high processing costs make transfer beneficial. For 1 client there are no additional benefits: the best allocation is still the all-server allocation.

We can see that processing costs of the components can have large influence on the optimal allocation. We can now analyze what happens if the processing costs of all components are much higher than those measured.

6.3 Extrapolation: higher processing costs

We assume a situation in which all processing costs of the components are four times higher than those measured. We also increased the size of the transferred query objects four times. This situation can occur if the input query is much more complex, requiring both more costly marshalling, as well as more costly optimization. The increased optimization costs are caused primarily by the larger number of transformations which have to be applied, and costs of traversal of larger query tree.

The relationship of elapsed batch optimization time to client population can be then illustrated on the following figure.

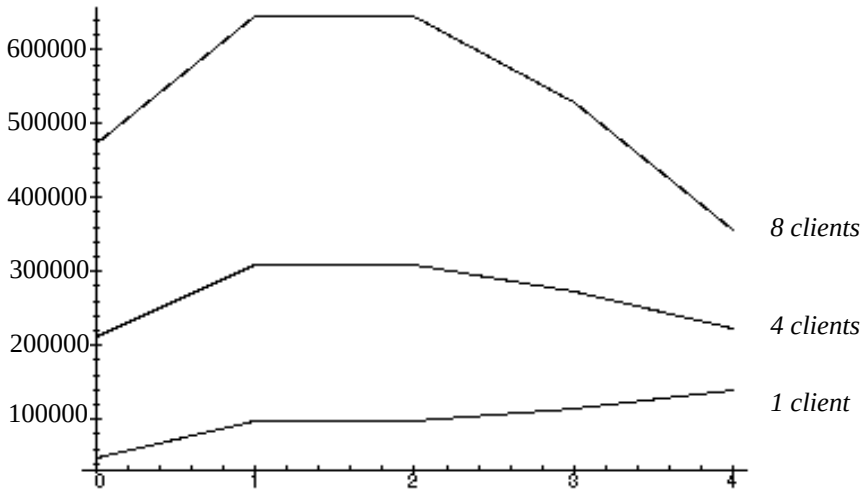

Figure 6-17 Extrapolation: elapsed batch optimization time as a function of client population (high processing costs).

As we can see, the situation has changed from the one measured (see Figure 6-12). For 8 clients, the all-client allocation becomes the most beneficial. This is caused by the fact that the processing costs increased while the component transfer costs remained the same: this is a typical result of posing a more complex query to the system. Note that we assumed that the transfer costs of the query object increase in the same rate as the processing costs: in many situations, the transfer costs of the query object will increase at a much lower rate. In this case, the benefit of client-side processing will be even more pronounced.

6.4 Extrapolation: no deployment costs

In the intranet setting, components may be pre-allocated to the clients before the query batch is submitted. In this case the T_d parameter in our model is set to *0*. We also speed up the marshalling by a factor of *2*. We obtain the following relationship for a 30 query batch.

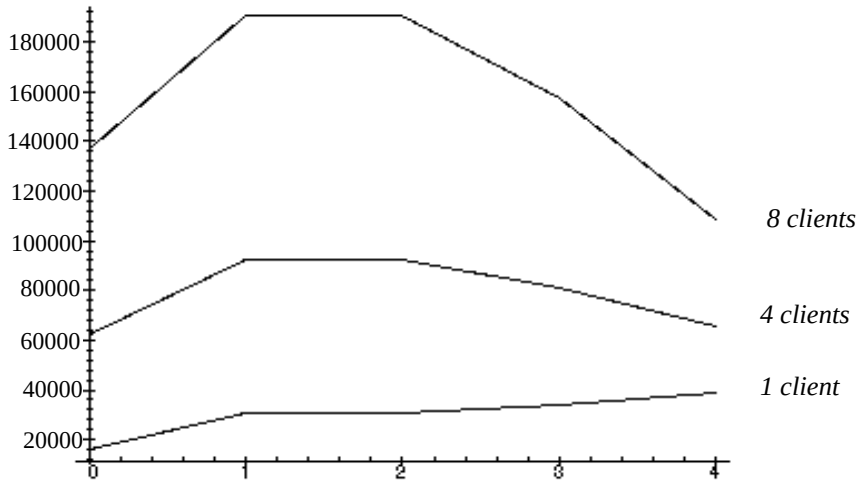

Figure 6-18 Extrapolation: elapsed batch optimization time as a function of client population (no deployment costs)

The removal of deployment costs leads to increased benefit of transfer for 4 and 8 clients. For 1 client, there is no benefit, as the server does not have costs connected with multiple connections. We note that the change in the result will be more pronounced for smaller batches, for which the deployment time forms a larger percentage of the elapsed batch optimization time.

6.5 Extrapolation: separate component server

An interesting extrapolation is the situation in which the component server is separate from the database server. Such a situation can occur in three-tiered function-based models [Ber96], in which application servers are placed between the clients and database servers. In such an architecture, the component transfer costs will have to be counted both for the client and the database server. We assumed identical network speeds between all three hosts: the component

server has the same processing speed as the database server. It also reacts in the same way to multiple connections. We observe the following relationship for 30 query batch.

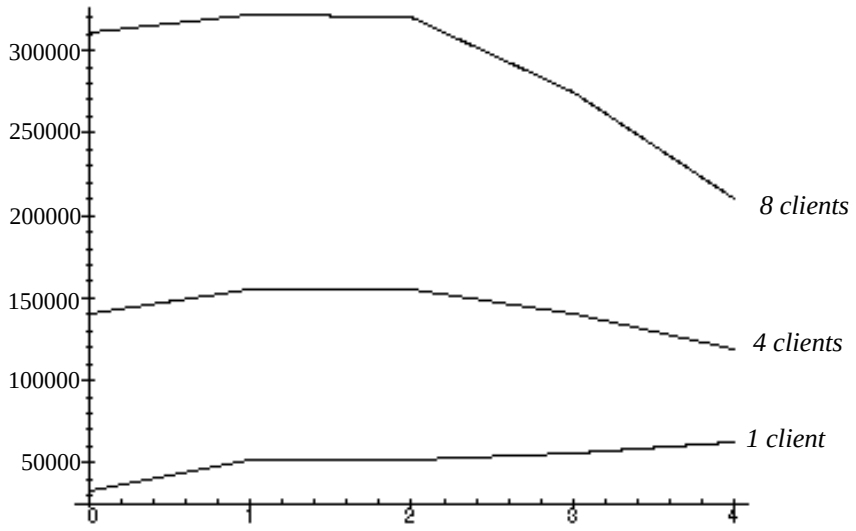

Figure 6-19 Extrapolation: elapsed batch optimization time as a function of client population (separate component server)

The influence of component allocation to the database server is observed for “low” allocations. The all-server allocation requires transfer of all components to the server: this causes it to lose its benefit for large number of connections. For 0comp case, the server has to perform all activities: unmarshal components received from the database server, marshal query strings, and processes queries, all while managing multiple connections. For 4comp case, the server has only to unmarshal query objects: the decrease in server processing time is larger than the increase in client deployment, processing and marshalling time. As we can see, introducing separate component servers can make client-side processing more beneficial.

The five extrapolated relationships are not the only ones which have been calculated. In Appendix E, we analyze the influence of the network speed and the difference in the processing speed between the client and the server on the model metric.

7. POSSIBLE EXTENSIONS AND FURTHER RESEARCH

The analytical model and the measurements were performed using simplifying assumptions, which restrict their usability. We discuss those restrictions and point to possible areas of further exploration.

In our research, we have assumed that there is one server-side host receiving the processed queries. An important extension of the research would be the introduction of the distribution aspect on the server side: both in the terms of distribution of data and server-side computation. If only the data is distributed, then partition techniques known from the distributed database research could be integrated in our model, so that query fragments would be transmitted to the servers holding the proper data [ÖzVa91]. Such a modification will require introduction of network transfer costs between each of the hosts and the clients. Besides distributing components to the clients, we can transmit them also on the server-side, similarly to the MARIPOSA system [Ston96].

An important restriction of our model was the sequential character of the component structure. The extension of the model for component graphs can be a subject of further research. It is true that query processing is often executed in sequential stages, but even then there are components which are used by many other components. For example, a data dictionary may be used both by the syntax checker, the preprocessor and the logical optimizer. In the simple case this can be modeled by coupling the data dictionary with the first component which uses it. However, in the situation of an arbitrary processing system (e.g. client-server encryption system or database design tool) we can be confronted with a situation in which the components are not structured in a sequence, but in an ordered graph, with a designated start and end node. In the graph, nodes can represent components and vertices the flow of data between components. In this case, the vertices can be annotated with the transfer costs and graph techniques such as flow network algorithms [Cor92] can be used to determine optimal partitions.

In the analytical model, we have assumed a uniform allocation of server resources to connections. This is the simplest form of allocation regime: other forms are possible. For example, a server can allocate resources based on priorities, or using load balancing. These forms of resource allocation would then have to be accounted for in the performance model.

In our model, the time spent by k components on the client is the sum of processing times for all those components. In some situations, the processing resources on a host can be restricted in such a way, that increased number of components on the hosts leads to exponential increase in processing times: for example, a host can be able to accept only 2 components, and transfer of each component above 2 will incur additional swapping costs. In the experiments, we have found that this never took place, and that's why we have not included this effect in the model. In real-life situations, where other tasks are executed on the client, this situation is possible. Our model can be extended with these considerations, as available memory and component sizes can easily be measured. Then, the formulas for elapsed time will include memory-related parameters which take the swapping effects into account. Additional research is needed to determine the precise extent of the swapping delays, as these can be modeled differently for different memory architectures.

In the measurements, even though it was relatively easy to measure per-component processing loads, it was difficult to estimate the sizes of the components deployed. These sizes depend on the sizes of Java class files used by each of these components, and the combination of class files for a component depends on the query posed. We have approximated those sizes in our model by averaging them over all components, because we were not able to obtain precise transfer sizes, which are governed by the Java virtual machine. Accurate modeling of deployment sizes may lead to smaller numerical discrepancies between the model and the measurements.

In the analysis, we have assumed that the client resources are not claimed by tasks other than the optimizer and the surrounding Java environment. In the measurement setup, we took care to reproduce this isolated situation. However, in real-life situation, the client components will always have to compete with other jobs for resources: jobs such as other Java applets, editors, WWW browsers. This will make it difficult to provide accurate client parameters to the model. One solution to that problem is dynamic monitoring of the virtual machine performance and subsequent dynamic changes to the f_c parameter used in the model. This would allow for a certain adaptability of the model and would be usable on fast networks: however, the influence of network delays on the accuracy of monitoring can be prohibitive in the Internet case.

In the general version of the analytical model in Appendix D and the extrapolation in Section 6.5, we have assumed that each of the connections will require deployment of the components

to the server side. In many cases, this involves deployment of the same components (e.g. logical optimizer) multiple times. In the current model, we separate the connections and server-side resources allocated to them: that's why we consider each connection as needing a separate deployment phase. However, in many cases it may be possible to cache components on the database server, so that they do not have to be deployed from the component server each time. The same consideration applies to the client side, in which a cache server can be placed in a client group: the components will then be deployed once from the component server to the group cache, and from the group cache to the client when requested. The inclusion of caches, and in general intermediaries in our model is an important area of possible future research.

Our model represents the network delays as a constant. While this may be accurate for intranets, network delays will always be variable in the Internet case. Even if those delays could be expressed in an analytical formula, the usability of the formula would be seriously restricted because of the impossibility to gather the parameters to calculate it. That is why we propose other than hierarchical means of component allocation for the Internet case.

Besides a constant network delays, we have not considered the effects of clients posing queries with varying frequencies, leading to requests arriving at the servers at different times. As mentioned above, the modeling of this situation would have to introduce arrival frequencies as parameters in the model. In such case, the proper way to build the model of the system would be to construct a queueing model (or queueing network model) of the system, and use the large body of available research to analyze its performance.

8. CONCLUSIONS

In the previous sections, we have introduced an intuitive description of the performance problem in the form of the activity schemas. We then captured the behavior of the system in an analytical performance model, expressing the query batch optimization metric in terms of model parameters. In cases of two of those parameters, we have performed measurements and compared their results to the analysis. We summarize the qualitative results of the measurements and analysis for the four activities in client-side query optimization: client-side processing, server-side processing, marshalling and network transfer.

In all measured situations, the most profitable partition was the allocation of all components to the server. This was caused primarily by **high marshalling costs of query trees**, which formed the largest part of the query processing activity in the experiments. While marshalling will not be a bottleneck in many real-life situations, our experiments show that this activity should be given much attention in performance considerations. In our situation, the extrapolation of the analytical model has shown that by decreasing the marshalling costs only, we can achieve significant performance benefits. The marshalling costs can be improved by influencing the following parameters:

- The implementation of the serialization libraries. Marshalling costs can be decreased by more efficient implementation of the serialization routines for complex object trees such as present in query optimizers. The implementation used in the test (beta version of Sun's Object Serialization library for Java) is expected to improve significantly in the future. We have examined the influence of those improvements using the analytical model.
- The host processing power. Because marshalling and unmarshalling is a processing activity, it is influenced by the power of the executing host.
- The number of clients. This factor influences the server-side marshalling costs, as the marshalling activities have to compete for resources with other connections. This problem can be tackled by specialized implementation of serialization libraries on the server side, which make use of available process primitives such as threads.
- The complexity of the query. Queries of increased complexity can be expected to increase the marshalling costs, due to the increased complexity of the query tree.

The **client-side processing** of the query is performed by the transferred components. In the measurements, we have not found the client-side processing to form a significant load on the client. In all cases, the client was able to cope with the component activities without resorting to swapping. That can be the reason why this processing never formed the bottleneck activity. In the experiments, however, the optimizer was the only task on the client. In reality, the optimizer will have to compete for resources on the client, in a similar way this happened on the server. This may lead to greater influence of the client processing time in the elapsed time. Client processing time is influenced also by the host processing power, and by the complexity of the input query. More complex queries will lead to increased processing time, and may lead to increased marshalling time and network transfer time. This is caused by the fact that for more

complex queries, not only more optimization rules will be applied, but the query tree will become more complex.

The **server-side processing** of the query can be a significant factor in the performance of the system. In the measurements, we found this processing to be strongly influenced by the size of the client population. At some point, the management of growing number of connections slows the processing of the query to such an extent, that server-side processing becomes a bottleneck: this is the moment at which the components can be transferred for optimum performance. The server-side processing can be influenced by the server processing power: however, in the server case much can be achieved through appropriate process architectures. As in the case of the client, the server-side processing time is influenced by the complexity of the query. We have observed that it is best for the server to begin with allocation of all components to the client, as this allows it to use its resources for management of connections. In our measurements, this situation occurred already for 8 clients.

We have found the **network speed** to be high enough not to form a bottleneck during the measurements. However, in the experiment setting we were able to use the network in a situation of extremely low load, at which the Ethernet has been found to operate at near-nominal levels. In case of everyday network traffic larger loads can be expected; however, experiments have found Ethernet to be relatively stable even at high loads [Bog88]. We have attempted to estimate the situation in which the network speed would lie at levels significantly lower than the observed (see Appendix E). However, the model is based on a constant estimation of network speed, which in the case of Internet is less valid than in the intranet case. It may therefore be the case that the Internet transfer speeds will form the bottleneck in our activity diagram. Future research is needed to accurately model the effective transfer delays on the Internet.

The most important conclusion of the measurements and the analytical modeling is the increased benefit of client-side optimization for sufficiently large client populations and query batch sizes. In our experiments, already the change from 4 to 8 clients brought about these effects. The calculations in the analytical model confirmed the observed result. Also, in all observed cases, processing of all components on the server was most beneficial, due to the high marshalling costs of query objects.

Herein lies the benefit of the approach: the analytical model and the measurements can be used to **gain insight** in the mechanisms and isolate important factors from irrelevant parameters. It is this insight which allows to improve the performance of the system, possibly through automatic determination of component partition as described in Chapter IV. Our model is a step in the direction of such a model. Although catering for a restricted intranet environment, it shows that a certain degree of consistency can be achieved between the predicted and measured performance of the system. The degree of consistency, while insufficient to provide quantitative predictions of system performance, allows for determination of the optimal component partition, which is the goal of our model.

The analytical model and the measurements are the main contributions of this chapter. However, we feel that we have also contributed to the understanding of the performance aspects of client-side distribution of components in general. Our model can be applied to any sequence of communicating components, and as such does not have to be used for query optimizers only. As distribution of components becomes more popular with the emergence of architectures such as Java, the performance issues are bound to play a larger role in placement decisions. We hope that our approach can be applied also in those future systems.

VII. Summary and future research

We briefly summarize the thesis and its contributions, and outline the main directions of possible research.

1. SUMMARY

In this thesis, we focused on aspects of complexity and scalability of query optimization systems. **Complexity** manifests itself in the emergence of new data models, query languages, and algebras. This evolution requires the use of optimization strategies, which determine the ordering and time of application of transformations. Such a **strategy**, geared towards replacement of nested subqueries by joins, is one of the contributions of this thesis. The strategy is based on the set of available operators and transformations: it provides the ordering and structuring of those transformations, and determines the splitting strategy needed to introduce join operators. The strategy has been implemented in the Joquer query optimizer, and its final form is the result of experimentation with different complex nested queries.

The intranet and Internet environments pose new **scalability** requirements on database systems, manifesting themselves in the variability in client population, network transfer speed, and complexity of the query workload. The problems of scale in database systems were often approached by server-side scale-up of server hardware and process architectures. We proposed **component allocation** to clients as an additional method of coping with scalability problems. Our component allocation model consists of two forms: the centralized model, based on global notion of utility, and a distributed model, based on the local notion of utility. Within the centralized model, the allocation of components is determined by optimization of a central **analytical performance model**. In the thesis, we have presented such a model and performed

measurements to assess its suitability. The measurements have shown that component transfer to the client can be beneficial for high levels of client population and sizes of query batches. The relative accuracy of the model allowed us to present extrapolations of the model for non-measured values of parameters, and develop intuition on the relative importance of query processing activities.

The **distributed model** of component allocation is another contribution of this thesis. It is well-suited for situations in which hosts have differing interests, making it impossible to establish a single measure of utility. In this case, we do not use a global performance model, but rather establish the allocation in an **computational economy**, in which agents (client and server) involve in **auction**-like interactions to purchase components according to their local preference. This approach allows the agents to have differing and conflicting goals, and still reach a satisfactory (pareto-optimal) allocation. We have presented the optimizer economy in detail: we have chosen its product, pricing mechanisms and auction-based method of exchange.

2. PROPOSALS FOR FUTURE RESEARCH

During the development of this thesis, additional aspects came to light which would benefit from further investigation. Many of those aspects have been noted in the course of the presentation in the thesis chapters. Here, we repeat the ones which we deem to be the most interesting.

There are a number of possible research directions within the unnesting strategy. The new formal approaches [Che97] to strategy representation show great promise in their ability of precise expression of optimizations. Their benefit lies in the fact that besides precision of representation, they can be used to perform validations of the strategy, and possibly automatic implementation. The strategies such as the one presented in this thesis are quite complex: they involve exchange of information between the transformations, and complex flows of control. As such, they are difficult to describe and implement: it would be therefore interesting to establish to what degree they can be represented by formal strategy representations.

The ultimate goal of formal development of query optimizers lies, however, not only in their formal representation, but also implementation. In this way, we would be able to create not

only provably correct representations, but also implementations. Ideally, the input of the optimizer generator should only describe the desired characteristics of output expressions: the optimizer generator will then generate both the algebra, the transformations, and the strategy to achieve the specified goal. The research in that direction would have undoubtedly take into account the results in the area of provable program correctness and program generation [Gries81].

The allocation model presented in this thesis leads to a number of possible research directions, some of which have already been mentioned in previous chapters. In the short term, the allocation model can be extended to include the physical optimization activity, with the possible use of a client-side data dictionary. In the long term, the component allocation model can be extended to other than sequential component structures, which will allow to apply component allocation to new application areas. In this case, graph techniques can be used to model communication between components and to determine the optimal partition (cutset) of components.

Besides improved modeling of the partitioned applications, we can extend our approach to environments less restrictively defined than in our model. While we do not allow the components to be allocated to clients other than the requesting one, it is perfectly feasible that this would be possible in intranet environments. There, multiple workstations will be available for allocation of components, even for optimization of queries for other clients. In the presence of distributed database servers, the same may become possible on the server side (as is the case in MARIPOSA). In such a situation, the components will have to be allocated to a network of processing hosts, leading to changes in both the centralized and market-based allocation.

3. FINAL REMARKS

We feel that both contributions of the thesis (the strategy and the component allocation model) are what makes this research interesting. It is often the case that optimizer strategies become so complex that their implementation becomes very difficult and they are very inefficient in their execution. Placement of the optimizer within the distributed environment may very well worsen its performance even more. Component allocation is one of the ways in which the performance can be improved, by using computing resources already available.

We feel that performance and scalability considerations belong in the design process of all software systems. Therefore, those issues have to be taken into account while designing methods and techniques, if they are ever to be applied in practice. We hope that the mixture of architectural and practical considerations presented in this thesis will improve its usefulness.

We also feel that component allocation models such as the one presented in this thesis can be applied to many other computing tasks besides query optimization. If this would happen, we would find that in most cases investments in hardware can be prevented by using the infrastructure which is already there: leading to prudent instead of wasteful use of resources.

Appendix A. Rule set analysis

The initial rule set proposed in [Ste95] is the starting point of our research. In the sequel, we present the improvements introduced in our rule set.

1. MODIFICATIONS OF THE INITIAL RULE SET

During the implementation of the optimizer, we were able to extensively experiment with the rule set of [Ste95]. This allowed us to detect **problems in the rule set**: problems caused by inconsistencies between rule sets used in different phases of the optimization and problems caused by missing transformation rules. For each of the detected problems, we provide the needed modification.

1.1 Inconsistencies between rule sets

The rules in our rule set are a combination of diverse rule sets. Problems arise if there are inconsistencies between rule sets; for example, if a later phase of optimization attempts to detect and transform expression forms which are already removed during earlier stages.

In [Ste95], the predicates in the query are transformed into Miniscope Normal Form using rules from [Bry89]. The rules presented there allow for removal of those expressions from the quantifier matrices, which have been brought there unnecessarily during the preceding transformation to Prenex Normal Form. We quote two of the rules from [Bry89], which are in turn versions of generalized De Morgan's laws for quantifiers [Gries93]:

Transformation A.1:

$$\forall x \in X \bullet p \equiv \neg \exists x \in X \bullet \neg p \qquad \text{(A-1)}$$

Transformation A.2:

$$\forall x \in X \bullet \neg p \equiv \neg \exists x \in X \bullet p \tag{A-2}$$

Those two rules transform universal quantifiers into existential quantifiers: a query in Miniscope Normal Form will have no universal quantifiers. However, we note that both rules in Rule 7.5 in [Ste95] have universal quantifiers in their left hand-sides. This means they will never be applied if specified as in the initial strategy, as those two rules will be applied after predicates are transformed into MNF. We have to reformulate those rules as follows:

Transformation A.3:

$$\Gamma[x : f \mid (\neg \exists y \in Y \bullet p[x, y])](X) \equiv \Gamma[x : f \mid \text{true}](X \text{ antijoin}(x, y ; p)\ Y) \tag{A-3}$$

Transformation A.4:

$$\Gamma[x : f \mid (\neg \exists y \in Y \bullet p[x, y])](X) \equiv \Gamma[x : f \mid \text{true}]((\Gamma[v : v \mid \neg p[x \to v_X, y \to v_Y]](X \times Y)) \div Y) \tag{A-4}$$

The replacement of universal quantifiers with negated existential quantifiers has consequences to other parts of the optimizer: specifically to the splitting of predicates.

1.2 Inconsistent splitting rules

Splitting rules in [Ste95] do not take into account nor universal, nor negated existential quantifiers. We illustrate the consequences with an example query

$$\alpha[x : \sigma[y : p[x, y] \wedge \neg \exists z \in Z \bullet q[x, z] \vee r[y, z]](Y)](X) \tag{A-5}$$

If the predicate were to be split using rules 7.8 from [Ste95], the negated existential predicate would be transformed into

$$\neg(\exists z \in Z \bullet q[x, z] \vee \exists z \in Z \bullet r[y, z]) \tag{A-6}$$

If the predicate is left in this form, rules for unnesting quantifiers within collect (Rules 7.4 and 7.5) cannot be applied without further splitting of the predicate. However, the splitting rules in [Ste95] do not allow for full splitting of the predicate into

$$\neg \exists z \in Z \bullet q[x, z] \wedge \neg \exists z \in Z \bullet r[y, z] \tag{A-7}$$

An additional splitting rule is needed:

Transformation A.5:

$$\neg \exists x \in X \bullet p \vee q \equiv \neg \exists x \in X \bullet p \wedge \neg \exists x \in X \bullet q \tag{A-8}$$

The rule allows for better isolation of quantifier matrices suitable for introduction of joins.

1.3 Rule notation

The applicability conditions do not apply to all unnesting rules, what is suggested by the notational convention used in [Ste95].

The rule 7.4.1 in [Ste95] introduces a semijoin in place of an existential quantifier. As we do not want to introduce free variables and base table references in the semijoin predicate *p*, the predicate *p* is required to be a closed dyadic predicate with respect to the variables *x,y* (it contains references to both *x* and *y*, and no other variables). This is enforced by the optimizer.

Rule 7.4.3, with identical left hand-side, does not pose the same conditions on the predicate *p*, as the predicate is not used in the resulting Cartesian predicate. In fact, there should be no applicability conditions on the predicate *p* in this rule. The notation *p(x,y)* in the initial rule set of [Ste95] gives the impression that *p* is an expression in which both *x* and *y* occur, which is incorrect (*p* does not have to contain references to any of the variables). To make this clear in the notation, we introduce the variable replacement notation $p[x \to v, y \to w]$, which denotes the expression *p* with all instances of *x* replaced with *v*, and all references to *y* replaced with *w*. We use this notation only on the right hand-sides of transformation rules. Consequently on the left hand-side, notation *p[x,y]* denotes an expression in which both of the variables *x, y* occur. Using the improved notation, we rewrite the rule 7.4.3 as:

Transformation A.6: (rewritten 7.4.3)

$$\Gamma[x : f \mid \exists y \in Y \bullet p](X) \equiv \Gamma[v : f[x \to v_x] \mid p[x \to v_x, y \to v_y]](X \times Y) \qquad \text{(A-9)}$$

We rewrite in the same manner rules 7.4.2, 7.5.2, 7.6.1, 7.6.2. The new versions are integrated in the unnesting rule set in Chapter III.

Transformation A.7: (rewritten 7.4.2)

$$\begin{aligned} &\Gamma[x : f \mid (\exists z \in \sigma[y : p[x, y]](Y) \bullet q)](X) \\ &\equiv \Gamma[v : f[x \to v_x] \mid q[x \to v_x, z \to v_y]](X \ \text{join}(x, y ; p)\ Y) \end{aligned} \qquad \text{(A-10)}$$

Note that predicate *q* does not have to be a closed dyadic predicate of *x,y*, as is suggested by the *q(x,y)* notation in [Ste95].

Transformation A.8: (rewritten 7.5.2):

$$\Gamma[x : f \mid (\neg\exists y \in Y \bullet p)](X) \equiv \alpha[x : f]((\Gamma[v : v \mid \neg p[x \to v_x, y \to v_y]](X \times Y)) \div Y) \qquad \text{(A-11)}$$

Transformation A.9: (rewritten 7.6.1)

$$\Gamma[x : (g \mid q)[x, \Gamma[y : (f \mid p)[x, y]](Y)]](X) \equiv \\ \Gamma[v : (g \mid q)[x \to v_x, \Gamma[y : f \mid p](Y) \to v \,.\, ys]](X \;\Delta_{x,\,y\,;\,f \mid p\,;\,ys}\; Y) \tag{A-12}$$

Transformation A.10: (rewritten 7.6.2)

$$\Gamma[x : (f \mid p)[x, Y]](X) \equiv \Gamma[v : (f \mid p)[x \to v_x, Y \to v \,.\, ys]](X \;\mathbin{❖}_{ys}\; Y) \tag{A-13}$$

We summarize the modifications in the initial rule set:

- We made the rule set of [Ste95]consistent with the transformation to Miniscope Normal Form. This resulted in changes to unnesting rules, and addition of a splitting rule
- We made notational changes to unnesting rules

1.4 Analysis of the initial optimization

Besides the rule set, we analyzed the structuring of the rule sets and the order of their application: the optimization strategy, using queries with diverse variable dependencies. This allowed us to detect problems in the initial optimization framework of [Ste95].

1.4.1 Improper rule structuring

In [Ste95], the unnesting rules are grouped in 3 groups: one group (Rule 7.4) transforms nested existential quantifiers, (Rule 7.5) - nested universal quantifiers, while the group (Rule 7.6) transforms nested collect queries. The strategy prescribes only that the rules should be applied in that sequence. However, in actual implementation the rules have to be structured in a more complex manner.

The optimization strategy of [Ste95] does not specify whether all unnesting rules should be applied at the same time. However, if all unnesting rules would be applied in the same step, we would get (already after one unnesting step) a nestjoin of all base tables in the expression, independent of their initial nesting level. This is caused by application of rule 7.6.2, which removes all base table references to the top level. Rule 7.6.2 is a “catch-all” rule: it will be applied always if there is a base table reference in the collect expression or the predicate. As an example, consider the following query (“for every hotel, find all guests which live in the same city as the hotel or are theater managers”)

$$\alpha[h : \sigma[c : (c \,.\, \text{city} = h \,.\, \text{city}) \vee \exists t \in \text{THEATERS} \bullet (c = t \,.\, \text{manager})](\text{GUESTS})](\text{HOTELS}) \tag{A-14}$$

According to the strategy of [Ste95], we apply the unnesting rules in top-down manner. Considering the applicability conditions for unnesting rules, the only rule which can be applied on the input query is the rule 7.6.2, which introduces a nested Cartesian product. Other unnesting rules cannot be applied because of non-dyadic selection predicate and the presence of base table in the predicate. If rule 7.6.2. is applied twice, we achieve:

$$\alpha[v : \sigma[c : (c\,.\,\text{city} = v\,.\,\text{city}) \vee \exists t \in v\,.\,ts \bullet (c = t\,.\,\text{manager})](v\,.\,cs)] \quad \text{(A-15)}$$
$$((\text{HOTELS} \;\mathbin{❖}_{cs}\; \text{GUESTS}) \;\mathbin{❖}_{ts}\; \text{THEATERS})$$

The result involves a nested Cartesian product of three tables. According to the strategy of [Ste95], the selection predicate should now be split (rule 7.8.2), leading to the expression

$$\alpha[v : \sigma[c : (c\,.\,\text{city} = v\,.\,\text{city})](v\,.\,cs) \cup \sigma[c : \exists t \in v\,.\,ts \bullet (c = t\,.\,\text{manager})](v\,.\,cs)] \quad \text{(A-16)}$$
$$((\text{HOTELS} \;\mathbin{❖}_{cs}\; \text{GUESTS}) \;\mathbin{❖}_{ts}\; \text{THEATERS})$$

In the following unnesting step (rule 7.4.1), a local semijoin can be introduced between attributes *v.ts* and *v.cs*.

$$\alpha[v : \sigma[c : (c\,.\,\text{city} = v\,.\,\text{city})](v\,.\,cs) \cup ((v\,.\,cs)\ \text{semijoin}(c, t\,;(c = t\,.\,\text{manager}))\,(v\,.\,ts))] \quad \text{(A-17)}$$
$$((\text{HOTELS} \;\mathbin{❖}_{cs}\; \text{GUESTS}) \;\mathbin{❖}_{ts}\; \text{THEATERS})$$

The result of the optimization, although succeeding in removing base table references from the map function, contains expensive nested Cartesian products. This has been caused by early application of the rule 7.6.2, which replaces base table reference with reference to the attribute of the nested product result. To avoid this effect, rule 7.6.2 should not be applied in the unnesting phase. We addressed the problem by defining the transformation ordering and ruleset structuring in Chapter III.

Appendix B. Transformation rules

In this chapter, we present the transformation rules used in the implemented query optimizer. We use the notation as described in Chapter III: capital letter variables denote table expressions, *f* and *p* denote respectively functions and predicates.
We describe the rules in the order of the optimization stages.

1. STANDARDIZATION RULES

During standardization, redundant formulations in input queries are combined using the following transformations:

Transformation B.1: (collect composition)

$$\Gamma[x{:}\,f\,|\,p](\Gamma[y{:}g|q](Y)) \equiv \Gamma[x{:}\,f[x \to g[y \to x]]\,|\,(p[x \to g[y \to x]] \wedge q[y \to x])](Y) \quad \text{(B-1)}$$

Transformation B.2: (existential quantifier composition)

$$\exists y \in \Gamma[x{:}\,f\,|\,p](X) \bullet q \equiv \exists x \in X \bullet p \wedge q[y \to f] \quad \text{(B-2)}$$

Transformation B.3: (universal quantifier composition)

$$\forall y \in \Gamma[x{:}\,f\,|\,p](X) \bullet q \equiv \forall x \in X \bullet \neg p \vee q[y \to f] \quad \text{(B-3)}$$

Transformation B.4: (set inclusion transformation)

$$x \in X \equiv \exists v \in X \bullet (v = x) \quad \text{(B-4)}$$

2. NORMALIZATION RULES

During normalization, two stages are distinguished:

2.1 Transformation to PNF

PNF transformations are taken from [Jar84]. Where applicable, we take into account the symmetry of conjunctions and disjunctions

Transformation B.5:

$$p \wedge \exists x \in X \bullet q \equiv \exists x \in X \bullet (q \wedge p) \tag{B-5}$$

Transformation B.6:

$$p \vee \exists x \in X \bullet q \equiv \begin{cases} \exists x \in X \bullet (q \vee p) & X \neq \varnothing \\ p & X = \varnothing \end{cases} \tag{B-6}$$

Transformation B.7:

$$p \wedge \forall x \in X \bullet q \equiv \begin{cases} \forall x \in X \bullet (q \wedge p) & X \neq \varnothing \\ p & X = \varnothing \end{cases} \tag{B-7}$$

Transformation B.8:

$$p \vee \forall x \in X \bullet q \equiv \forall x \in X \bullet (q \vee p) \tag{B-8}$$

Transformation B.9:

$$p \vee \forall x \in X \bullet q \equiv \begin{cases} \forall x \in X \bullet (q \vee p) & X \neq \varnothing \\ p & X = \varnothing \end{cases} \tag{B-9}$$

Transformation B.10: (compose existential quantifier)

$$\exists x \in A \bullet p \vee \exists y \in A \bullet q \equiv \exists x \in A \bullet (p \vee q[y \rightarrow x]) \tag{B-10}$$

Transformation B.11: (compose universal quantifier)

$$\forall x \in A \bullet p(x) \wedge \forall y \in A \bullet q(x) \equiv \forall x \in A \bullet (p \wedge q[y \leftarrow x]) \tag{B-11}$$

Transformation B.12: (negated universal quantifier)

$$\neg \forall x \in X \bullet p \equiv \exists x \in X \bullet \neg p \tag{B-12}$$

Transformation B.13: (negated existential quantifier)

$$\neg \exists x \in X \bullet p \equiv \forall x \in X \bullet \neg p \tag{B-13}$$

2.2 Transformation to MNF

The predicates transformed into MNF can be transformed to Miniscope Normal Form. The following rules are modified versions of [Bry89]. We denote the set of free variables in expression *p* as *FV(p)*.

Transformation B.14: (double negation)

$$\neg\neg p \equiv p \tag{B-14}$$

Transformation B.15: (de Morgan's)

$$\neg(p \wedge q) \equiv \neg p \vee \neg q \tag{B-15}$$

Transformation B.16: (de Morgan's)

$$\neg(p \vee q) \equiv \neg p \wedge \neg q \tag{B-16}$$

Transformation B.17:

$$\forall x \in X \bullet p \equiv \neg\exists x \in X \bullet \neg p \tag{B-17}$$

Transformation B.18:

$$\exists x \in X \bullet p \equiv p \qquad x \notin FV(p) \tag{B-18}$$

Transformation B.19:

$$\exists x \in X \bullet (p \wedge q[x]) \equiv p \wedge \exists x \in X \bullet q \qquad x \notin FV(p) \tag{B-19}$$

Transformation B.20:

$$\exists x \in X \bullet (p \vee q[x]) \equiv \begin{cases} p \vee \exists x \in X \bullet q & x \notin FV(p), X \neq \emptyset \\ \text{false} & X = \emptyset \end{cases} \tag{B-20}$$

3. TRANSLATION RULES

3.1 Unnesting rules

For completeness we quote unnesting rules already presented in Chapter III:

Transformation B.21: (semijoin introduction)

$$\Gamma[x : f \mid (\exists y \in Y \bullet p[x, y])](X) \equiv \alpha[x : f](X \text{ semijoin}(x, y : p)\ Y) \tag{B-21}$$

Transformation B.22: (join introduction)

$$\Gamma[x : f \mid (\exists z \in \sigma[y : p[x, y]](Y) \bullet q)](X) \equiv \Gamma[v : f[x \to v_x] \mid q[x \to v_x, z \to v_y]](X \text{ join}(x, y ; p)\ Y) \quad \text{(B-22)}$$

Transformation B.23: (product introduction)

$$\Gamma[x : f \mid \exists y \in Y \bullet p](X) \equiv \Gamma[v : f[x \to v_x] \mid p[x \to v_x, y \to v_y]](X \times Y) \quad \text{(B-23)}$$

Transformation B.24: (antijoin introduction)

$$\Gamma[x : f \mid (\neg \exists y \in Y \bullet p[x, y])](X) \equiv \alpha[x : f](X \text{ antijoin}(x, y ; p)\ Y) \quad \text{(B-24)}$$

Transformation B.25: (division introduction):

$$\Gamma[x : f \mid (\neg \exists y \in Y \bullet p)](X) \equiv \alpha[x : f]((\Gamma[v : v \mid \neg p[x \to v_x, y \to v_y]](X \times Y)) \div Y) \quad \text{(B-25)}$$

Transformation B.26: (nestjoin introduction)

$$\Gamma[x : (g \mid q)[x, \Gamma[y : (f \mid p)[x, y]](Y)]](X) \equiv \Gamma[v : (g \mid q)[x \to v_x, \Gamma[y : f \mid p](Y) \to v\ .\ ys]](X\ \Delta_{x, y\ ;\ f \mid p\ ;\ ys}\ Y) \quad \text{(B-26)}$$

Transformation B.27: (nested Cartesian product introduction)

$$\Gamma[x : (f \mid p)[x, Y]](X) \equiv \Gamma[v : (f \mid p)[x \to v_x, Y \to v\ .\ ys]](X\ \text{❖}_{ys}\ Y) \quad \text{(B-27)}$$

3.2 Splitting rules

The splitting rules are divided into the rules for splitting collect predicates and functions:

3.2.1 Splitting predicates

Transformation B.28: (conjunctive collect predicate)

$$\Gamma[x : f \mid p \wedge q](X) \equiv \Gamma[x : f \mid q](\sigma[x : p](X)) \quad \text{(B-28)}$$

Transformation B.29: (disjunctive collect predicate)

$$\Gamma[x : f \mid p \vee q](X) \equiv \Gamma[x : f \mid p](X) \cup \Gamma[x : f \mid q](Y) \quad \text{(B-29)}$$

Transformation B.30: (conjunctive quantifier matrix)

$$\exists x \in X \bullet p \wedge q \equiv \exists x \in \sigma[x : p](X) \bullet q \quad \text{(B-30)}$$

Transformation B.31: (disjunctive quantifier matrix)

$$\exists x \in X \bullet p \vee q \equiv \exists x \in X \bullet p \vee \exists x \in X \bullet q \quad \text{(B-31)}$$

Transformation B.32:

$$\neg \exists x \in X \bullet p \vee q \equiv \neg \exists x \in X \bullet p \wedge \neg \exists x \in X \bullet q \quad \text{(B-32)}$$

3.2.2 Splitting functions

Similarly to collect predicates, collect functions can be split using the following rules:

Transformation B.33:

$$\Gamma[x : f[g[x]] \mid p](X) \equiv \alpha[x : f[g \rightarrow x]](\Gamma[x : g \mid p](X)) \quad \text{(B-33)}$$

In the above rule, *x* does not occur free in *f* outside of *g*.

Transformation B.34:

$$\Gamma[x : f[g_1[x], \ldots, g_n[x]] \mid p](X) \equiv \quad \text{(B-34)}$$
$$\alpha[x : f[g_1 \rightarrow x\,.\,a_1, \ldots, g_n \rightarrow x\,.\,a_n]](\Gamma[x : \langle a_1 = g_1, \ldots, a_n = g_n \rangle \mid p](X))$$

In the above left-hand expression, *x* does not occur free in *f* outside of $g_i[x]$.

Transformation B.35:

$$\Gamma[x : f \mid p[g_1[x], \ldots, g_n[x]]](X) \equiv \quad \text{(B-35)}$$
$$\Gamma[v : f[x \rightarrow v_X] \mid p[g_1 \rightarrow v\,.\,a_1, \ldots, g_n \rightarrow v\,.\,a_n]]$$
$$(\alpha[x : x \text{ except } \langle a_1 = g_1, \ldots, a_n = g_n \rangle](X))$$

In the above left-hand expression, *x* does not occur in *p* outside of $g_i[x]$.

Transformation B.36:

$$\exists x \in X \bullet p[g[x]] \equiv \exists v \in \alpha[x : g](X) \bullet p[g \rightarrow v] \quad \text{(B-36)}$$

In the above left-hand expression, *x* does not occur in *p* outside of *g[x]*.

Transformation B.37:

$$\exists x \in X \bullet p[g_1[x], \ldots, g_n[x]] \equiv \quad \text{(B-37)}$$
$$\exists v \in \alpha[x : \langle a_1 = g_1, \ldots, a_n = g_n \rangle](X) \bullet p[g_1 \rightarrow v\,.\,a_1, \ldots, g_n \leftarrow v\,.\,a_n]$$

In the above left-hand expression, *x* does not occur in *p* outside of $g_i[x]$.

3.3 Pushing rules

These rules are used to push expressions to join operands.

Transformation B.38:

$$\sigma[v : p[v_X]](X \text{ join}(x, y\,;\, q)\ Y) \equiv \sigma[v : p](X) \text{ join}(x, y\,;\, q)\ Y \quad \text{(B-38)}$$

The transformation is applicable if *p* contains references to attributes of *X*, but not *Y*.

Transformation B.39:

$$\sigma[v : p[v_Y]](X \text{ join}(x, y\,;\, q)\ Y) \equiv X \text{ join}(x, y\,;\, q)\ (\sigma[v : p[v_Y]](Y)) \quad \text{(B-39)}$$

The transformation is applicable if *p* contains references to attributes of *Y*, but not *X*.

Transformation B.40:

$$\sigma[v : p[v_X, v_Y]](X \text{ join}(x, y ; q)\ Y) \equiv X \text{ join}(x, y ; (q \wedge p[v_X \rightarrow x, v_Y \rightarrow y]))\ Y \quad \text{(B-40)}$$

Transformation B.41:

$$\sigma[v : p](X \times Y) \equiv \sigma[v : p](X \text{ join}(x, y ; \text{true})\ Y) \quad \text{(B-41)}$$

This transformation is introduced to allow pushing of expressions to operators according to the rules above also for Cartesian products.

Transformation B.42:

$$\sigma[v : p](X \text{ semijoin}(x, y ; q)\ Y) \equiv \sigma[v : p](X) \text{ semijoin}(x, y ; q)\ Y \quad \text{(B-42)}$$

Transformation B.43:

$$\sigma[v : p](X \text{ antijoin}(x, y ; q)\ Y) \equiv \sigma[v : p](X) \text{ antijoin}(x, y ; q)\ Y \quad \text{(B-43)}$$

Transformation B.44:

$$\sigma[v : p](X\ \Delta_{x, y ; f \mid q ; ys}\ Y) \equiv \sigma[v : p](X)\ \Delta_{x, y ; f \mid q ; ys}\ Y \quad \text{(B-44)}$$

In the above rule, *p* references attributes of X, and does not reference *v.ys*.

Transformation B.45:

$$\pi_A(X \text{ semijoin}(x, y ; p)\ Y) \equiv \pi_A(X) \text{ semijoin}(x, y ; p)\ Y \quad \text{(B-45)}$$

Transformation B.46:

$$\pi_A(X \text{ antijoin}(x, y ; p)\ Y) \equiv \pi_A(X) \text{ antijoin}(x, y ; p)\ Y \quad \text{(B-46)}$$

In both above rules, the set of attributes *A* is included within the set of attributes referenced in the predicate *p*. This is denoted as $A \subseteq \text{Attr}(p)$.

Transformation B.47:

$$\pi_{AB}(X \times Y) \equiv \pi_A(X) \times \pi_B(Y) \quad \text{(B-47)}$$

We denote the set of attributes of table *X* as *SC(X)*. In this case, the transformation is applicable if $A \subseteq \text{SC}(X)$ and $B \subseteq \text{SC}(Y)$.

Transformation B.48:

$$\pi_{AB}(X \text{ join}(x, y ; p)\ Y) \equiv \pi_A(X) \text{ join}(x, y ; p)\ \pi_B(Y) \quad \text{(B-48)}$$

The applicability conditions are: $A \subseteq \text{SC}(X)$, $B \subseteq \text{SC}(Y)$ and $\text{Attr}(p) \subseteq A \cup B$.

Transformation B.49:

$$\pi_{A\mathrm{ys}}(X \;\Delta_{x,y\,;\,f\,|\,p\,;\,ys}\; Y) \equiv \pi_A(X) \;\Delta_{x,y\,;\,f\,|\,p\,;\,ys}\; Y \tag{B-49}$$

The applicability conditions are: $A \subseteq \mathrm{SC}(X)$ and $\mathrm{Attr}_X(f) \cup \mathrm{Attr}_X(p) \subseteq A$. Notation $\mathrm{Attr}_X(f)$ denotes attributes of operand X referenced within expression f. The second applicability condition determines therefore that the projection must preserve those attributes of X which are needed for evaluation of the collect function and predicate. On the left hand-side, the nestjoin attribute *ys* must be present in the projection list.

Appendix C. Example optimizations

The unnesting strategy presented in Chapter III has been developed during testing of the implemented optimizer using various alternative strategies. In the testing, we have used a set of example queries containing various forms of nesting and variable dependencies. In this appendix, we present the query set, and list the transformations applied by the optimizer to lead to the optimized expressions. Where applicable, we examine alternative strategies and show their influences on resulting optimized expressions.

1. TEST QUERIES

The test query set has been designed to emulate diverse types of nesting and variable dependencies. The query set consists of 4 types of queries. Within each of the types, multiple example queries are obtained by substituting negations, conjunctions and disjunctions in the predicates. The following types are included in the test set (in the formulas, $\boldsymbol{c}$ and $\boldsymbol{c_1}$ denotes conjunction or disjunction, and $\boldsymbol{Q}$ denotes quantifier $\exists$ or $\neg\exists$:

Type A: (Chain)

$$\alpha[x : \sigma[y : p[x, y] \quad \boldsymbol{c} \quad \boldsymbol{Q}z \in Z \bullet r[y, z]](Y)](X) \tag{C-1}$$

Type B:(Tree I)

$$\alpha[x : \sigma[y : p[x, y] \quad \boldsymbol{c} \quad \boldsymbol{Q}z \in Z \bullet q[x, z]](Y)](X) \tag{C-2}$$

Type C:(Tree II)

$$\alpha[x : \sigma[y : \boldsymbol{Q}z \in Z \bullet (r[y, z] \quad \boldsymbol{c} \quad q[x, z])](Y)](X) \tag{C-3}$$

Type D:(Cyclic)

$$\alpha[x : \sigma[y : p[x, y] \;\; \boldsymbol{c} \;\; \boldsymbol{Q}z \in Z \bullet (r[y, z] \;\; \boldsymbol{c_1} \;\; q[x, z])](Y)](X) \tag{C-4}$$

For each of the types, there are different possible queries depending on the quantifiers and connectives used. We structure the examples in 4 groups, each corresponding to values of ***c*** and ***Q***. In each group, we will have 5 queries: 1 for each of types A, B, C, and two of type D (one for conjunction as $\boldsymbol{c_1}$ and one for disjunction as $\boldsymbol{c_1}$). This leads to the total of 20 queries in the test set. In the following sections, we present the transformations applied by the optimizer during optimization of those queries. Variables v_i represent temporary variables introduced in the query by the optimizer. Notation v_{1x} represents projection of the (nest)join result on attributes of operand *X*.

2. EXAMPLE QUERIES

2.1 c = disjunction, Q = existential

Query C.1: (Type A)

$$\begin{aligned}
&\alpha[x : \sigma[y : p[x, y] \vee \exists z \in Z \bullet r[y, z]](Y)](X) \equiv \\
&\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \exists z \in Z \bullet r[y, z]](Y)](X) \equiv \\
&\alpha[x : \sigma[y : p[x, y]](Y) \cup (Y \text{ semijoin}(y, z\,;\, r[y, z])\; Z)](X) \equiv \\
&\alpha[v_1 : v_1 \,.\, v_2 \cup (Y \text{ semijoin}(y, z\,;\, r[y, z])\; Z)](X \;\; \Delta_{x, y\,;\, y \,|\, p[x, y]\,;\, v_2} \; Y) \equiv \\
&\alpha[v_1 : v_1 \,.\, v_2 \cup v_1 \,.\, v_3]((X \;\; \Delta_{x, y\,;\, y \,|\, p(x, y)\,;\, v_2} \; Y) \;\; \mathbin{❖}_{v_3} (Y \text{ semijoin}(y, z\,;\, r(y, z))\; Z))
\end{aligned} \tag{C-5}$$

During processing, the predicate is split, and joins are introduced: first the semijoin and the nestjoin. Note that range split cannot be avoided: a Cartesian product cannot be introduced using rule III.3 because the collect and the quantifier are not on adjacent nesting levels. Because the semijoin is nested within the collect function, a nested Cartesian product is created.

Query C.2: (Type B)

$$\begin{aligned}
&\alpha[x : \sigma[y : p[x, y] \vee \exists z \in Z \bullet q[x, z]](Y)](X) \equiv \\
&\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \exists z \in Z \bullet q[x, z]](Y)](X) \equiv \\
&\alpha[v_1 : v_1 \,.\, v_2 \cup \sigma[y : \exists z \in Z \bullet q[x \to v_{1X}, z]](Y)](X \;\; \Delta_{x, y\,;\, y \,|\, p[x, y]\,;\, v_2} \; Y) \equiv \\
&\alpha[v_1 : v_1 \,.\, v_2 \cup \Gamma[v_3 : v_{3y} \,|\, q[x \to v_{1X}, z \to v_{3Z}]](Y \times Z)](X \;\; \Delta_{x, y\,;\, y \,|\, p[x, y]\,;\, v_2} \; Y) \equiv \\
&\alpha[v_1 : v_1 \,.\, v_2 \cup v_1 \,.\, v_5]((X \;\; \Delta_{x, y\,;\, y \,|\, p[x, y]\,;\, v_2} \; Y) \;\; \Delta_{v_4, v_3\,;\, v_{3Y} \,|\, q[x \to v_{4X}, z \to v_{3Z}]\,;\, v_5} \; (Y \times Z))
\end{aligned} \tag{C-6}$$

As in the above query, the predicate has to be split, and no product transformations can be applied.

The result can be improved by introduction of markjoin transformations III.15 and III.16. If those transformations were to be integrated in our framework as described in Chapter III, then the optimization would proceed as follows:

$$\begin{array}{l}\alpha[x : \sigma[y : p[x, y] \vee \exists z \in Z \bullet q[x, z]](Y)](X) \equiv \\ \alpha[v_1 : \sigma[y : p[x \to v_{1X}, y] \vee v_1 \,.\, v_2](Y)](X \;\; \perp_{x,\, z\,;\, q[x,\, z]\,;\, v_2} \;\; Z) \equiv \\ \alpha[v_4 : (v_4 \,.\, v_3)]((X \;\; \perp_{x,\, z\,;\, q[x,\, z]\,;\, v_2} \;\; Z) \;\; \Delta_{v_1,\, y\,;\, y \,|\, p[x \to v_{1X},\, y] \vee v_1 \,.\, v_2\,;\, v_3} \;\; Y)\end{array} \tag{C-7}$$

The alternative result is better than the previous: there is only one reference to collection *Y*, and the result does not contain a Cartesian product.

Query C.3: (Type C)

$$\begin{array}{l}\alpha[x : \sigma[y : \exists z \in Z \bullet (r[y, z] \vee q[x, z])](Y)](X) \equiv \\ \alpha[x : \Gamma[v_1 : v_{1Y} \,|\, (r[y \to v_{1Y},\, x,\, z \to v_{1Z}] \vee q[x,\, z \to v_{1Z}])](Y \times Z)](X) \equiv \\ \alpha[v_2 : (v_2 \,.\, v_3)](X \;\; \Delta_{x,\, v_1\,;\, v_{1Y} \,|\, r[y \to v_{1Y},\, x,\, z \to v_{1Z}] \vee q[x,\, z \to v_{1Z}]\,;\, v_3} (Y \times Z))\end{array} \tag{C-8}$$

In the transformation, range split has been prevented by introduction of the Cartesian product. To illustrate the benefits of our strategy, we show an alternative optimization of the query: one in which the predicate is fully split before joins are introduced.

$$\begin{array}{l}\alpha[x : \sigma[y : \exists z \in Z \bullet (r[y, z] \vee q[x, z])](Y)](X) \equiv \\ \alpha[x : \sigma[y : \exists z \in Z \bullet r[y, z] \vee \exists z \in Z \bullet q[x, z]](Y)](X) \equiv \\ \alpha[x : \sigma[y : \exists z \in Z \bullet r[y, z]](Y) \cup \sigma[y : \exists z \in Z \bullet q[x, z]](Y)](X) \equiv \\ \alpha[x : (Y \text{ semijoin}(y, z\,;\, r[y, z])\; Z) \cup \sigma[y : \exists z \in Z \bullet q[x, z]](Y)](X) \equiv \\ \alpha[x : (Y \text{ semijoin}(y, z\,;\, r[y, z])\; Z) \cup \Gamma[v_1 : v_{1Y} \,|\, q[x, z \to v_{1Z}]](Y \times Z)](X) \equiv \\ \alpha[v_2 : (Y \text{ semijoin}(y, z\,;\, r[y, z])\; Z) \cup v_2 \,.\, v_3](X \;\; \Delta_{x,\, v_1\,;\, v_{1y} \,|\, q[x,\, z \to v_{1Z}]\,;\, v_3} (Y \times Z)) \equiv \\ \alpha[v_2 : v_2 \,.\, v_4 \cup v_2 \,.\, v_3]((X \;\; \Delta_{x,\, v_1\,;\, v_{1Y} \,|\, q[x,\, z \to v_{1Z}]\,;\, v_3} (Y \times Z)) \;\; ❖_{v_4} (Y \text{ semijoin}(y, z\,;\, r[y, z])\; Z))\end{array} \tag{C-9}$$

The result is worse than in our strategy: besides the nestjoin between X and Cartesian product, we additionally have to execute the semijoin and nested Cartesian product.

Query C.4: (Type D)

$$\begin{array}{l}\alpha[x : \sigma[y : p[x, y] \vee \exists z \in Z \bullet (r[y, z] \vee q[x, z])](Y)](X) \equiv \\ \alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \exists z \in Z \bullet (r[y, z] \vee q[x, z])](Y)](X) \equiv \\ \alpha[v_1 : v_1 \,.\, v_2 \cup \sigma[y : \exists z \in Z \bullet (r[y, z] \vee q[x \to v_1, z])](Y)](X \;\; \Delta_{x,\, y\,;\, y \,|\, p[x,\, y]\,;\, v_2} \;\; Y) \equiv \\ \alpha[v_1 : v_1 \,.\, v_2 \cup \Gamma[v_3 : v_{3Y} \,|\, (r[y \to v_{3Y},\, z \to v_{3Z}] \vee q[x \to v_1,\, z \to v_{3Z}])](Y \times Z)] \\ (X \;\; \Delta_{x,\, y\,;\, y \,|\, p[x,\, y]\,;\, v_2} \;\; Y) \equiv \\ \alpha[v_1 : v_1 \,.\, v_2 \cup v_1 \,.\, v_5] \\ ((X \;\; \Delta_{x,\, y\,;\, y \,|\, p[x,\, y]\,;\, v_2} \;\; Y) \;\; \Delta_{v_4,\, v_3\,;\, v_{3Y} \,|\, (r[y \to v_{3Y},\, z \to v_{3Z}] \vee q[x \to v_1,\, z \to v_{3Z}])\,;\, v_5} (Y \times Z))\end{array} \tag{C-10}$$

As in the previous query, we consider the optimization using an alternative strategy: applying the join transformations only after expressions have been fully split. We do not list all splitting transformations, but begin directly with the fully split expression:

$$\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \exists z \in Z \bullet q[x, z]](Y) \cup \sigma[y : \exists z \in Z \bullet r[y, z]](Y)](X) \equiv \quad \text{(C-11)}$$
$$\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \exists z \in Z \bullet q[x, z]](Y) \cup (Y \text{ semijoin}(y, z\,;\, r[y, z])\ Z)](X) \equiv$$
$$\alpha[x : \sigma[y : p[x, y]](Y) \cup \Gamma[v_1 : v_{1y} \mid q[x, z \to v_{1Z}]](Y \times Z) \cup (Y \text{ semijoin}(y, z\,;\, r[y, z])\ Z)](X) \equiv$$
$$\alpha[v_2 : v_2 \,.\, v_3 \cup \Gamma[v_1 : v_{1y} \mid q[x \to v_{2X}, z \to v_{1Z}]](Y \times Z) \cup (Y \text{ semijoin}(y, z\,;\, r[y, z])\ Z)]$$
$$(X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_3}\ Y) \equiv$$
$$\alpha[v_5 : v_5 \,.\, v_3 \cup v_5 \,.\, v_4 \cup (Y \text{ semijoin}(y, z\,;\, r[y, z])\ Z)]$$
$$((X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_3}\ Y)\ \ \Delta_{v_2, v_1\,;\, v_{1Y} \mid q[x \to v_{2X}, z \to v_{1Z}]\,;\, v_4}(Y \times Z)) \equiv$$
$$\alpha[v_5 : v_5 \,.\, v_3 \cup v_5 \,.\, v_4 \cup v_5 \,.\, v_6]$$
$$(((X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_3}\ Y)\ \ \Delta_{v_2, v_1\,;\, v_{1Y} \mid q[x \to v_{2X}, z \to v_{1z}]\,;\, v_4}(Y \times Z))\ \ \ltimes\!\!\!\!\bullet_{v_6}\ (Y \text{ semijoin}(y, z\,;\, r[y, z])\ Z))$$

As in the previous example, full splitting has led to a worse result: additional semijoin and nested Cartesian product has to be introduced.

Query C.5: (Type D)

$$\alpha[x : \sigma[y : p[x, y] \vee \exists z \in Z \bullet (r[y, z] \wedge q[x, z])](Y)](X) \equiv \quad \text{(C-12)}$$
$$\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \exists z \in Z \bullet (r[y, z] \wedge q[x, z])](Y)](X) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup \sigma[y : \exists z \in Z \bullet (r[y, z] \vee q[x \to v_{1X}, z])](Y)](X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup \sigma[y : \exists z \in \sigma[v_3 : r[y, z \to v_3]](Z) \bullet q[x \to v_{1X}, z]](Y)](X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup \Gamma[v_4 : v_{4y} \mid q[x \to v_{1X}, z \to v_{4Z}]]$$
$$(Y \text{ join}(y, v_3\,;\, r[y, z \to v_3])\ Z)](X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup v_1 \,.\, v_6]$$
$$((X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y)\ \ \Delta_{v_5, v_4\,;\, v_{5Y}\,;\, q[x \to v_{5X}, z \to v_{4Z}]\,;\, v_6}(Y \text{ join}(y, v_3\,;\, r[y, z \to v_3])\ Z))$$

During optimization, we first split the disjunction and then introduce a nestjoin. We subsequently perform a split (guided by the planned join between *Y* and *Z*) and introduce a join. The join is finally brought to the top level. The alternative strategy is full splitting: performing both splits together and then introduce joins. This, however, would require planning all possible joins, so that proper choices can be made in conjunctive splitting. In our strategy, we return to unnesting after every split, so we only have to plan one step ahead.

2.2 c = disjunction, Q = negated existential

Query C.6: (Type A)

$$\alpha[x : \sigma[y : p[x, y] \vee \neg\exists z \in Z \bullet r[y, z]](Y)](X) \equiv \quad \text{(C-13)}$$
$$\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \neg\exists z \in Z \bullet r[y, z]](Y)](X) \equiv$$
$$\alpha[x : \sigma[y : p[x, y]](Y) \cup (Y \text{ antijoin}(y, z\,;\, r[y, z])\ Z)](X) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup (Y \text{ antijoin}(y, z\,;\, r[y, z])\ Z)](X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup v_1 \,.\, v_3]((X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y)\ \mathbin{❖}_{v_3}\ (Y \text{ antijoin}(y, z\,;\, r[y, z])\ Z))$$

Query C.7: (Type B)

$$\alpha[x : \sigma[y : p[x, y] \vee \neg\exists z \in Z \bullet q[x, z]](Y)](X) \equiv \quad \text{(C-14)}$$
$$\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \neg\exists z \in Z \bullet q[x, z]](Y)](X) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup \sigma[y : \neg\exists z \in Z \bullet q[x \to v_{1X}, z]](Y)](X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup (\sigma[v_3 : \neg q[x \to v_{1X}, z \to v_{3Z}]](Y \times Z)) \div Z](X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup (v_1 \,.\, v_5 \div Z)]((X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y)\ \Delta_{v_4, v_3\,;\, v_{3Y} \mid \neg q[x \to v_{4X}, z \to v_{3Z}]\,;\, v_5}\ (Y \times Z)) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup (v_1 \,.\, v_5 \div v_1 \,.\, v_6)]$$
$$(((X\ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y)\ \Delta_{v_4, v_3\,;\, v_{3Y} \mid \neg q[x \to v_{4X}, z \to v_{3Z}]\,;\, v_5}\ (Y \times Z))\ \mathbin{❖}_{v_6}\ Z)$$

The resulting expression suffers from the lack of markjoin rules, which would allow us to avoid expensive products and divisions. If markjoin rules were to be added, the processing would be as follows:

$$\alpha[x : \sigma[y : p[x, y] \vee \neg\exists z \in Z \bullet q[x, z]](Y)](X) \equiv \quad \text{(C-15)}$$
$$\alpha[v_1 : \sigma[y : p[x \to v_{1X}, y] \vee \neg(v_1 \,.\, v_2)](Y)](X\ \perp_{x, z\,;\, q[x, z]\,;\, v_2}\ Z) \equiv$$
$$\alpha[v_3 : (v_3 \,.\, v_4)]((X\ \perp_{x, z\,;\, q[x, z]\,;\, v_2}\ Z)\ \Delta_{v_1, y\,;\, y \mid p[x \to v_{1X}, y] \vee \neg(v_1 \,.\, v_2)\,;\, v_4}\ Y)$$

The result does not contain multiple references to the same tables, and no expensive product operators.

Query C.8: (Type C)

$$\alpha[x : \sigma[y : \neg\exists z \in Z \bullet (r[y, z] \vee q[x, z])](Y)](X) \equiv \quad \text{(C-16)}$$
$$\alpha[x : (\sigma[v_1 : \neg(r[y \to v_{1Y}, z \to v_{1Z}] \vee q[x, z \to v_{1Z}])](Y \times Z) \div Z)](X) \equiv$$
$$\alpha[v_2 : (v_2 \,.\, v_3 \div Z)](X\ \Delta_{x, v_1\,;\, v_1 \mid \neg(r[y \to v_{1Y}, z \to v_{1Z}] \vee q[x, z \to v_{1Z}])\,;\, v_3}\ (Y \times Z)) \equiv$$
$$\alpha[v_2 : (v_2 \,.\, v_3 \div v_2 \,.\, v_4)]((X\ \Delta_{x, v_1\,;\, v_1 \mid \neg(r[y \to v_{1Y}, z \to v_{1Z}] \vee q[x, z \to v_{1Z}])\,;\, v_3}\ (Y \times Z))\ \mathbin{❖}_{v_4}\ Z)$$

The above transformations avoid the range split by performing product transformation on a partially split expression. Alternative strategy involves full splitting of predicates:

$$
\begin{aligned}
&\mathfrak{x}[x : \sigma[y : \neg\exists z \in Z \bullet (r[y, z] \vee q[x, z])](Y)](X) \equiv \\
&\mathfrak{x}[x : \sigma[y : \neg\exists z \in Z \bullet r[y, z] \wedge \neg\exists z \in Z \bullet q[x, z]](Y)](X) \equiv \\
&\mathfrak{x}[x : \sigma[y : \neg\exists z \in Z \bullet q[x, z]](\sigma[y : \neg\exists z \in Z \bullet r[y, z]](Y))](X) \equiv \\
&\mathfrak{x}[x : \sigma[y : \neg\exists z \in Z \bullet q[x, z]](Y \text{ antijoin}(y, z ; r[y, z])\ Z)](X) \equiv \\
&\mathfrak{x}[v_1 : \sigma[y : \neg\exists z \in Z \bullet q[x \to v_{1X}, z]](v_1 \,.\, v_2)](X \ \mathbin{❖_{v_2}}\ (Y \text{ antijoin}(y, z ; r[y, z])\ Z)) \equiv \\
&\mathfrak{x}[v_1 : \sigma[y : \neg\exists z \in v_1 \,.\, v_3 \bullet q[x \to v_{1X}, z]](v_1 \,.\, v_2)]((X \ \mathbin{❖_{v_2}}\ (Y \text{ antijoin}(y, z ; r[y, z])\ Z)) \ \mathbin{❖_{v_3}}\ Z)
\end{aligned}
\tag{C-17}
$$

The achieved result is not significantly better than the one achieved without the range split. The second result contains an antijoin, but also more nested Cartesian products and a quantifier in the map function. As removal of quantifiers is one of the goals of our strategy, we prefer the first result.

Query C.9: (Type D)

$$
\begin{aligned}
&\alpha[x : \sigma[y : p[x, y] \vee \neg\exists z \in Z \bullet (r[y, z] \vee q[x, z])](Y)](X) \equiv \\
&\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \neg\exists z \in Z \bullet (r[y, z] \vee q[x, z])](Y)](X) \equiv \\
&\alpha[v_1 : v_1 \,.\, v_2 \cup (\sigma[v_3 : \neg(r[y \to v_{3Y}, z \to v_{3Z}] \vee q[x \to v_{1X}, z \to v_{3Z}])](Y \times Z) \div Z)] \\
&(X \ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y) \equiv \\
&\alpha[v_1 : v_1 \,.\, v_2 \cup (v_1 \,.\, v_5 \div Z)] \\
&((X \ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y) \ \ \Delta_{v_4, v_3\,;\, v_{3Y} \mid \neg(r[y \to v_{3Y}, z \to v_{3Z}] \vee q[x \to v_{4X}, z \to v_{3Z}])\,;\, v_5} (Y \times Z)) \equiv \\
&\alpha[v_1 : v_1 \,.\, v_2 \cup (v_1 \,.\, v_5 \div v_1 \,.\, v_6)] \\
&(((X \ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y) \ \ \Delta_{v_4, v_3\,;\, v_{3Y} \mid \neg(r[y \to v_{3Y}, z \to v_{3Z}] \vee q[x \to v_{4X}, z \to v_{3Z}])\,;\, v_5} (Y \times Z)) \ \mathbin{❖_{v_6}}\ Z)
\end{aligned}
\tag{C-18}
$$

In the processing, we first introduce a union. Then, to avoid the splitting of the quantifier predicate, we introduce a division of a product, and a nestjoin of collections *X* and *Y*. The product is then brought to the top level and finally the reference to *Z* is removed by introduction of the nested Cartesian product.

Alternative strategy involves full splitting:

$$
\begin{aligned}
&\alpha[x : \sigma[y : p[x, y] \vee \neg\exists z \in Z \bullet r[y, z] \vee q[x, z]](Y)](X) \equiv \\
&\alpha[x : \sigma[y : p[x, y]](Y) \cup \sigma[y : \neg\exists z \in Z \bullet r[y, z] \wedge \neg\exists z \in Z \bullet q[x, z]](Y)](X) \equiv \\
&\alpha[v_1 : v_1 \,.\, v_2 \cup \sigma[y : \neg\exists z \in Z \bullet r[y, z] \wedge \neg\exists z \in Z \bullet q[x \to v_{1X}, z]](Y)](X \ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y) \equiv \\
&\alpha[v_1 : v_1 \,.\, v_2 \cup \sigma[y : \neg\exists z \in Z \bullet q[x \to v_{1X}, z]](\sigma[y : \neg\exists z \in Z \bullet r[y, z]](Y))](X \ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y) \equiv \\
&\alpha[v_1 : v_1 \,.\, v_2 \cup \sigma[y : \neg\exists z \in Z \bullet q[x \to v_{1X}, z]](Y \text{ antijoin}(y, z ; r[y, z])\ Z)](X \ \Delta_{x, y\,;\, p[x, y]\,;\, v_2}\ Y) \equiv \\
&\alpha[v_1 : v_1 \,.\, v_2 \cup \sigma[y : \neg\exists z \in Z \bullet q[x \to v_{1X}, z]](v_1 \,.\, v_3)] \\
&((X \ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y) \ \mathbin{❖_{v_3}}\ (Y \text{ antijoin}(y, z ; r[y, z])\ Z)) \equiv \\
&\alpha[v_1 : v_1 \,.\, v_2 \cup \sigma[y : \neg\exists z \in v_1 \,.\, v_4 \bullet q[x \to v_{1X}, z]](v_1 \,.\, v_3)] \\
&(((X \ \Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\ Y) \ \mathbin{❖_{v_3}}\ (Y \text{ antijoin}(y, z ; r[y, z])\ Z)) \ \mathbin{❖_{v_4}}\ Z)
\end{aligned}
\tag{C-19}
$$

It is difficult to choose the expressions based on the operator heuristics only. Both expressions contain a nestjoin (*X*,*Y*) and a nested Cartesian product with the *Z* collection. In the first result, the nestjoin result is nestjoined with Cartesian product (*Y*,*Z*), while in the second we have a

nested Cartesian product with the antijoin (*Y,Z*). The antijoin will have a better performance than a Cartesian product, but the nestjoin will have better performance than the Cartesian product. This leads to the conclusion that both first and second results have map **collections** of comparable performance. However, the map functions of both results are differing: in the first result, we have a division, while in the second a negated quantifier is present. As our goal is removal of nested quantifiers from the query expression, we prefer the first result above the second: it contains less calculus-level elements than the second one.

Query C.10: (Type D)

$$\alpha[x : \sigma[y : p[x,y] \vee \neg\exists z \in Z \bullet (r[y,z] \wedge q[x,z])](Y)](X) \equiv \quad \text{(C-20)}$$
$$\alpha[x : \sigma[y : p[x,y]](Y) \cup \sigma[y : \neg\exists z \in Z \bullet (r[y,z] \wedge q[x,z])](Y)](X) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup \sigma[y : \neg\exists z \in Z \bullet (r[y,z] \wedge q[x \to v_{1X}, z])](Y)](X \;\Delta_{x,y\,;\,y \mid p[x,y]\,;\,v_2}\; Y) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup \sigma[y : \neg\exists z \in \sigma[v_3 : q[x \to v_{1X}, z \to v_3]](Z) \bullet r[y,z]](Y)](X \;\Delta_{x,y\,;\,y \mid p[x,y]\,;\,v_2}\; Y) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup \sigma[y : \neg\exists z \in v_1 \,.\, v_5 \bullet r[y,z]](Y)]$$
$$((X \;\Delta_{x,y\,;\,y \mid p[x,y]\,;\,v_2}\; Y) \;\Delta_{v_4, v_3\,;\,v_{3Z} \mid q[x \to v_{4X}, z \to v_3]\,;\,v_5}\; Z) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup (Y \text{ antijoin}(y, z\,;\, r[y,z])\; v_1 \,.\, v_5)]$$
$$((X \;\Delta_{x,y\,;\,y \mid p[x,y]\,;\,v_2}\; Y) \;\Delta_{v_4, v_3\,;\,v_{3Z} \mid q[x \to v_{4X}, z \to v_3]\,;\,v_5}\; Z) \equiv$$
$$\alpha[v_1 : v_1 \,.\, v_2 \cup (v_1 \,.\, v_6 \text{ antijoin}(y, z\,;\, r[y,z])\; v_1 \,.\, v_5)]$$
$$(((X \;\Delta_{x,y\,;\,y \mid p[x,y]\,;\,v_2}\; Y) \;\Delta_{v_4, v_3\,;\,v_{3Z} \mid q[x \to v_{4X}, z \to v_3]\,;\,v_5}\; Z) \;\text{❖}_{v_6}\; Y)$$

In this query, we first split the disjunction, and introduce a nestjoin (*X,Y*). Then, the quantifier predicate is split, as to introduce a nestjoin with *Z*. The set-valued attributes introduced in the nestjoins are then antijoined. We note that it is not beneficial to put the predicate *r* in the selection predicate during split, because no join (*Y,Z*) can be introduced. According to our splitting strategy, the expression is then traversed from the top level to detect the first collection whose iteration variable participates in *q* or *r*. The *X* collection is the first one: that's why expression *q(x,z)* is put in the selection predicate.

2.3 c = conjunction, Q = existential

Query C.11: (Type A)

$$\alpha[x : \sigma[y : p[x,y] \wedge \exists z \in Z \bullet r[y,z]](Y)](X) \equiv \quad \text{(C-21)}$$
$$\alpha[x : \sigma[y : p[x,y]](\sigma[y : \exists z \in Z \bullet r[y,z]](Y))](X) \equiv$$
$$\alpha[x : \sigma[y : p[x,y]](Y \text{ semijoin}(y, z\,;\, r[y,z])\; Z)](X) \equiv$$
$$\alpha[v_1 : (v_1 \,.\, v_2)](X \;\Delta_{x,y\,;\,y \mid p[x,y]\,;\,v_2}\; (Y \text{ semijoin}(y, z\,;\, r[y,z])\; Z))$$

Our splitting strategy detects the possibility to introduce the semijoin (*Y,Z*). The semijoin is then nestjoined with the collection *X*. To illustrate the benefits of guided splitting, we present the alternative optimization:

$$
\begin{aligned}
&\alpha[x : \sigma[y : p[x, y] \wedge \exists z \in Z \bullet r[y, z]](Y)](X) \equiv \\
&\alpha[x : \sigma[y : \exists z \in Z \bullet r[y, z]](\sigma[y : p[x, y]](Y))](X) \equiv \\
&\alpha[v_1 : \sigma[y : \exists z \in Z \bullet r[y, z]](v_1 . v_2)](X \;\Delta_{x, y ; y \mid p[x, y] ; v_2}\; Y) \equiv \\
&\alpha[v_1 : ((v_1 . v_2) \text{ semijoin}(y, z ; r[y, z]) \; Z)](X \;\Delta_{x, y ; y \mid p[x, y] ; v_2}\; Y) \equiv \\
&\alpha[v_1 : ((v_1 . v_2) \text{ semijoin}(y, z ; r(y, z)) \; (v_1 . v_3))]((X \;\Delta_{x, y ; y \mid p(x, y) ; v_2}\; Y) \;\text{❖}_{v_3}\; Z)
\end{aligned}
\tag{C-22}
$$

The result contains a nested semijoin and is much worse than the first result. We can observe how splitting choices influence the result of optimization.

Query C.12: (Type B)

$$
\begin{aligned}
&\alpha[x : \sigma[y : p[x, y] \wedge \exists z \in Z \bullet q[x, z]](Y)](X) \equiv \\
&\alpha[x : \sigma[y : \exists z \in Z \bullet q[x, z]](\sigma[y : p[x, y]](Y))](X) \equiv \\
&\alpha[v_1 : \sigma[y : \exists z \in Z \bullet q[x \to v_{1X}, z]](v_1 . v_2)](X \;\Delta_{x, y ; y \mid p[x, y] ; v_2}\; Y) \equiv \\
&\alpha[v_1 : \Gamma[v_3 : v_{3y} \mid q[x \to v_{1X}, z \to v_{3Z}]](v_1 . v_2 \times Z)](X \;\Delta_{x, y ; y \mid p[x, y] ; v_2}\; Y) \equiv \\
&\alpha[v_1 : \Gamma[v_3 : v_{3y} \mid q[x \to v_{1X}, z \to v_{3Z}]](v_1 . v_2 \times v_1 . v_4)]((X \;\Delta_{x, y ; y \mid p[x, y] ; v_2}\; Y) \;\text{❖}_{v_4}\; Z)
\end{aligned}
\tag{C-23}
$$

The optimization proceeds by splitting the conjunction to allow for nestjoin (*X,Y*). We note that the other splitting alternative is less profitable: it does not lead to join between *Y* and *Z*. The attribute reference $v_1 . v_2$ cannot be joined with *Z*, because the predicate *q* does not contain references to both iteration variables *z* and *y*. The only possible transformations are the introduction of a Cartesian product and removal of base table reference *Z*.

The result achieved cannot be improved through different splitting strategy. The only significant improvement is the introduction of a markjoin, which makes use of the predicate *q(x,z)*. If markjoin transformations are used, the optimization would proceed as follows:

$$
\begin{aligned}
&\alpha[x : \sigma[y : p[x, y] \wedge \exists z \in Z \bullet q[x, z]](Y)](X) \equiv \\
&\alpha[v_1 : \sigma[y : p[x \to v_{1X}, y] \wedge v_1 . v_2](Y)](X \;\perp_{x, z ; q[x, z] ; v_2}\; Z) \equiv \\
&\alpha[v_4 : v_4 : v_3]((X \;\perp_{x, z ; q[x, z] ; v_2}\; Z) \;\Delta_{v_1, y ; y \mid p[x \to v_{1X}, y] \wedge v_1 . v_2 ; v_3}\; Y)
\end{aligned}
\tag{C-24}
$$

Query C.13: (Type C)

$$
\begin{aligned}
&\alpha[x : \sigma[y : \exists z \in Z \bullet (r[y, z] \wedge q[x, z])](Y)](X) \equiv \\
&\alpha[x : \sigma[y : \exists z \in \sigma[v_1 : r[y, z \to v_1]](Z) \bullet q[x, z]](Y)](X) \equiv \\
&\alpha[x : \Gamma[v_2 : v_{2Y} \mid q[x, z \to v_{2Z}]](Y \text{ join}(y, v_1 ; [y, z \to v_1]) \; Z)](X) \equiv \\
&\alpha[v_3 : (v_3 . v_4)](X \;\Delta_{x, v_2 ; v_{2Y} \mid q[x, z \to v_{2Z}] ; v_4}\; (Y \text{ join}(y, v_1 ; [y, z \to v_1]) \; Z))
\end{aligned}
\tag{C-25}
$$

The splitting strategy chooses the conjunct *r* to include in the selection. This allows for introduction of a join in the following nesting step.

Query C.14: (Type D)

$$\alpha[x : \sigma[y : p[x, y] \wedge \exists z \in Z \bullet (r[y, z] \vee q[x, z])](Y)](X) \equiv \quad \text{(C-26)}$$
$$\alpha[x : \sigma[y : \exists z \in Z \bullet (r[y, z] \vee q[x, z])](\sigma[y : p[x, y]](Y))](X) \equiv$$
$$\alpha[v_1 : \sigma[y : \exists z \in Z \bullet (r[y, z] \vee q[x \to v_{1X}, z])](v_1 \,.\, v_2)](X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \equiv$$
$$\alpha[v_1 : \Gamma[v_3 : v_{3Y} \mid (r[y \to v_{3Y}, z \to v_{3Z}] \vee q[x \to v_{1X}, z \to v_{3Z}])](v_1 \,.\, v_2 \times Z)]$$
$$(X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \equiv$$
$$\alpha[v_1 : \Gamma[v_3 : v_{3Y} \mid (r[y \to v_{3Y}, z \to v_{3Z}] \vee q[x \to v_{1X}, z \to v_{3Z}])](v_1 \,.\, v_2 \times v_1 \,.\, v_4)]$$
$$((X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \;\mathbin{❖_{v_4}}\; Z)$$

The optimization splits first the conjunctive selection predicate to introduce the nestjoin (*X*,*Y*). Then, Cartesian product is introduced to avoid splitting of disjunctive predicate. Finally, collection *Z* is moved to the top level.

We examine alternative splitting strategy, in which we split expressions fully, and product transformation is not executed.

$$\alpha[x : \sigma[y : p[x, y] \wedge \exists z \in Z \bullet (r[y, z] \vee q[x, z])](Y)](X) \equiv \quad \text{(C-27)}$$
$$\alpha[x : \sigma[y : \exists z \in Z \bullet (r[y, z] \vee q[x, z])](\sigma[y : p[x, y]](Y))](X) \equiv$$
$$\alpha[x : \sigma[y : \exists z \in Z \bullet r[y, z] \vee \exists z \in Z \bullet q[x, z]](\sigma[y : p[x, y]](Y))](X) \equiv$$
$$\alpha[x : \sigma[y : \exists z \in Z \bullet r[y, z]](\sigma[y : p[x, y]](Y)) \cup \sigma[y : \exists z \in Z \bullet q[x, z]](\sigma[y : p[x, y]](Y))](X) \equiv$$
$$\alpha[v_1 : \sigma[y : \exists z \in Z \bullet r[y, z]](v_1 \,.\, v_2) \cup \sigma[y : \exists z \in Z \bullet q[x \to v_{1X}, z]](v_1 \,.\, v_2)]$$
$$(X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \equiv$$
$$\alpha[v_1 : ((v_1 \,.\, v_2) \text{ semijoin}(y, z\,;\, r[y, z])\; Z) \cup \sigma[y : \exists z \in Z \bullet q[x \to v_{1X}, z]](v_1 \,.\, v_2)]$$
$$(X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \equiv$$
$$\alpha[v_1 : ((v_1 \,.\, v_2) \text{ semijoin}(y, z\,;\, r[y, z])\; (v_1 \,.\, v_3)) \cup \sigma[y : \exists z \in v_1 \,.\, v_3 \bullet q[x \to v_{1X}, z]](v_1 \,.\, v_2)]$$
$$((X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \;\mathbin{❖_{v_3}}\; Z)$$

The full splitting strategy is clearly less profitable than in the first case. The result contains a nested quantifier in the collect function and a nested semijoin.

Query C.15: (Type D)

$$\alpha[x : \sigma[y : p[x, y] \wedge \exists z \in Z \bullet (r[y, z] \wedge q[x, z])](Y)](X) \equiv \quad \text{(C-28)}$$
$$\alpha[x : \sigma[y : \exists z \in Z \bullet (r[y, z] \wedge q[x, z])](\sigma[y : p[x, y]](Y))](X) \equiv$$
$$\alpha[v_1 : \sigma[y : \exists z \in Z \bullet (r[y, z] \wedge q[x \to v_{1X}, z])](v_1 \,.\, v_2)](X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \equiv$$
$$\alpha[v_1 : \sigma[y : \exists z \in \sigma[v_3 : r[y, z \to v_3]](Z) \bullet q[x \to v_{1X}, z]](v_1 \,.\, v_2)](X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \equiv$$
$$\alpha[v_1 : \Gamma[v_4 : v_{4y} \mid q[x \to v_{1X}, z \to v_{4Z}]](v_1 \,.\, v_2 \text{ join}(y, v_3\,;\, r[y, z \to v_3])\; Z)](X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \equiv$$
$$\alpha[v_1 : \Gamma[v_4 : v_{4y} \mid q[x \to v_{1X}, z \to v_{4Z}]](v_1 \,.\, v_2 \text{ join}(y, v_3\,;\, r[y, z \to v_3])\; v_1 \,.\, v_5)]$$
$$((X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \;\mathbin{❖_{v_5}}\; Z)$$

The chosen splitting strategy leads to introduction of a nestjoin (*X*,*Y*). Then, the attribute reference $v_1 \,.\, v_2$ can be joined with *Z* after the conjunctive quantifier predicate is split properly. The collection reference *Z* is finally removed.

The expression allows for multiple choices in splitting strategy. We present here the derivation with full splitting:

$$\alpha[x : \sigma[y : p[x, y] \wedge \exists z \in Z \bullet (r[y, z] \wedge q[x, z])](Y)](X) \equiv \qquad \text{(C-29)}$$
$$\alpha[x : \sigma[y : p[x, y]](\sigma[y : \exists z \in Z \bullet (r[y, z] \wedge q[x, z])](Y))](X) \equiv$$
$$\alpha[x : \sigma[y : p[x, y]](\sigma[y : \exists z \in \sigma[v_1 : r[y, z \rightarrow v_1]](Z) \bullet q[x, z]](Y))](X) \equiv$$
$$\alpha[x : \sigma[y : p[x, y]](\Gamma[v_2 : v_{2Y} \mid q[x, z \rightarrow v_{2Z}]](Y \text{ join}(y, v_1 ; r[y, z \rightarrow v_1])\ Z))](X) \equiv$$
$$\alpha[v_3 : \sigma[y : p[x \rightarrow v_{3X}, y]](v_3 \,.\, v_4)](X \ \Delta_{x,\, v_2 ;\, v_{2Y} \mid q[x,\, z \rightarrow v_{2Z}] ;\, v_4}(Y \text{ join}(y, v_1 ; r[y, z \rightarrow v_1])\ Z))$$

The alternative result is better than the first result. However, this can only be achieved if the splitting strategy includes multi-step planning for join introductions. In the above example, instead of the predicate *p*, we put the quantifier in the inner selection predicate. Then, we choose the predicate *r* to be placed in the inner selection predicate. This allows for a join of tables Y and Z, while in the first result the nestjoin is created first. In the first strategy, we choose for such a splitting which allows us to introduce a join in the **next** unnesting step. The choice for splitting in the second example is based on the possibility of join introduction further in the optimization process (not in the nest unnesting step). To realize such a strategy, the system should therefore **plan** the sequence of splitting steps leading to the introduction of a join. However, as we have seen in the previous examples, too much splitting, while allowing the join to be introduced, can cause negative effects in other parts of the expression. That's why such a planning system cannot work with the introduction of a local join as its only goal. In our implementation, we have chosen for a simple planning system which prefers splits leading to direct join introduction. This allows us to avoid multi-step planning, and provides satisfactory results throughout the whole spectrum of example expressions.

2.4 c = conjunction, Q = negated existential

Query C.16: (Type A)

$$\alpha[x : \sigma[y : p[x, y] \wedge \neg\exists z \in Z \bullet r[y, z]](Y)](X) \equiv \qquad \text{(C-30)}$$
$$\alpha[x : \sigma[y : p[x, y]](\sigma[y : \neg\exists z \in Z \bullet r[y, z]](Y))](X) \equiv$$
$$\alpha[x : \sigma[y : p[x, y]](Y \text{ antijoin}(y, z ; r[y, z])\ Z)](X) \equiv$$
$$\alpha[v_1 : (v_1 \,.\, v_2)](X \ \Delta_{x,\, y ;\, y \mid p[x,\, y] ;\, v_2}(Y \text{ antijoin}(y, z ; r[y, z])\ Z))$$

The optimization chooses the predicate to split based on the possibility of antijoin introduction. Subsequently, the antijoin result is nestjoined with collection X.

Query C.17: (Type B)

$$\begin{aligned}
&\alpha[x : \sigma[y : p[x, y] \wedge \neg\exists z \in Z \bullet q[x, z]](Y)](X) \equiv \\
&\alpha[x : \sigma[y : \neg\exists z \in Z \bullet q[x, z]](\sigma[y : p[x, y]](Y))](X) \equiv \\
&\alpha[v_1 : \sigma[y : \neg\exists z \in Z \bullet q[x \rightarrow v_{1X}, z]](v_1 \,.\, v_2)](X \;\Delta_{x, y\,;\, y \,|\, p[x, y]\,;\, v_2}\; Y) \equiv \\
&\alpha[v_1 : (\sigma[v_3 : \neg q[x \rightarrow v_{1X}, z \rightarrow v_{3Z}]](v_1 \,.\, v_2 \times Z) \div Z)](X \;\Delta_{x, y\,;\, y \,|\, p[x, y]\,;\, v_2}\; Y) \equiv \\
&\alpha[v_1 : (\sigma[v_3 : \neg q[x \rightarrow v_{1X}, z \rightarrow v_{3Z}]](v_1 \,.\, v_2 \times v_1 \,.\, v_4) \div v_1 \,.\, v_4)]((X \;\Delta_{x, y\,;\, y \,|\, p[x, y]\,;\, v_2}\; Y) \;❖_{v_6} Z)
\end{aligned} \tag{C-31}$$

The splitting choice allows to introduce a nestjoin in the next unnesting step. However, the negated quantifier in the map function can only be removed by introduction of a division. This is caused by the lack of rules for unnesting quantifiers within map functions (markjoin transformations). If markjoin transformations are applied, we get the following optimization:

$$\begin{aligned}
&\alpha[x : \sigma[y : p[x, y] \wedge \neg\exists z \in Z \bullet q[x, z]](Y)](X) \equiv \\
&\alpha[v_1 : \sigma[y : p[x \rightarrow v_{1X}, y] \wedge \neg(v_1 \,.\, v_2)](Y)](X \;\bot_{x, z\,;\, q[x, z]\,;\, v_2}\; Z) \equiv \\
&\alpha[v_3 : (v_3 \,.\, v_4)]((X \;\bot_{x, z\,;\, q[x, z]\,;\, v_2}\; Z) \;\Delta_{v_1, y\,;\, y \,|\, p[x \rightarrow v_{1X}, y] \wedge \neg(v_1 \,.\, v_2)\,;\, v_4}\; Y)
\end{aligned} \tag{C-32}$$

Markjoin rules allow for better result: there are no expensive operators in the map function.

Query C.18: (Type C)

$$\begin{aligned}
&\alpha[x : \sigma[y : \neg\exists z \in Z \bullet (r[y, z] \wedge q[x, z])](Y)](X) \equiv \\
&\alpha[x : \sigma[y : \neg\exists z \in \sigma[v_1 : q[x, z \rightarrow v_1]](Z) \bullet r[y, z]](Y)](X) \equiv \\
&\alpha[v_3 : \sigma[y : \neg\exists z \in v_3 \,.\, v_2 \bullet r[y, z]](Y)](X \;\Delta_{x, v_1\,;\, v_1 \,|\, q[x, z \rightarrow v_1]\,;\, v_2}\; Z) \equiv \\
&\alpha[v_3 : Y \text{ antijoin}(y, z\,;\, r[y, z])\; v_3 \,.\, v_2](X \;\Delta_{x, v_1\,;\, v_1 \,|\, q[x, z \rightarrow v_1]\,;\, v_2}\; Z) \equiv \\
&\alpha[v_3 : v_3 \,.\, v_4 \text{ antijoin}(y, z\,;\, r[y, z])\; v_3 \,.\, v_2]((X \;\Delta_{x, v_1\,;\, v_1 \,|\, q[x, z \rightarrow v_1]\,;\, v_2}\; Z) \;❖_{v_4} Y)
\end{aligned} \tag{C-33}$$

The result achieved cannot be improved by choosing the conjunct r as the inner select predicate. Even though the collections Y and Z are on the adjacent nesting levels, no join can be introduced because of the negation in the quantifier. Our splitting strategy prescribes in such situation to search for a nestjoin to be introduced in the next unnesting step. The nestjoin is then introduced, and the resulting set-valued attribute can be antijoined with collection Y.

Query C.19: (Type D)

$$\begin{aligned}
&\alpha[x : \sigma[y : p[x, y] \wedge \neg\exists z \in Z \bullet (r[y, z] \vee q[x, z])](Y)](X) \equiv \\
&\alpha[x : \sigma[y : \neg\exists z \in Z \bullet (r[y, z] \vee q[x, z])](\sigma[y : p[x, y]](Y))](X) \equiv \\
&\alpha[v_1 : \sigma[y : \neg\exists z \in Z \bullet (r[y, z] \vee q[x \rightarrow v_{1X}, z])](v_1 \,.\, v_2)](X \;\Delta_{x, y\,;\, y \,|\, p[x, y]\,;\, v_2}\; Y) \equiv \\
&\alpha[v_1 : (\sigma[v_3 : \neg(r[y \rightarrow v_{3Y}, z \rightarrow v_{3Z}] \vee q[x \rightarrow v_{1X}, z \rightarrow v_{3Z}])](v_1 \,.\, v_2 \times Z) \div Z)] \\
&(X \;\Delta_{x, y\,;\, y \,|\, p[x, y]\,;\, v_2}\; Y) \equiv \\
&\alpha[v_1 : (\sigma[v_3 : \neg(r[y \rightarrow v_{3Y}, z \rightarrow v_{3Z}] \vee q[x \rightarrow v_{1X}, z \rightarrow v_{3Z}])](v_1 \,.\, v_2 \times v_1 \,.\, v_4) \div v_1 \,.\, v_4)] \\
&((X \;\Delta_{x, y\,;\, y \,|\, p[x, y]\,;\, v_2}\; Y) \;❖_{v_4} Z)
\end{aligned} \tag{C-34}$$

The optimization first splits the conjunctive predicate to allow for nestjoin introduction in the following unnesting step. We then avoid the range split by introducing the division of a Cartesian product. To assess the quality of our strategy, we present the alternative optimization, in which the expressions are split fully:

$$
\begin{array}{l}
\alpha[x : \sigma[y : p[x, y] \wedge \neg\exists z \in Z \bullet (r[y, z] \vee q[x, z])](Y)](X) \equiv \\
\alpha[x : \sigma[y : \neg\exists z \in Z \bullet (r[y, z] \vee q[x, z])](\sigma[y : p[x, y]](Y))](X) \equiv \\
\alpha[x : \sigma[y : \neg\exists z \in Z \bullet q[x, z] \wedge \neg\exists z \in Z \bullet r[y, z]](\sigma[y : p[x, y]](Y))](X) \equiv \\
\alpha[x : \sigma[y : \neg\exists z \in Z \bullet q[x, z] \wedge \neg\exists z \in Z \bullet r[y, z]](\sigma[y : p[x, y]](Y))](X) \equiv \\
\alpha[x : \sigma[y : \neg\exists z \in Z \bullet q[x, z]](\sigma[y : \neg\exists z \in Z \bullet r[y, z]](\sigma[y : p[x, y]](Y)))](X) \equiv \\
\alpha[v_1 : \sigma[y : \neg\exists z \in Z \bullet q[x \rightarrow v_{1X}, z]](\sigma[y : \neg\exists z \in Z \bullet r[y, z]](v_1 \,.\, v_2))](X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \equiv \\
\alpha[v_1 : \sigma[y : \neg\exists z \in Z \bullet q[x \rightarrow v_{1X}, z]]((v_1 \,.\, v_2) \text{ antijoin}(y, z\,;\, r[y, z])\; Z)](X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \equiv \\
\alpha[v_1 : \sigma[y : \neg\exists z \in v_1 \,.\, v_3 \bullet q[x \rightarrow v_{1X}, z]]((v_1 \,.\, v_2) \text{ antijoin}(y, z\,;\, r[y, z])\; (v_1 \,.\, v_3))] \\
((X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \;\mathbin{❖_{v_3}}\; Z)
\end{array}
\tag{C-35}
$$

The expression in the map set is the same as in the first formulation. However, the map function in the second case contains a quantifier and a nested antijoin. In the first result, quantifier is removed.

Query C.20: (Type D)

$$
\begin{array}{l}
\alpha[x : \sigma[y : p[x, y] \wedge \neg\exists z \in Z \bullet (r[y, z] \wedge q[x, z])](Y)](X) \equiv \\
\alpha[x : \sigma[y : \neg\exists z \in Z \bullet (r[y, z] \wedge q[x, z])](\sigma[y : p[x, y]](Y))](X) \equiv \\
\alpha[v_1 : \sigma[y : \neg\exists z \in Z \bullet (r[y, z] \wedge q[x \rightarrow v_{1X}, z])](v_1 \,.\, v_2)](X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \equiv \\
\alpha[v_1 : \sigma[y : \neg\exists z \in \sigma[v_3 : q[x \rightarrow v_{1X}, z \rightarrow v_3]](Z) \bullet r[y, z]](v_1 \,.\, v_2)](X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \equiv \\
\alpha[v_1 : \sigma[y : \neg\exists z \in v_1 \,.\, v_5 \bullet r[y, z]](v_1 \,.\, v_2)] \\
((X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \;\Delta_{v_4, v_3\,;\, v_{3Z} \mid q[x \rightarrow v_{4X}, z \rightarrow v_3]\,;\, v_5}\; Z) \equiv \\
\alpha[v_1 : (v_1 \,.\, v_2 \text{ antijoin}(y, z\,;\, r[y, z])\; v_1 \,.\, v_5)]((X \;\Delta_{x, y\,;\, y \mid p[x, y]\,;\, v_2}\; Y) \;\Delta_{v_4, v_3\,;\, v_{3Z} \mid q[x \rightarrow v_{4X}, z \rightarrow v_3]\,;\, v_5}\; Z)
\end{array}
\tag{C-36}
$$

As before, we present an alternative optimization, in which expressions are split fully, and splitting is not guided by the next unnesting step:

$$
\begin{array}{l}
\alpha[x : \sigma[y : p[x, y] \wedge \neg\exists z \in Z \bullet (r[y, z] \wedge q[x, z])](Y)](X) \equiv \\
\alpha[x : \sigma[y : p[x, y]](\sigma[y : \neg\exists z \in Z \bullet (r[y, z] \wedge q[x, z])](Y))](X) \equiv \\
\alpha[x : \sigma[y : p[x, y]](\sigma[y : \neg\exists z \in \sigma[v_1 : q[x, z \rightarrow v_1]](Z) \bullet r[y, z]](Y))](X) \equiv \\
\alpha[v_2 : \sigma[y : p[x \rightarrow v_{2X}, y]](\sigma[y : \neg\exists z \in v_2 \,.\, v_3 \bullet r[y, z]](Y))](X \;\Delta_{x, v_1\,;\, v_1 \mid q[x, z \rightarrow v_1]\,;\, v_3}\; Z) \equiv \\
\alpha[v_2 : \sigma[y : p[x \rightarrow v_{2X}, y]](Y \text{ antijoin}(y, z\,;\, r[y, z])\; (v_2 \,.\, v_3))](X \;\Delta_{x, v_1\,;\, v_1 \mid q[x, z \rightarrow v_1]\,;\, v_3}\; Z) \equiv \\
\alpha[v_2 : \sigma[y : p[x \rightarrow v_{2X}, y]]((v_2 \,.\, v_4) \text{ antijoin}(y, z\,;\, r[y, z])\; (v_2 \,.\, v_3))] \\
((X \;\Delta_{x, v_1\,;\, v_1 \mid q[x, z \rightarrow v_1]\,;\, v_3}\; Z) \;\mathbin{❖_{v_4}}\; Y)
\end{array}
\tag{C-37}
$$

The alternative formulation is worse than the first formulation: it contains a nested Cartesian product.

Appendix D. Analytical model for distinct component server

In Chapter VI, we have presented an analytical model for client-side distribution of components. The model was restricted to the situation when the components were initially placed on the database server, which played also the role of the component server. We consider here a more general situation: one in which the components are placed on a distinct component server. The modifications to the model concentrate in the deployment phase.

We define the connection as an association of 1 client, 1 database server and the network between them. In the general model, we distinguish another agent: the component server. The network connections between the client, the database server and the component server are represented as capacity parameters: number of bytes transferred per unit of time. For the network between the client and the database server, the parameter is C_n, between the client and the component server it is C_c, and between the component and database server it is C_s.

Initially, all components are located on the component server. When a client intends to pose queries to the database server, the component server determines which components are to be placed on the client and the database server and transfers them to both agents.

The actual optimization proceeds identically to the model in Chapter VI.

Additional parameters of the model concern the processing power of the components server. We introduce factor f_s as the relative power of the component server in comparison to the database server.

During deployment, the components are marshalled ("wrapped") on the component server side, transferred over the network to the client and the database server, and unmarshalled ("unwrapped") on the clients and the database server. We characterize the marshalling and transfer delays for the new situation. We characterize these delays by parameters ξ_c and ξ_s (all

defined as a delay per 1 byte). Delay ξ_c represents marshalling of components on the component server, their transfer from the component server to the client, and their unmarshalling and installation on the client. Delay ξ_s represents marshalling of components on the component server, their transfer from the component server to the database server, and their unmarshalling and installation on the server. Similarly to processing time, we represent the marshalling activities by the parameters t_{wc} and t_{wq}, which denote the delays spent on the database server while marshalling (or unmarshalling) a unit of computation or a query object. These parameters allow us to formulate the delays ξ_c and ξ_s:

$$\begin{aligned} \xi_c &= t_{wc}\left(\frac{c^r}{f_s} + \frac{1}{f_c}\right) + \frac{1}{C_c} \\ \xi_s &= t_{wc}c^r\left(\frac{1}{f_s} + 1\right) + \frac{1}{C_s} \end{aligned} \qquad \text{(D-1)}$$

In the new situation, the deployment time T_d consists of transfer of components from the component server to both the client and the database server:

$$T_d = \xi_c \sum_{i=1}^{k} b_{ci} + \xi_s \sum_{i=k+1}^{n} b_{ci} \qquad \text{(D-2)}$$

The optimization time T_o is the same as in the first model.

$$T_o = \frac{1}{f_c} \sum_{i=1}^{k} t_{pi} + b_k\xi_n + c^r \sum_{i=k+1}^{n} t_{pi} \qquad \text{(D-3)}$$

The final formula for the total time T is therefore as follows:

$$T = \xi_c \sum_{i=1}^{k} b_{ci} + \xi_s \sum_{i=k+1}^{n} b_{ci} + q\left(\frac{1}{f_c} \sum_{i=1}^{k} t_{pi} + c^r \sum_{i=k+1}^{n} t_{pi} + b_k\xi_n\right) \qquad \text{(D-4)}$$

Appendix E. Performance model analysis

In Chapter VI, we have presented the comparison of the results of the measurements and the analytical model for a number of relevant model parameters: the client population and the size of the query batch. Other parameters, which could not be involved in the measurements, also can influence the elapsed optimization time. We consider two of those parameters: the difference between the client and server and the network transfer speed. We also consider the Internet environment, with larger client populations and slower network. For this environment, we analyze the influence of factors analyzed for the intranet: the query batch size, the client population and the marshalling speed.

1. MODEL: BATCH SIZE AS FACTOR IN THE INTERNET CONTEXT

In Chapter VI, we analyzed the intranet environment. We use our model to extrapolate the Internet case: with a higher number of clients and slower network (but with the same high marshalling costs). For a network of 100 clients ($c=100$) and network speed of 30,000 bytes per second, we calculate the following figure for connection with a weak client ($f_c=0.2$).

The lines represent different batch sizes of 1, 10 and 30 (from lowest line to highest).

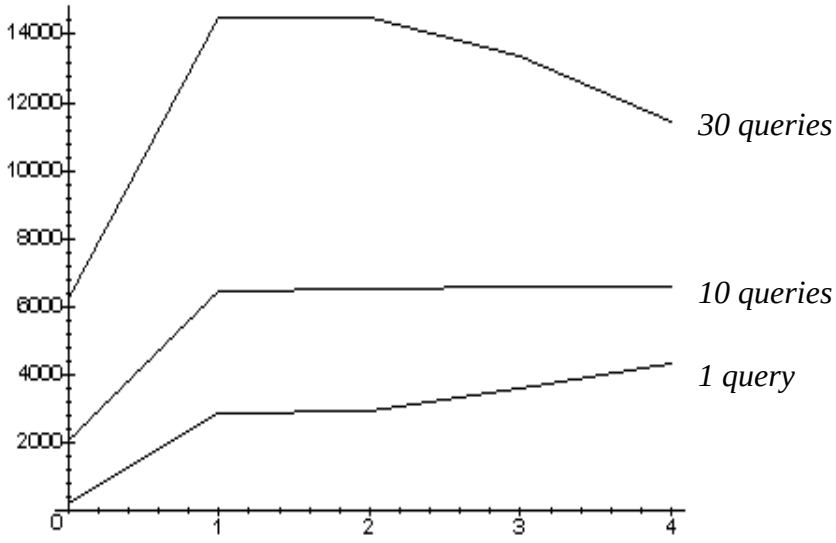

Figure E-1 Model: elapsed time as a function of query batch size: Internet case.

Similarly as in the intranet case, the most beneficial component allocation is the placement of all components on the server. The shape of the graph does not differ very much from the intranet case: this is caused by the simultaneous effects of two parameter changes. On one hand we have added more clients, which makes client-side distribution feasible. On the other hand, we have a much slower network, which makes component and query transfer less profitable.

2. MODEL: CLIENT POPULATION AS A FACTOR IN THE INTERNET CASE

Similarly as in the case of query batch size, we use the analytical model to predict the influence of client population in the Internet case. On the figure below, we vary the client population between 50 and 300, while the network transfer speed is set at 30,000 bytes per second.

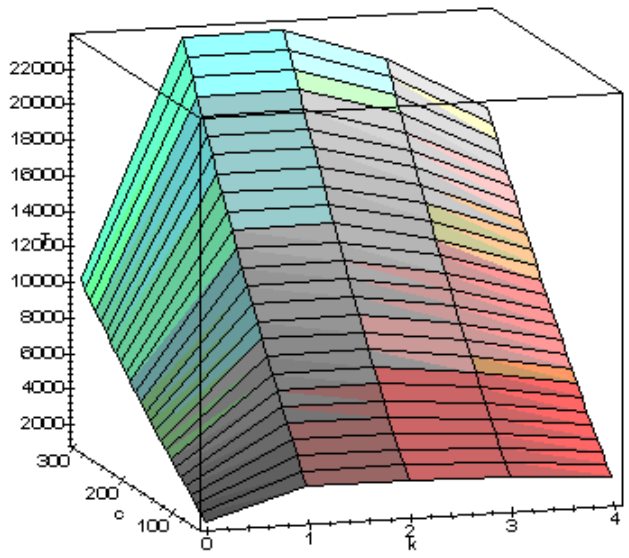

Figure E-2 Model extrapolation: elapsed time in relation to client population, Internet case.

The high marshalling costs for the large query tree objects make component transfer infeasible in all situations. We note the growing difference between elapsed time for k=1 and k=4: the increased connection costs take up the resources on the server. This leads to decreased resources on the server, and client-side transfer of all components becomes more feasible.

3. MODEL: ELAPSED TIME AS A FUNCTION OF MARSHALLING SPEED.

In Chapter VI, we have analyzed the influence of the marshalling speed in the intranet case (Figure 6-16). On the Internet, we reach the following situation (100 clients, network transfer speed of 30,000 bytes per second).

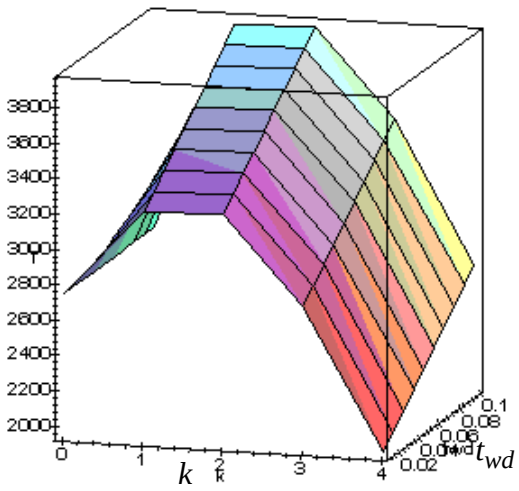

Figure E-3 Model: elapsed time as a function of marshalling speed.

Increase in the marshalling speed (lower t_{wd} values) make component transfer beneficial, as could be observed in the intranet environment.

We have analyzed the influence of three of the model factors (batch size, population size and the marshalling speed) for the Internet case. We now take other model parameters as factors: the marshalling speed (for the Internet case), the difference in processing power between the client and the database server, and the network transfer speed. For each of those parameters, we present their influence on the elapsed batch optimization time.

4. MODEL: ELAPSED BATCH OPTIMIZATION TIME AS A FUNCTION OF THE DIFFERENCE BETWEEN THE CLIENT AND THE SERVER

We analyze the influence of client processing power on the elapsed batch optimization time. For the intranet case (*c=8*, C_n*=1000000*), we obtain following figure. On the horizontal axes we have the component partition *k* and the client ratio f_c.

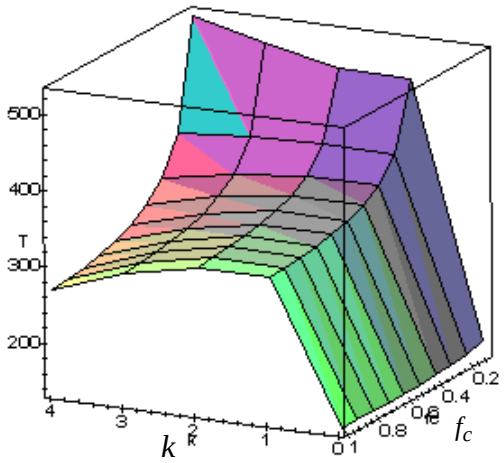

Figure E-4 Model: elapsed time as a function of the difference between client and server.

In all cases, the optimal elapsed time is achieved for *k=0*, thus when no components are transferred. No matter how powerful the clients are, it is always more beneficial to execute components on the server. This is caused by the relatively high marshalling costs, which, even for a fast network, outweigh the client-side processing benefits. As there are only 8 clients, server does not have to manage many connections and can easily cope both with the processing of components and the deserialization costs. We note that 8 clients is a relatively small level for intranet applications: larger client numbers bring about the benefits of client-side execution. The analytical model provides a practical conclusion: for this combination of input parameters, server-side execution will always be better, and investment in client power only will not make distribution feasible.

We now turn to the Internet case, with a population of 100 clients with network speed of 30,000 bytes per second. The client power is varied between f_c=*0.01* and *0.1*.

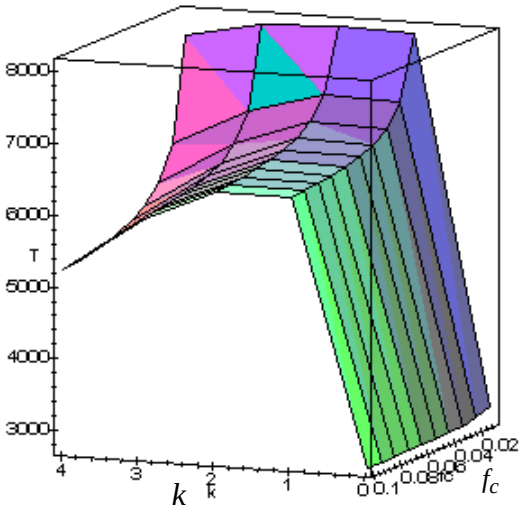

Figure E-5 Model: elapsed time as a function of the difference between client and server.

As was seen before, the high transfer costs of query tree prohibit the transfer of components. We can observer, however, the influence of stronger client in the increased feasibility of k=4 as a “second best” component partition.

We now turn to the network transfer speed as a factor.

5. MODEL: ELAPSED TIME AS A FUNCTION OF NETWORK TRANSFER SPEED

We first consider the intranet case, with 8 clients and fast network connection. The network speed is varied between 500,000 and 5,000,000 bytes per second).

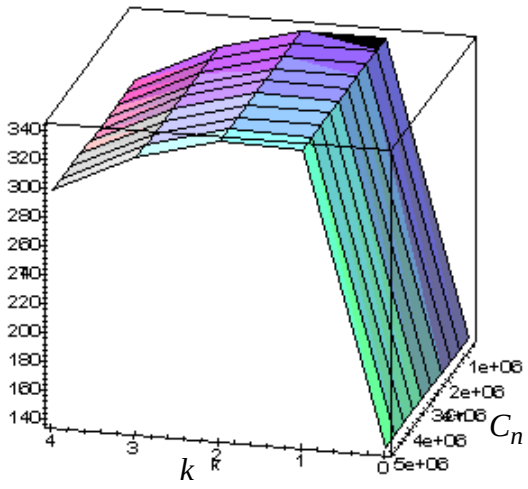

Figure E-6 Model: elapsed time as a function of network speed.

In all cases it is more beneficial to execute components on the server. In this situation, the network does not form the bottleneck: therefore any increase in network speed does not make client-side processing more feasible.

In the Internet case (network speed varying between 2,000 and 20,000 bytes/second), we achieve the following result.

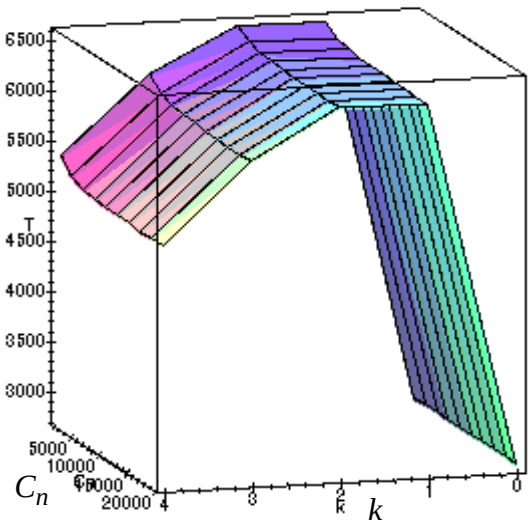

Figure E-7 Model: elapsed time as a function of network speed.

As was observed on the previous figures, prohibitively high marshalling costs make component transfer infeasible.

i. Literature list

[Abit86] S. Abiteboul, N. Bidoit, “Non First Normal Form Relations: An Algebra Allowing Data Restructuring”, *Journal of Computer and System Sciences*, 33, 1986

[Aper92] P.M.G. Apers, C.A. van den Berg, P.W.P.J. Grefen, M.L. Kersten & A.N. Wilschut, “PRISMA/DB: A Parallel Main-Memory Relational DBMS”, *IEEE Transactions on Knowledge and Data Engineering*, December 1992.

[Astra76] M.M. Astrahan et al., “System R: Relational Approach to Database Management”, *ACM Trans. on Database Systems*, 1(2), June 1976.

[Agor95] Agorics, Inc., “Going, Going, Gone”, available at `http://www.webcom.com/agorics/auctions/auction1.html`

[BaBZ93] H. Balsters, R.A. de By, R. Zicari, “Typed Sets as a Basis for object-Oriented Database Schemas”, *Proc. ECOOP*, Kaiserslautern, 1993.

[BaCD92] F. Bancilhon, C. Delobel, P. Kanellakis, “Building an Object-Oriented Database System - The Story of O_2”, Morgan Kaufmann Publishers, San Mateo, California, 1992.

[Bake96] Information on Query Optimizer Bakeoff project is available via the EREQ project page at `http://www.cse.ogi.edu/DISC/projects/ereq/ereq.html`

[Ben97] Java Optimization Resources page available at `http://www.cs.cmu.edu/~jch/java/resources.html`

[Ber96] A. Berson, “Client/Server Architecture”, 2nd ed., McGraw-Hill, 1996.

[BeKi89] E. Bertino, W. Kim, “Indexing Techniques for Queries on Nested Objects”, Technical Report ACT-OODS-132-89, MCC, 1989.

[Bog88] D.R. Boggs, J.C. Mogul and C.A. Kent, “Measured Capacity of an Ethernet: Myths and Reality”, *ACM Computer Communications Review* 18 (4), pp. 222-234, 1988.

[Bry89] F. Bry, “Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited”, *Proc. ACM SIGMOD Conference*, Portland, Oregon, June 1989, pp. 193-204.

[Cap97] M. Capoccia, “Dixy: A Distributed Data Dictionary System on the Internet”, M.Sc. Thesis, University of Twente, 1997.

[Car95] L. Cardelli, “A Language with Distributed Scope”, *DEC Technical Report*, available at `http://www.research.digital.com/SRC/Obliq/Obliq.html`

[Catt94] R.G.G. Cattell, “Object data management: object-oriented and extended relational database systems”, Addison-Wesley, 1994.

[Catt96] R.G.G. Cattell, ed. “The Object Database Standard: ODMG-93. Release 1.2”, Morgan Kaufmann Publishers, Inc., San Francisco, 1996.

[CDK94] G. Coulouris, J. Dollimore, T. Kindberg, “Distributed Systems Concepts and Design”, Addison-Wesley, 1994.

[Ches94] W. R. Cheswick, S. M. Bellovin, “Firewalls and Internet Security: repelling the wily hacker”, Addison-Wesley, 1994.

[Che96] Mitch Cherniack, Stanley B. Zdonik, “Rule Languages and Internal Algebras for Rule-Based Optimizers”, *Proc. SIGMOD Conference*, 1996.

[Che97] Mitch Cherniack, Stan Zdonik, Joon-Suk Lee, Kee-Eung Kim, “Composing Rules the COKO-KOLA Way”, Technical Report, Brown University, 1997.

[Cho82] T.C.K. Chou, J.A. Abraham, “Load balancing in distributed systems”, *IEEE Trans. on Software Engineering*, 8(4), 1982.

[Clu92] S. Cluet, C. Delobel, “A General Framework for the Optimization of Object-Oriented Queries”, *Proc. SIGMOD Conference*, 1992.

[Clu94] S. Cluet, G. Moerkotte, “Classification and Optimization of Nested Queries in Object Bases”, *Technical Report*, INRIA, 1994.

[Cod70] E. F. Codd, “A Relational Model for Large Shared Data Banks”, *Comm. of the ACM*, 13(6), June 1970.

[Colb90] L. S. Colby, “A Recursive Algebra for Nested Relations”, *Information Systems*, 15(5), 1990.

[Cor92] T. H. Cormen, C. E. Leierson, R. L. Rivest, “Introduction to Algorithms”, The MIT Press, 1992.

[Dada86] P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, G. Walch, “A DBMS Prototype to Support Extended NF2 Relations: An Integrated View of Flat Tables and Hierarchies”, *Proc. ACM SIGMOD Conf.*, Washington, DC, 1986.

[Dan91] S. Daniels et al., “Query Optimization in Revelation: An Overview”, *IEEE Data Engineering Bulletin*, 14(2), June 1991.

[ElNa94] R. Elmasri, S. B. Navathe, “Fundamentals of Database Systems”, 2nd ed., Benjamin/Cummings Publishing Company, Inc., 1994.

[FaMa95] L. Fageras, D. Maier, “Towards an Effective Calculus for Object Query Language”, *Proc. SIGMOD Conference*, 1995.

[Fer88] D. Ferguson, Y. Yemini, C. Nikolaou, “Microeconomic Algorithms for Load Balancing in Distributed Computer Systems”, *Proc. Distributed Computer Systems*, 1988.

[Fer95] D. F. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini, “Economic Models for Allocating Resources in Computer Systems”, in *Market based Control of Distributed Systems*, Ed. Scott Clearwater, World Scientific Press, 1995.

[Fiel97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, “Hypertext Transfer Protocol - HTTP 1.1”, RFC 2068, available at `http://www.w3.org/Protocols/rfc2068/rfc2068`

[FlKeS94] Jan Flokstra, Maurice van Keulen, Jacek Skowronek, “The IMPRESS DDT: A Database Design Toolbox based on a Formal Specification Language”. In R.T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD Conference*, Minneapolis, Minnesota, May 24-27, 1994.

[GaWo87] R.A. Ganski, A.K.T. Wong, “Optimization of Nested SQL Queries Revisited”, *Proc. ACM SIGMOD*, 1987.

[Gos94] J. Gosling, H. Mc Gilton, “The Java(tm) Language Environment: A White Paper”, available at `http://www.javasoft.com/whitePaper/java-whitepaper-1.html`.

[Gra86] G. Graefe, “Software Modularization with the EXODUS Optimizer Generator”, *Database Engineering,* Vol. 5, 1986.

[Gra87] G. Graefe, D. DeWitt, “The EXODUS Optimizer Generator”, *Proc. SIGMOD Conference*, 1987.

[Gra93a] G. Graefe, “Query Evaluation Techniques for Large Databases”, *ACM Computing Surveys*, Vol. 25, No. 2, 1993.

[Gra93b] G. Graefe, “The Volcano Optimizer Generator: Extensibility and Efficient Search”, *Proc. Data Engineering Conference*, 1993.

[Gries81] D. Gries, “The Science of Programming”, Springer-Verlag, 1981.

[Gries93] D. Gries, F. B. Schneider, “A Logical Approach to Discrete Math”, Springer Verlag, 1993.

[Gru97] T. Grust, J. Kröger, D. Gluche, A. Heuer, M. H. Scholl, “Query Evaluation in CROQUE - Calculus and Algebra Coincide”, Technical Report, University of Rostock, available at `http://wwwdb.informatikkkk,uni-rostock.de/Research/CROQUE.engl.html`

[Haa89] P. Haas, J.C. Freytag, G.M. Lohma, H. Pirahesh, “Extensible Query Processing in Starburst”, *Proc. SIGMOD Conference*, 1989.

[Hag86] R.B. Hagmann, D. Ferrari, “Performance Analysis of Back-End Database Architectures”, *ACM Trans. on Database Systems*, vol. 11, no. 1, March 1986.

[Hube88] B.A. Huberman (ed.), “The Ecology of Computation”, North-Holland, 1988.

[Infer97] Inferno documentation is available at `http://inferno.bell-labs.com/inferno/`

[Jai90] R. Jain, “The Art of Computer Systems Performance Analysis”, John Wiley, 1990.

[Jar84] M. Jarke, J. Koch, “Query Optimization in Database Systems”, *ACM Computing Surveys*, Vol. 16, No. 2, June 1984.

[JaSc82] G. Jaeschke, H. J. Schek, “Remarks on the Algebra of Non First Normal Form Relations”, In *Proc. of the Symposium on Principles of Database Systems*, pp. 124-138, ACM, March 1982.

[JaSe97] Java Web Server (formerly Jeeves) information avaialable at `http://jserv.javasoft.com/`

[KaDeW94] N. Kabra, D. DeWitt, “OPT++: An Object-Oriented Implementation for Database Query Optimization”, *Technical Report*, University of Wisconsion, 1994.

[KeMo90] A. Kemper, G. Moerkotte, “Access Support in Object Bases”, *Proc. SIGMOD*, 1991.

[KeMo94] A. Kemper, G. Moerkotte, “Object-oriented database management : applications in engineering and computer science”,Prentice Hall, 1994.

[KeSk96] M. van Keulen, J. Skowronek, P.M.G. Apers, H. Balsters, H.M. Blanken, R.A. de By, J. Flokstra, “A Framework for Representation, Validation and Implementation of Database Application Semantics”, In Database Application Semantics, Chapman & Hall, 1996.

[Kim82] W. Kim, “On Optimizing an SQL-like Nested Query”, *ACM Trans. on Database Systems*, Vol. 7, No. 3, 1982.

[KMPS94] A. Kemper, G. Moerkotte, K. Peithner, M. Steinbrunn, “Optimizing Disjunctive Queries with Expensive Predicates”, *Proc. ACM SIGMOD*, Minneapolis, 1994.

[KMPS95] A. Kemper, G. Moerkotte, K. Peithner, M. Steinbrunn, “Bypassing Joins in Disjunctive Queries”, *Proc. VLDB Conf.*, Zurich, 1995.

[Leu93] T. Leung, G. Mitchell, B. Subramanian, B. Vance, S. Vandenberg, and S. Zdonik, “The Aqua Data Model and Algebra”, TR CS-93-09, Brown University.

[Lan91a] R. Lanzelotte, P. Valduriez, “Extending the Search Strategy in a Query Optimizer”, In *Proc. of the 17th VLDB Conference*, 1991.

[Lan91b] R. Lanzelotte, P. Valduriez, M. Ziane, J.P. Cheiney, “Optimization of Non-Recursive Queries in OODBs”, *Proc. of Second International Conference on Deductive and Object-Oriented Databases*, December 1991.

[Lan92] Lanzelotte R., Valduriez P., Zait M., “Optimization of Object-Oriented Recursive Queries using Cost-Controlled Strategies”, Proc. SIGMOD’92.

[Lee88] M.K. Lee, J.C. Freytag, G.M. Lohman, ”Implementing an Interpreter for Functional Rules in a Query Optimizer”, Proc. 14th VLDB, Los Angeles, 1988.

[Loh88] G. Lohman, “Grammar-like Functional Rules for Representing Query Optimization Alternatives”, *Proc. SIGMOD Conference*, 1988.

[Mag96] MAGNUM project information is available at
`http://www.wins.uva.nl/~boncz/magnum/`

[Mel93] J. Melton, A. R. Simon, “Understanding the new SQL: a complete guide”, Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[Mill88] M.S. Miller, K.A. Drexler, “Markets and Computation: Agoric Open Systems”, in [Hube88].

[Mis92] P. Mishra, M.H. Eich, “Join Processing in Relational Databases”, ACM Computing Survey, vol. 24, no. 1, March 1992.

[Mit87] I. Mitrani, “Modelling of Computer and Communication Systems”, Cambridge University Press, 1987.

[Mit93] G. Mitchell, “Extensible Query Processing in an Object-Oriented Database”, Ph. D. Thesis.

[Neu94] B. Clifford Neumann, “Scale in Distributed Systems”, In *Readings in Distributed Computing Systems*, IEEE Computer Society Press, 1994.

[ÖzVa91] M. Tamer Özsu, P. Valduriez, “Principles of Distributed Database Systems”, Prentice Hall, 1991.

[Pir92] H. Pirahesh, J. Hellerstein, W. Hasan, “Extensible/Rule Based Query Rewrite Optimization in Starburst”, Proc. SIGMOD 92.

[RoKS88] M. A. Roth, H.F. Korth, A. Silberschatz, “Extended algebra and calculus for Nested Relational Databases”, *ACM Trans. On Database Systems*, 13(4), December 1988.

[RoSt87] L. Rowe, M. Stonebraker, “The POSTGRES Data Model”, *Proc. 13th VLDB Conference*, 1987.

[ScSc86] H.-J. Schek, M. H. Scholl, “The Relational Model with Relation-Valued Attributes”, *Information Systems*, 11(2), 1986, pp. 137-147.

[ScSi90] E. Sciore, J. Sieg Jr. “A Modular Query Optimizer Generator”, Proc. Data Engineering, Los Angeles, 1990.

[Sel79] P. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, T.G. Price, “Access Path Selection in a Relational Database Management System”, *Proc. SIGMOD Conference*, 1979.

[SmHu96] P. Smith, N.C. Hutchinson “Heterogeneous process migration: The Tui system”, *Technical Report 96-04*, University of British Columbia; February 1996.

[Ste94a] H. Steenhagen, P. Apers, H. Blanken, R.A. de By, “From Nested-Loop to Join Queries in OODB”, *Proc. 20th VLDB Conference*, 1994.

[Ste94b] H. Steenhagen, P. Apers, H. Blanken, “Optimization of Nested Queries in a Complex Object Model”, *Proc. EDBT Conference*, Cambridge, March 1994.

[Ste95] H. Steenhagen, “Optimization of Object Query Languages”, *Ph.D. Thesis*, University of Twente, 1995.

[Ston76] M. Stonebraker, M. Wong, E. Kreps and G. Held, “The Design and Implementation of INGRES”, *ACM Trans. on Database Systems*, 1(3), September 1976.

[Sto77] H.S. Stone, “Multiprocessor scheduling with the aid of network flow algorithms”, *IEEE Trans. on Software Engineering*, 3(1), 1977.

[Ston86] M. Stonebraker, L. Rowe, “The Design of POSTGRES”, Pr*oc. SIGMOD Conference*, ACM, 1986.

[Ston90] M. Stonebraker, L. Rowe, M. Hirohama, “The Implementation of POSTGRES”, *IEEE Trans. on Knowledge and Data Engineering*, 2(1), March 1990.

[Ston96] M. Stonebraker, P. M. Aoki, A. Pfeiffer, A. Sah, J. Sidell, C. Staelin, A. Yu, “MARIPOSA: A Wide-Area Distributed Database System”, *VLDB Journal 5*, *1* (Jan.1996), p. 48-63.

[Sun96] Barry D. Bowen and Carolyn W.C. Wong, “Spinning the Internal Web”, SunWorld Online, April 1996, available at
`http://www.sun.com/sunworldonline/swol-04-1996/swol-04-intranet.html`

[Tan90] A. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen, G. van Rossum, “Experiences with the Amoeba Distributed Operating System”, *Comms. ACM*, vol. 33, no. 12, 1990.

[Tan92] A. Tanenbaum, “Modern Operating Systems”, Englewood Cliffs NJ., Prentice Hall, 1992.

[ThFi86] S. J. Thomas, P. C. Fischer, “Nested Relational Structures”, *Advances in Computing Research III, The Theory of Databases*, P.C, Kanellakis, ed., JAIpress, 1986.

[Van93] S. Vandenberg, “Algebras for Object-Oriented Query Languages”, *Ph. D. Thesis*, U. Wisconsin.

[Wald92] C.A. Waldspurger, T. Hogg, B. Huberman, J. Kephart and S. Stornetta, “Spawn: A Distributed Computational Economy”, *IEEE Trans. on Software Engineering* 18, 2, Feb. 1992.

[WiAp91] A.N. Wilschut & P.M.G. Apers, “Dataflow query execution in a parallel main-memory DBMS”, *Proceedings 1st PDIS Conference*, Miami, Florida, USA, December 1991.

[WiFA95] A.N. Wilschut, J. Flokstra, P.M.G. Apers, “Parallel Evaluation of multi-join queries”, *Proceedings of the 1995 SIGMOD conference*, San Jose, California, USA, 1995.

[Wool95] M. Woldridge, N. Jennings, eds., “Intelligent Agents - Theories, Architectures, and Languages”, Lecture Notes in Artificial Intelligence, Springer-Verlag, Vol. 890, 1995.

[Zdo90] S. Zdonik, D. Maier, “Readings in Object-Oriented Database Systems”, Morgan Kaufmann, San Mateo, CA, 1990.

[Zed90] H.S.M. Zedan (ed.), “Distributed Computer Systems: Theory and Practice”, Butterworths, 1990.

ii. Index

A

activity diagrams VI-3, VI-5
alternative splitting strategy III-22
Amoeba V-7
analytical model VI-2
antijoin III-4, III-15, C-6, C-7, C-11, C-12
    introduction B-4
applicability checks V-5
auction IV-9
    auctioning mechanisms II-19

B

bid IV-15
    bidding price IV-11
bottleneck activity VI-7
budget IV-12
    budget set IV-9, IV-11, IV-12

C

Cartesian product III-15, III-16, C-8
centralized allocation IV-4
client population VI-8, VI-11, VI-24, VI-34, VI-35
client prefix IV-16
client-server SQL integration II-15
client-side data dictionary V-11, V-12
COKO II-14, V-17
collect expression III-3
collects within collects III-14
component
    broker IV-9, IV-10, IV-15, IV-17

partition IV-4, IV-9, VI-9
server VI-4
component partition VI-4, VI-9
conjunctive predicate split III-23, B-4
conjunctive quantifier matrix split B-4
consumers IV-10, IV-17
bidding IV-11
cost II-3
CROQUE II-13

D

Data Dictionary V-11, V-19
database server VI-4
deal IV-15
demand IV-14
deployment VI-5, VI-10
descoping of conjunctive matrix III-33
disjunctive predicate split B-4
disjunctive quantifier matrix split B-4
distributed allocation IV-4, IV-7
distributed computing system V-7
distributed TP managers II-15
division III-5
introduction B-4
Dutch auction II-19
Dutch-English auction system IV-9
dyadic expression III-9

E

eager consumers IV-16
elapsed batch optimization time VI-2, VI-5, VI-21, VI-24, VI-32, VI-33, VI-34, VI-35
empty client bids IV-18
English auction II-19
EPOQ II-12, V-16, V-17
existential quantifier composition III-32
existential quantifiers within selects III-13
EXODUS II-9, V-15, V-16
extensibility dimensions in optimizer construction II-7

F

factor VI-9

formal approaches to optimizer construction II-13
frontend and backend systems II-20
full split III-20, C-4, C-9
funding IV-10

H

heuristic determination of utility IV-13
heuristic ordering of transformations III-17

I

Inferno V-8, V-9, V-10
Internet IV-2
intranet IV-2

J

Java V-7, V-9, V-14, VI-37
join III-4, III-16
    introduction III-9, B-3
    transformations III-18

K

KOLA II-14, V-17

L

Lanzelotte & Valduriez II-11
logical optimization II-3
    algebra operators II-3

M

map III-3
MARIPOSA II-21, VI-36
markjoin III-39, III-40, C-5, C-8
    introduction III-39
markup IV-12
    vector IV-14
marshalling VI-5, VI-39

measurement VI-2
metric VI-2, VI-5
Miniscope Normal Form III-32, A-1, B-3
model-based utility determination IV-12, IV-14
modularisation of optimizers II-11
monoid calculus II-13

N

negated existential quantifiers within selects III-13
negated markjoin introduction III-39
nested Cartesian product III-5, III-16, III-40, C-6, C-12
    introduction B-4
nested product transformation III-18
nested relational data models II-4
nestjoin III-5, III-16, III-25, A-4, C-3, C-6, C-9, C-10, C-12
    introduction B-4
    transformation III-18
network transfer VI-5
normalization III-31, III-32

O

object data models II-6
Obliq V-7, V-9
optimization VI-5
    strategy II-3

P

parameters VI-2
parsing/unparsing V-10
partition IV-10
path expression II-6
performance model IV-4
physical optimization II-3
    operators II-2
Prenex Normal Form III-32, A-1
pricing IV-10
    price vector IV-11
process migration V-7
processing VI-5
product III-4
    introduction III-10, B-4

transformations III-18
projection III-3

Q

quantifiers within collect functions III-38
query II-2
language II-2
object V-14
optimization II-2
optimizer economy IV-8
processing II-2
query batch VI-12
query optimization I-2
Query Optimizer Bakeoff III-28

R

range split III-20, C-2, C-3
transformations III-20
relational
algebra II-4
calculus II-4
model II-4
reluctant consumers IV-15, IV-16
representation
query expression V-3
rule sets V-6
strategy V-5
request for bids IV-11
resource allocation II-17
0-1 integer programming techniques II-17
adaptive load balancing models II-18
exchange-based economies II-19
graph-theoretic approach II-17
hierarchic II-17
market-based II-18
price-base models II-19
static models II-17
rulesets III-18

S

selection III-3

semijoin III-4, III-15, C-8
    introduction III-9, B-3
serialization V-10
server postfix IV-17
servlet V-14
splitting III-19
    disjunctive predicates III-20
    strategy III-26
standardization III-31
Starburst V-16
strategy III-2
System R II-7

T

table expression III-2
test query set C-1
TMa III-4
TMql III-2
    expression forms III-2
transformation rules notation III-8
translation III-31
TUI V-7, V-9

U

unnesting
    loop III-26
    rules III-8
    strategy II-5, III-27
usage patterns
    off-line IV-2
    on-line IV-2
utility II-16, IV-11, IV-12

V

Volcano II-10, V-15, V-16

iii. Samenvatting

Een Database Management Systeem (DBMS) heeft tot taak om, zoals de naam al aangeeft, gegevens te beheren. Een DBMS biedt faciliteiten om gegevens in te voeren, te veranderen, te wijzigen en via een hoog-niveau querytaal (‘vraagtaal’) snel toegankelijk te maken. Het gegevensmodel van een DBMS legt de structuur van de gegevens vast. Een veel gebruikt gegevensmodel is het relationele model en relationele DBMS’en zijn geschikt om administratieve toepassingen te ondersteunen.
Het uitbreiden van de toepassingsgebieden, met name in de richting van geografische en technische toepassingen, is onderwerp van recent onderzoek. Het relationele gegevensmodel blijkt niet zo geschikt voor deze ‘nieuwe’ toepassingen. In dit proefschrift concentreren we ons op het geneste relationele gegevensmodel dat relaties als attribuutwaarden toestaat en is dus een uitbreiding van het relationele gegevensmodel. Deze uitbreiding maakt het model geschikter voor het ondersteunen van toepassingen uit niet-administratieve gebieden.
Een DBMS moet vragen uitgedrukt in een hoog-niveau querytaal, in ons geval de taal TMql, snel afhandelen. Een belangrijke rol wordt hierbij gespeeld door een speciale component, de zogenaamde query optimizer. Bij de vertaling en optimalisatie van vragen wordt gebruik gemaakt van een logische en een fysieke algebra. De omzetting van een vraag naar een efficiënte expressie in de logische algebra wordt vaak logische optimalisatie genoemd. Daarentegen is fysieke optimalisatie de omzetting van expressies in de logische algebra naar expressies in de fysieke algebra. Dit proefschrift houdt zich bezig met logische query optimalisatie en in het bijzonder met technieken om vragen voor het geneste relationele model snel af te handelen.
Speciale aandacht wordt besteed aan twee aspecten van query optimizers, namelijk complexiteit en schaalbaarheid. De complexiteit wordt veroorzaakt doordat de vragen geneste relaties betreffen en de schaalbaarheid betreft de mate waarin optimizers succesvol kunnen worden ingezet in grote netwerken, zoals het Internet. Dit proefschrift beschrijft modellen, technieken en algoritmen die kunnen helpen bij het bouwen van schaalbare logische query optimizers voor databases die gemodelleerd zijn volgens het geneste relationele model.
Het complexiteitaspect komt in Hoofdstuk III aan de orde. In de context van het geneste relationele model presenteren we een query optimizer die gebaseerd is op een verzameling van transformatieregels en een optimalisatiestrategie. De transformatieregels zijn gebaseerd op heuristieken die voordelige transformaties beschrijven: vanuit inefficiente calculus expressies met quantifiers tot efficiëntere expressies met algebraische operatoren. De verzameling van transformatieregels is gestructureerd in deelverzamelingen die het mogelijk maken om het transformatieproces in stappen te verdelen. De optimalisatiestrategie bepaalt de volgorde van de stappen en de manier van navigatie in de complexe query expressie. De regels en de strategie zijn geïmplementeerd in de Joquer query optimizer die een subset van een geneste query

taal TMql als input accepteert. De uiteindelijke strategie is ontstaan na experimenten waarin de Joquer optimizer werd uitgeprobeerd op verschillende TMql queries.
Het schaalbaarheidsprobleem wordt in Hoofdstuk IV beschreven. Het probleem kan optreden als de optimizer van een database-server vele vragen tegelijkertijd te beantwoorden krijgt; denk aan de omgeving waarin een database server via Internet benaderd wordt door vele clients. Als de TMql-optimizer uitsluitend op de database server aanwezig is, kan de afhandeling van vele vragen vertraging vooroorzaken. Onze oplossing voor het schaalbaarheidsprobleem is gebaseerd op het ontlasten van de server door een gedeelte van optimalisatie naar de clients te verplaatsen. De optimizer wordt in componenten opgedeeld en een deel daarvan wordt naar de client verplaatst. Op de client wordt de vraag gedeeltelijk geoptimaliseerd en vervolgens wordt het tussenresultaat naar de server verstuurd waar de rest van de optimalisatie plaatsvindt. De hypothese is dat het inschakelen van de client bij het optimaliseren voordelen kan bieden ten opzichte van optimalisatie door de server alleen.
We onderscheiden twee gevallen. In het eerste is het mogelijk een gemeenschappelijke doelfunctie voor de client en server te definiëren (zie Hoofdstuk VI), terwijl dat in het tweede niet mogelijk is. In het laatste geval kunnen autonome clients geïnteresseerd zijn in een snelle afhandeling van hun queries, terwijl de server belang heeft bij het maximaliseren van het aantal aangesloten clients: de doelen verschillen dus. In deze situatie is het onmogelijk om een enkel performance criterium te gebruiken, omdat elk van de autonome partijen een andere notie van betere performance heeft. Voor deze situatie stellen we een markt-gebaseerd allocatiemodel voor waarin de clients en de server een verdeling van de componenten vinden door onderhandelingen volgens het veilingprincipe uit te voeren.
Hoofdstuk V beschrijft de methoden en technieken die gebruikt worden in de implementatie van de Joquer query optimizer. Met name de diverse componenten van de optimizer komen aan de orde.
In Hoofdstuk VI komen we terug op het schaalbaarheidsprobleem en wel voor een client/server omgeving waarin een te optimaliseren doelfunctie voor het hele systeem gedefinieerd kan worden. Dat is in het algemeen mogelijk in bedrijven met gesloten netwerken (intranets). De reeds genoemde hypothese, namelijk dat het voordelig kan zijn clients in te schakelen bij optimalisatie, testen we op twee manieren: door het gebruik van een analytisch model en door het uitvoeren van experimenten. Als het analytisch model accuraat is, kan het gebruikt worden bij de verdeling van de optimizer-componenten over de clients en de server. Dan wordt bij gegeven invoerparameters, onder andere het aantal clients, bepaald hoe de componenten verdeeld moeten worden over de clients en de server. De experimenten zijn gebruikt om het analytische model te valideren. Het blijkt dat in sommige situaties het toewijzen van componenten aan clients inderdaad voordelig is.

iv. Acknowledgments

It is with great pleasure that I would like to express my thanks to a number of friends and coworkers which made my stay at the University of Twente a great experience.
First and foremost, I would like to thank my promotor, Peter Apers, for making this research possible. He allowed this research to proceed in paths which were not predicted at its initiation, but which I found interesting to pursue: this encompasses to me the essence of good management of research projects. In times of crisis, he provided the steadying hand which could put the project back on track.
I would like to thank Henk Blanken, my day-to-day supervisor, for the endless discussions and the many readings of my thesis parts. Together with Annita Wilschut, they formed the perfect team, allowing my ideas to stand the test of discussion. Their experience prevented me from "cutting corners" in my research, and make it more consistent with established practices. They accepted the new aspects in the research and allowed me to pursue them, probably at the expense of the areas envisioned initially. That courage and support I value immensely. I wish them all success in their further careers.
I would like to express thanks to the members of the committee: Sape Mullender, M. Tamer Ozsu, and Paul de Bra, to whom I am also grateful for great cooperation in the preparation and lecturing in the PAO post-academic courses.

I would like to thank my father, who was a research fellow on the Delft University in the years 1989-1990, for introducing me to the Netherlands during my first visit in 1989. This period led probably to my decision to look for study and research opportunities here when I received the TEMPUS grant in 1991. His friends in the Netherlands, Prof. Johan Arbocz of the TUD and his wife Margo, helped me during my first days in the Netherlands. My first contacts in Twente were Karel de Jonge, the dean of the faculty at that time, and Henk Blanken: they had the courage to invite the strange visitor from Eastern Europe for a period of research work. The Information Systems workgroup accepted me in their midst, and with me the occasional mutilations of their home language. Maurice van Keulen has shared the room with me for the last few years: he provided the invaluable local aspect to my Dutch education. He was a great coworker during the IMPRESS project; I am also grateful to other members of the project, among them Herman, Rolf, Jan, Reinier, Karina, for a great period of interesting, international work. I would like also to thank other members of the workgroup: Jan, Mark, Arjen, Rick, Sandra, Els, Sunil, Jochem and Erik, for the discussions, jokes, lunch walks and coffee breaks. Hein and

Rene, besides being coworkers, proved to be great vacation companions on our sailing trips. Jan Flokstra, and members of the technical faculty staff, helped to set up the experiments, which were performed during (cold) Christmas vacations of 1996. I am also grateful to the members of the faculty which allowed remote usage of their workstations during experiments.
I would like to thank the faculty staff for helping with my initial accomodation problems, and my friends of the student flat of Matenweg 4 for the fastest Dutch language and culture course on the planet.

Last but not least, I would like to thank my wife Gosia, for supporting me in this period, accepting the late hours, the absentmindedness and the constant hum of the PC in our living room.