Algebraic Query Optimization in Database Systems

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Diplom-Informatiker
Wolfgang Scheufele
aus Ravensburg

Mannheim, 1999

Dekan: Prof. Dr. Guido Moerkotte, Universität Mannheim
Referent: Prof. Dr. Guido Moerkotte, Universität Mannheim
Korreferent: Prof. Dr. Wolfgang Effelsberg, Universität Mannheim

Tag der mündlichen Prüfung: 22.12.1999

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Outline 2

2 Background and Overview **5**

2.1 Query Compilation and Execution 5

2.2 Problem Description and Terminology 8

2.3 Related Work 9

2.4 Overview of Complexity Results 10

3 Optimal Left-deep Execution Plans **13**

3.1 Chain Queries with Joins and Cross Products 13

3.1.1 Basic Definitions and Lemmata 13

3.1.2 The First Algorithm 33

3.1.3 The Second Algorithm 48

3.1.4 A Connection Between the Two Algorithms 52

3.2 Acyclic Queries with Joins and Selections 58

3.2.1 Preliminaries 58

3.2.2 Ordering Expensive Selections and Joins 62

4 Optimal Bushy Execution Plans **69**

4.1 The Complexity of Computing Optimal Bushy Processing Trees 69

4.2 General Queries with Joins, Cross Products and Selections 75

4.2.1 A Dynamic Programming Algorithm for the Basic Execution Space 75

4.2.2 An Efficient Implementation Using Bit Vectors 79

4.2.3 A First DP Algorithm for the Enlarged Execution Space 89

4.2.4 A Second DP Algorithm for the Enlarged Execution Space 96
4.2.5 Analysis of Complexity 102
4.2.6 Generalizations 105
4.2.7 Performance Measurements 107
4.2.8 Dynamic Programming: Variants and Applicability 109

5 Summary and Outlook **113**

5.1 Summary 113
5.2 Future Research 114

6 Zusammenfassung **115**

A An Implementation **117**

List of Figures

2.1 Query processing stages 6

2.2 Execution plan 7

2.3 A left-deep and a bushy tree 9

3.1 A decomposition tree. 28

3.2 A computation tree. 30

3.3 Subchains at various stages of Algorithm I 49

3.4 Plucking and grafting of subtrees 53

4.1 The first case in the NP-hardness proof 72

4.2 The second case in the NP-hardness proof 73

4.3 Structure of an optimal plan 75

4.4 Algorithm Optimal-Bushy-Tree(R,P) 77

4.5 Recurrences for computing fan-selectivity and size 81

4.6 Using ragged arrays to reduce storage space 84

4.7 Splitting conjunctive predicates 90

4.8 Edges incident to at least one node from a set of nodes 92

4.9 Edges incident an odd number of times to nodes from a set of nodes 92

4.10 Edges in a subgraph induced by a set of nodes 92

4.11 Removal of an edge yields multiple connected components 108

Acknowledgements

I like to thank my advisor, Prof. Guido Moerkotte for all the support and encouragement he provided. Furthermore, I like to thank Prof. Wolfgang Effelsberg for co-advising my work. I also like to thank Beate Rossi who read through a draft of this thesis.

Chapter 1

Introduction

1.1 Motivation

In modern database systems queries are expressed in a declarative query language such as SQL or OQL [MS92, CBB+97]. The users need only specify what data they want from the database, not how to get the data. It is the task of the *database management system (DBMS)* to determine an efficient strategy for evaluating a query. Such a strategy is called an *execution plan*. A substantial part of the DBMS constitutes the *query optimizer* which is responsible for determining an optimal execution plan. Query optimization is a difficult task since there usually exist a large number of possible execution plans with highly varying evaluation costs.

The core of query optimization is *algebraic query optimization*. Queries are first translated into expressions over some algebra. These algebraic expressions serve as starting point for *algebraic optimization*. Algebraic optimization uses algebraic rewrite rules (or algebraic equivalences) to improve a given expression with respect to all equivalent expressions (expressions that can be obtained by successive applications of rewrite rules). Algebraic optimization can be heuristic or cost-based. In heuristic optimization a rule improves the expression most of the time (but not always). Cost-based optimization, however, uses a cost function to guide the optimization process. Among all equivalent expressions an expression with minimum cost is computed. The cost function constitutes a critical part of a query optimizer. It estimates the amount of resources needed to evaluate a query. Typical resources are CPU time, the number of I/O operations, or the number of pages used for temporary storage (buffer/disk pages).

Without optimization, some queries might have excessively hight processing costs. Query optimization is extremely useful and often makes the computation of complex queries really possible. If queries are stated interactively, they probably contain only a few joins [LCW93]. However, if queries are generated by an application, considerably more joins and selections can be involved. Such queries are encountered in object-oriented database systems [KM94a] where the I/O cost of evaluating path expressions can be reduced by transforming path expressions into joins over object extents [KM94b, CD92], or in deductive database systems where complex rules involving many predicates in the body lead to many joins [KBZ86, KZ88]. Another source for complex queries are query generating database system front ends and complex nested views in decision-support applications.

Despite the long tradition of query optimization in database research the prevailing method to optimize queries is still dynamic programming as first described in the seminal paper [SAC+79]. In dynamic programming, all equivalent execution plans are enumerated by starting with trivial plans and successively building new plans from smaller plans while pruning comparable suboptimal

plans. Although this approach is very flexible, its time complexity is often inacceptable—especially for large problems. It is not uncommon that the time to optimize a query by far exceeds the time to process the query. Such high optimization times can only be tolerated if the query will be executed many times such that the optimization cost pays for itself. Nevertheless, dynamic programming is often the fastest algorithm known for query optimization problems. The question arises for which query optimization problems can we find efficient dedicated polynomial time query optimization algorithms and which problems can be proven NP-hard—thus leaving little hope for efficient polynomial algorithms. At present few is known about the complexity of optimizing algebraic expressions, and only one problem class is known for which a dedicated optimization algorithm has been developed. It is the aim of this thesis to resolve the complexity status of problem classes that have not yet been sufficiently investigated and to develop new optimization algorithms for these problems that outperform the previously known algorithms.

1.2 Outline

In this thesis, we investigate the complexity of various subclasses of the problem of computing optimal processing trees for conjunctive queries (queries involving only joins, cross products and selections). The subproblems arise from restricting the following problem features in various ways:

- the operations that occur in the query
- the shape of the query graphs
- the shape of the processing trees
- the operators allowed in processing trees

More precisely, we distinguish between queries that contain joins and cross products only and queries that additionally contain selections. Query graphs are classified as either chain-shaped, star-shaped, acyclic or general. We consider the two standard types of processing trees, namely left-deep trees and bushy trees. Processing trees may contain additional cross products or not. Each of these features influences the complexity of the problem. We treat the problem classes from two sides. One goal is to determine the complexity status (i.e. in P or NP-hard) of these problems, and our other goal is to devise more efficient algorithms for these problems that outperform the previously known algorithms.

Chapter 2 gives a brief overview of query processing in relational database systems. The problem of optimizing conjunctive queries is introduced and the related work is discussed. Furthermore, a summary of complexity results in this area is provided.

Chapter 3 deals with the optimization of *left-deep execution plans*. Left-deep execution plans are mainly used for reasons of simplicity and performance. First, the space of left-deep trees is much smaller than the space of bushy trees, thus reducing optimization time. Second, left-deep trees simplify both query optimizer and run time system. For example, only a single join operator is needed that joins a base relation to an intermediate result, and there is no need to materialize intermediate results (unlike bushy trees).[1]
One of the simplest join ordering problems with unknown complexity status is the problem to compute optimal left-deep trees with cross products for chain queries. We derive two efficient algorithms for this problem. The first algorithm produces the optimal plan but we could not prove that it has polynomial run time. The second algorithm runs in polynomial

[1] provided there is enough buffer space available

time but we could not prove that it produces the optimal result. A conjecture is stated, which implies that both algorithms run in polynomial time and produce the optimal plan.
Another simple but important type of queries are acyclic queries with expensive selections. Several researchers have proposed algorithms for computing optimal left-deep trees with joins and selections—all these algorithms having exponential run time. By modeling selections as joins we show that the algorithm of Ibaraki and Kameda [IK84] can be applied to the problem. The resulting algorithm runs in polynomial time but has the following limitations. First, expensive selections can only be placed on the path from the leftmost leaf node to the root of the tree, and second, the cost function has to fulfill the ASI property from [IK84].

Chapter 4 addresses the optimization of *bushy execution plans*. The space of bushy plans is larger than the space of left-deep plans but may contain considerably cheaper plans [OL90]. The question that immediately arises is whether we can expect polynomial algorithms for this more general problem. We prove that the problem is NP-hard, independent of the query graph. Thus, unless P=NP, there is no way to construct optimal bushy processing trees in polynomial time.
Consequently, the rest of the chapter is dedicated to the general problem of computing optimal bushy processing trees for general queries with expensive join and selection predicates. Although several researchers have proposed algorithms for this problem, apart from [CS97] all approaches later turned out to be wrong. We present three formally derived, correct dynamic programming algorithms for this problem. Our algorithms can handle different join algorithms, split conjunctive predicates, and exploit structural information from the join graph to speed up the computation. The time and space complexities of the algorithms are analyzed carefully and efficient implementations based on bitvector arithmetic are presented.

Chapter 5 summarizes the achievements and outlines areas of future research.

Chapter 2

Background and Overview

2.1 Query Compilation and Execution

This section provides a brief overview of the basic architecture of a database management system (DBMS) and the way queries are processed by the DBMS. We restrict ourselves to sequential[1] DBMS. For an overview of query processing, see [Gra93, ME92].

Query processing describes the process a query (being submitted by either a user or an application) is compiled and finally executed by the database management system. It typically comprises the following stages (see Figure 2.1).[2] First the query is submitted to the *scanner* which transforms the query string into a stream of tokens. The *parser* reads the token stream and validates it against the grammar of the query language SQL. The result of this step is an abstract syntax tree representing the syntactical structure of the query. The second stage comprises *factorization, semantic analysis, access & integrity control*, and *translation into internal representation*. All these tasks can be performed together in a single pass over the abstract syntax tree. Factorization introduces unique numbers (information units, IUs) to each intermediate result (attribute) such that all operators take only IUs as arguments and produce a unique IU. That is, if two operators are of the same type and have identical arguments they must have the same IU. Factorization ensures that common subexpressions are represented only once. In the semantic analysis the existence and validity of each object reference (table, attribute, view etc) is checked against the database schema. Besides, the validation of object references, access rights and integrity constraints are tested. In the translation step the abstract syntax tree is translated into a more useful internal representation. There are many internal representations. Most of them are calculus expressions, operator graphs over some algebra or tableaux representations. A very powerful intermediate representation is the Query Graph Model [PHH92]. Another task which is often performed in this stage is the transformation of boolean expressions into conjunctive normal form. The resulting clauses are called "boolean factors". Both conjunctive and disjunctive normal form have the property that NOT operators are pushed inside boolean expressions which is important for a correct handling of NULL values.[3] Furthermore, often transitive join predicates are added and constants are propagated across equality predicates (constant propagation).

The next stage is *view resolution* (view expansion, view merging) where each occurrence of a view table is replaced by the respective view definition. View resolution is analogous to macro expansion in programming languages. Note that view resolution produces nested queries which

[1] that is, there is one active processor and the system is not distributed over multiple machines
[2] In other systems some steps have other names, they may miss, be permuted or merged together.
[3] otherwise three-valued logic is necessary

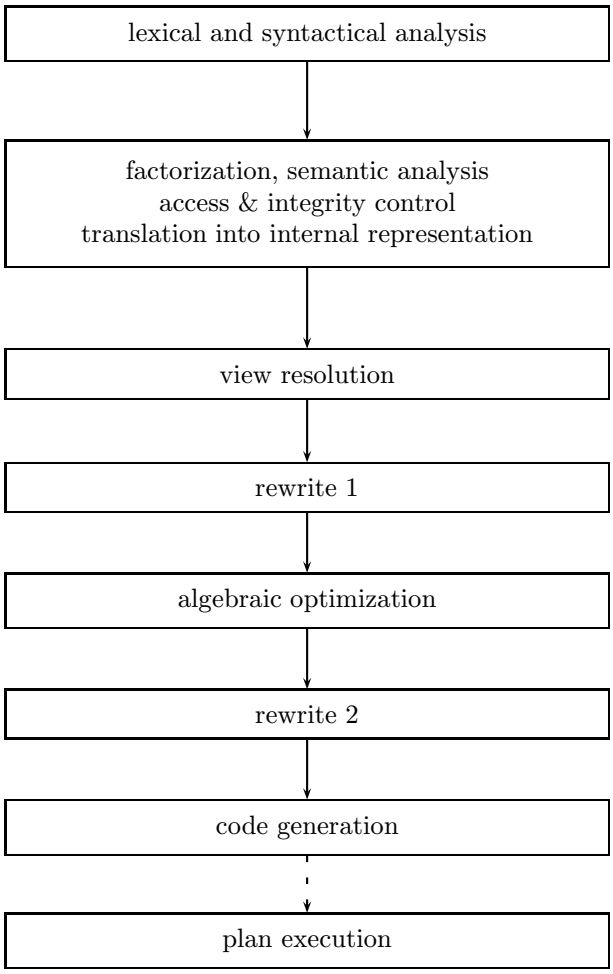


Figure 2.1: Query processing stages

requires unnesting. Although unnesting enlarges the search space by increasing the number of joins which leads to higher optimization times the larger search space may contain considerably cheaper plans. An alternative to view expansion is *view materialization*. However, view materialization calls for a mechanism that invalidates a materialized view and triggers its re-computation if an underlying base relation changes. The re-computation can either be from scratch or incrementally. In the latter case only the changes are propagated through the view. Materialized views should be fully incorporated into the query optimization process, i.e. the query optimizer should be able to replace parts of a query by materialized views, if this lowers costs. The maintainance and optimization of materialized views is still a topic of research.

It is the task of the *query optimizer* to consider different alternatives of executing the query and to pick the cheapest[4] execution plan. The query optimizer can be very complex and may require considerable computational resources. The result of the query optimization step is an *execution plan* (evaluation plan). Execution plans are annotated trees whose nodes are operators and whose edges indicate the data flow among the operators. Operators take as input one or more data streams and produce an output data stream. We distinguish between *logical* and *physical operators*. Logical operators are operators in the algebra of the internal representation. Examples of logical operators are selection, join, cross-product, grouping, etc. Most physical operators are implementations of logical operators, e.g. sequential-scan, index-scan, nested-loop-join, merge-join, hash-join, hash-grouping etc. Examples of physical operators that have no logical counter part are sort, index-scan, etc. Often algebraic optimization is performed in two phases. First, an operator tree involving *logical operators* (logical plan) is computed which is then transformed into an operator tree involving physical operators (physical plan). Figure 2.2 shows an example of an execution plan.

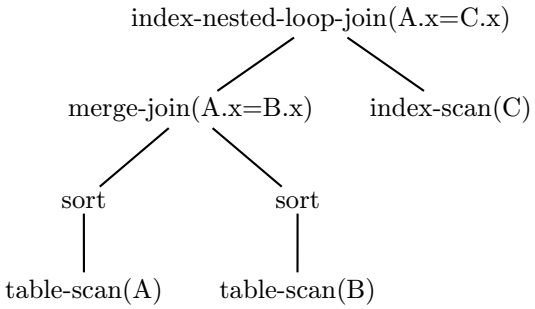


Figure 2.2: Execution plan

The *rewrite 1* stage mainly performs unnesting. That is, if possible, nested queries are rewritten into "flat" queries which allow more efficient evaluation and better algebraic optimization. If unnesting is not possible (or not implemented), other techniques can be applied. For example, semi-join based techniques like "magic rewriting" can be used to create a specialized view that can be evaluated separately, but computes fewer irrelevant tuples. Magic rewriting can also be applied to correlated queries ("magic decorrelation"). The "predicate move around" technique moves (or duplicates) predicates between queries and subqueries in order to yield as many restrictions in a block as possible. The next phase—*algebraic optimization*—is the core of the query optimizer. Algebraic optimization has received a lot of attention. In the algebraic optimization phase for each block of the query, an operator tree (plan) is generated whose nodes are physical algebraic operators. The next phase—*rewrite 2*—is again a rewrite phase. Here, small cosmetic rewrites are applied in order to prepare the plan for the subsequent code generation phase. The *query execution engine* implements a set of physical operators. In order to be executed, the execution plan has to be either translated into machine code to be executed directly or into intermediate code to be interpreted by the query execution engine. This translation takes place in the *code generation* step. Finally, the query is executed by the query execution engine (runtime system). Query code can be either static or dynamic. Static code basically executes the same sequence of statements at every execution whereas in dynamic code the sequence of instructions depends on run time parameters like the true cardinalities of certain relations or intermediate results, the buffer space available, etc. *Dynamic optimization* tries to react to inaccurate parameter estimations by evaluating these parameters along the execution of the query and comparing them to the estimated parameters. If estimated and computed parameters differ considerably, appropriate action can be taken, e.g. switching to a different execution plan.

[4]the cheapest plan with respect to the class of considered execution plans (search space)

This sequel is mainly concerned with the algebraic cost-based query optimization. For a detailed overview of general query optimization techniques see [JK84, GLSW93, Cha98]. An extensive survey of query evaluation techniques is [Gra93]. A good overview of heuristic and stochastic query optimization algorithms is [SPMK93, SMK97].

2.2 Problem Description and Terminology

A central problem in query optimization is to determine an efficient strategy to evaluate queries consisting of multiple joins and selections. Due to the commutativity and associativity of the join operator and the interchangeability of joins and selections such queries can be evaluated in a huge number of ways—with heavily varying costs.

We shall henceforth consider queries involving selections, joins and cross products *selection-join-queries* (another name is *conjunctive queries* [Ull89]). Select-join-queries are fully described by a set of base relations $\mathcal{R}$ and a set of query predicates $\mathcal{P}$. Each predicate $P \in \mathcal{P}$ is either a selection predicate referring to a single relation or a join predicate referring to two or more relations. We distinguish between two types of predicates. *Basic predicates* are simple built-in predicates or predicates defined via user-defined functions which may be expensive to compute. *Composite predicates* are boolean expressions formed out of basic predicates. For example, the predicate $R.x = S.x \wedge S.y \geq 0$ is a composite predicate consisting of the basic predicates $R.x = S.x$ and $S.y \geq 0$. The predicates of a query induce a *join graph* (or *query graph*) $G = (\mathcal{R}, E)$. E is the set of edges $e \subseteq R$ such that there exists a predicate $P \in \mathcal{P}$ relating the relations in e. We shall be mainly concerned with *binary* join predicates, i.e. $|e| = 2$ for any edge $e \in E$. Join graphs can be classified according to their topology. Common topologies are chains, stars, acyclic graphs, and general graphs.

To describe an instance of a join ordering problem we need to specify the following statistical parameters which are used to estimate the evaluation costs of plans. A prerequisite for estimating the costs of a plan are estimations of the sizes of all occurring intermediate results. Reasonable estimations for these sizes can be obtained by means of base relation cardinalities and predicate selectivities. The selectivity of a predicate is the expected fraction of tuples that qualifies. The cardinality of a relation R is denoted as $|R|$. If P is a selection predicate that refers to a relation R its selectivity is defined as

$$f_P = \frac{|\sigma_P(R)|}{|R|}.$$

Similarly, if P is a join predicate referring to the relations R and S, its selectivity is given by

$$f_P = \frac{|R \bowtie_P S|}{|R \times S|}.$$

If user-defined functions are involved the evaluation costs of a predicate can vary considerably. To account for this, we introduce a cost factor c_P associated with each predicate P. c_P measures the average cost of evaluating the predicate for one input tuple.

A *processing tree* (execution plan) is a labeled rooted tree whose leaves represent base relations and whose internal nodes correspond to selection, join or cross product operators. We often speak of trees and plans instead of processing trees and execution plans. Processing trees are classified according to their shape. The main distinction is between left-deep trees and bushy trees. In a *left-deep tree* the right subtree of an internal node is always a leaf. *Bushy trees* have no restriction on their shape. Figure 2.3 gives an example of a left-deep and a bushy tree. We say that a plan *avoids cross products* if there does not exist a second plan with fewer cross products that computes the same query. If the join graph is connected, this means that there should not be any cross products.

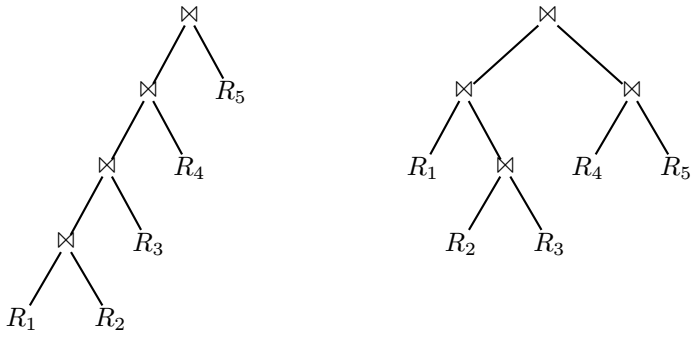


Figure 2.3: A left-deep and a bushy tree

In order to estimate the cost to evaluate a processing tree we must supply a *cost function*. Cost functions estimate the amount of resources needed to evaluate a query. Typical resources are CPU time, the number of I/O operations, or the number of pages used for temporary storage (buffer or disk pages). Usually a weighted average over several resources is used as costs. It is not surprising that cost functions play a critical role in a query optimizer. If the cost function is not accurate enough, even the best optimizer may come up with bad execution plans. Devising accurate cost functions is a challenging problem which is beyond the scope of this thesis.

The cost functions we use are rather simple. They sum up the costs of all the operators in the execution plan. The cost of an operator in the execution plan is defined in terms of the sizes of its input streams. More on the cost function can be found in chapter 3.1.1, 3.2.1, and 4.1.

2.3 Related Work

Several researchers addressed the problem of ordering binary joins in an n-way join. The standard and—even today—most prevailing method to solve this optimization problem is dynamic programming [SAC+79]. A fast implementation of a dynamic programming algorithm for bushy trees and cross products is described in [VM96]. In [OL90], Ono and Lohman discussed the complexity of dynamic programming algorithms for the join ordering problem. They also gave the first real world examples to show that abandoning cross products can lead to more expensive plans.
Besides dynamic programming, another approach, which is widely used in commercial optimizers, is transformation-based query optimization. These algorithms exhaustively enumerate the search space by successively transforming an initial execution plan. Unfortunately this approach has the drawback of considering partial plans many times. In [Pel97, PGLK96, PGLK97b, PGLK97a] an approach is described which avoids the duplicate generation of partial plans. Algorithms for left-deep plans or bushy plans and for connected acyclic or connected complete query graphs are presented. All these algorithms are ad-hoc designs and no generalization to arbitrary rule sets is known.
An NP-hardness result for the join ordering problem for general query graphs was established in 1984 [IK84]. Later on, a further result showed that even the problem of determining optimal left-deep trees with cross products for star queries is NP-hard [CM95]. The first polynomial time optimization algorithm was devised by Ibaraki and Kameda [IK84] in 1984. Their IK-algorithm solved the join ordering problem for the case of left-deep processing trees without cross products, acyclic join graphs and a nontrivial cost function counting disk accesses for a special block-wise nested-loop algorithm. The IK-algorithm was subsequently improved by Krishnamurthy, Boral and Zaniolo [KBZ86] to work in time $O(n^2)$. They assumed that the database is memory resident and used a simpler and more common cost function. The IK- and KBZ-algorithms both apply an algorithm for job sequencing [MS79]. In [CM95], the authors present an algorithm to find optimal

left-deep processing trees with cross products for star queries in time $O(2^c + n \log n)$ where c is the number of cross products in a solution. Concerning the area of non-exact algorithms, there are several approaches, too. The most simple ones are greedy type algorithms. These algorithms assume that an optimal solution (or good solution if the algorithm is viewed as of heuristic type) can always be extended to a larger likewise optimal (good) solution. An example is the heuristic of always joining the next two unused partial plans such that the intermediate result size of the resulting plan (or the join cost, the join selectivity or the costs of the resulting plan) is always minimal [SPMK93, STY93, GLSW93, SPMK93, Feg97]. The first published nontrivial heuristic for treating cyclic queries appeared in [KBZ86]. They applied the KBZ-algorithm to a minimal spanning tree of the join graph. Other approaches are based on general deterministic search methods probabilistic optimization methods [SL95, SL96, SS96]. A comparative overview of the best-known approaches to the join ordering problem can be can be found in [KRHM95, SPMK93].

The above mentioned approaches order joins only. Whereas this optimization problem attracted much attention in the database research community, much less investigations took place for optimizing boolean expressions not containing any join predicate [KMPS94, KMS92].

Only few approaches exist to the problem of ordering joins and selections with expensive predicates. In the LDL system [CGK89] and later on in the Papyrus project [CS93] expensive selections are modeled as artificial relations which are then ordered by a traditional join ordering algorithm producing left-deep trees. This approach suffers from two disadvantages. First, the time complexity of the algorithm cannot compete with the complexity of approaches which do not model selections and joins alike and, second, left-deep trees do not admit plans where more than one cheap selection is "pushed down". Another approach is based upon the "predicate migration algorithm" [HS93, Hel94, Hel98] which solves the simpler problem of interleaving expensive selections in an existing join tree. The authors of [HS93, Hel94, Hel98] suggest to solve the general problem by enumerating all join orders while placing the expensive selections with the predicate migration algorithm—in combination with a system R style dynamic programming algorithm endowed with pruning. The predicate migration approach has several severe shortcomings. It may degenerate to exhaustive enumeration, it assumes a linear cost model and it does not always yield optimal results [CS96]. Recently, Chaudhuri and Shim presented a dynamic programming algorithm for ordering joins and expensive selections [CS96]. Although they claim that their algorithm computes optimal plans for all cost functions, all query graphs, and even when the algorithm is generalized to bushy processing trees and expensive join predicates, the alleged correctness has not been proven at all. In fact, it is not difficult to find counterexamples disproving the correctness for even the simplest cost functions and processing trees. This bug was later discovered and the algorithm restricted to work on regular cost functions only [CS97]. Further, it does not generate plans that contain cross products. The algorithm is not able to consider different join implementations. Especially the sort-merge join is out of the scope of the algorithm due to its restriction to regular cost functions. A further disadvantage is that the algorithm does not perform predicate splitting.

2.4 Overview of Complexity Results

This section gives a brief overview of the complexity results known so far for join ordering problems. Consider the following classification of join ordering problems. Each class is abbreviated by a four letter string XYZW with X $\in$ {J,S}, Y $\in$ {N,C}, Z $\in$ {L,B} and W $\in$ {E,C,S,A,G}. The letters have the following meaning.

1. X – query type:

   J joins only

S joins and selections

2. Y – cross products:

   N not allowed

   C allowed

3. Z – processing trees:

   L left-deep trees

   B bushy trees

4. W – join graph:

   E empty (no edges)

   C chain

   S star

   A acyclic

   G general

For example, `JCLA` stands for the problem of computing an optimal left-deep processing tree with cross products for queries with acyclic join graphs.
Table 2.1 summarize the complexity results[5] for join ordering problems using the above classification scheme.

Although the complexity of the class `SNLA` is still unknown, Chaudhuri and Shim have shown that a special case of the problem can be solved in time polynomial in the number of selections and exponential in the number of joins [CS97]. Their dynamic programming algorithm is restricted to "regular cost functions" which are a subclass of ASI cost functions. All previous algorithms are exponential both in the number of joins and the number of selections. Nevertheless, the complexity bound derived in [CS97] are not as good as it first seems. Applying our algorithm in [SM96] to all subsets of pushed selections has a lower exponent in the asymptotic complexity than the algorithm of Chaudhuri and Shim, provided not all relations have a selection predicate.

Devising an NP-hardness proof for one of the problem classes `SNLC`, `SNLS`, and `SNLA` turns out to very hard since there seems to be no "pivot operation" whose cost dominates the cost of all other operators in the plan. If such a pivot operation existed—as in the proofs [CM95, SM97]—bounding the cost of all possible plans would be easy. But without such an operation, an NP-hardness proof seems out of sight.

[5] A comment on the relation between problem classes and their complexities. Although one would think that if a class of join ordering problems is NP-hard for the space of left-deep processing trees it should be NP-hard for bushy trees too, and if it is NP-hard for plans without cross products it should also be NP-hard for plans with cross products, this need not be the case. In general, if a problem class is NP-hard we cannot necessarily deduce that a more general class of problems is NP-hard too. When we compare two problem classes we have to distinguish between the space of inputs and the space of outputs. More exactly, if a problem is NP-hard for a certain class of input parameters $\mathcal{P}$ it is also NP-hard for any less restricted class of parameters $\mathcal{P}'$ (i.e. $\mathcal{P} \subset \mathcal{P}'$). On the other hand, if a problem is NP-hard for valid outputs O, the problem may be solvable in polynomial time for a larger set O' (i.e. $O \subseteq O'$). A prominent example is *integer linear programming* which is NP-hard, whereas *linear programming* is in P (cf. [Pap94]).

Table 2.1: Complexity of Join Ordering Problems

JNLC	*In P* for ASI cost functions. (A consequence of the result for problem JNLA.)
JNLS	*In P* for ASI cost functions. (A consequence of the result for problem JNLA.)
JNLA	*In P* for ASI cost functions [IK84, KBZ86].
JNLG	*NP-complete* for a (complex) cost function for a special nested-loop join [IK84]. Later, the problem has been proven NP-hard even for the simple cost function C_{out} [CM95].
JNBC	*Unknown complexity.*
JNBS	*Unknown complexity.*
JNBA	*Unknown complexity.*
JNBG	*Unknown complexity.*
JCLE	*In P*—just sort the relations by their cardinalities.
JCLC	*Unknown complexity.* (But conjectured to be in P [SM97].)
JCLS	*NP-hard* for the cost function C_{out} [CM95].
JCLA	*NP-hard.* (A consequence of the result for problem JCLS.)
JCLG	*NP-hard.* (A consequence of the result for problem JCLS.)
JCBE	*NP-hard.* ([SM97] and this thesis)
JCBC	*NP-hard.* (A consequence of the result for problem JCBE.)
JCBS	*NP-hard.* (A consequence of the result for problem JCBE.)
JCBA	*NP-hard.* (A consequence of the result for problem JCBE.)
JCBG	*NP-hard.* (A consequence of the result for problem JCBE.)
SNLC	*Unknown complexity.*
SNLS	*Unknown complexity.*
SNLA	*In P* if the set of pushed selections can be "guessed" [SM96] and the cost function has the ASI property.
SNLG	*Unknown complexity.*

Chapter 3

Generation of Optimal Left-deep Execution Plans

3.1 Chain Queries with Joins and Cross Products

One of the simplest problem classes in algebraic optimization is the computation of optimal left-deep trees for chain queries. Chain queries are very common among relational and object-oriented database systems, For example, in object-oriented database systems it is usually more efficient to evaluate a path expression through a sequence of joins over the corresponding extents than by pointer chasing [CD92].

If cross products are not allowed the problem can be solved in polynomial time for cost functions with ASI property [IK84]. However, if cross products are allowed, the complexity status is still unresolved. In this section we investigate the problem and derive two novel algorithms. The first algorithm is correct but we could not prove that it has polynomial time complexity, whereas the second algorithm has polynomial time complexity but we could not prove its correctness. In practice both algorithms yield identical results.

3.1.1 Basic Definitions and Lemmata

An instance of the *join-ordering problem for chain queries* (or a *chain query* for short) is fully described by the following parameters. First, n relations $R_1, \ldots, R_n$ are given. The size of relation R_i $(1 \leq i \leq n)$ is denoted by $|R_i|$ or n_{R_i}. Without loss of generality, we assume that no base relation is empty[1]. Second, a query graph G is given. The relations $R_1, \ldots, R_n$ form the nodes of G and its edges are $\{\{R_i, R_{i+1}\} \mid 1 \leq i < n\}$. That is, the query graph forms a chain:

$$R_1 — R_2 — \ldots — R_n$$

Every edge $\{R_i, R_{i+1}\}$ of the query graph is associated by an according *selectivity* factor $f_{i,i+1} = |R_i \bowtie R_{i+1}|/|R_i \times R_{i+1}|$. We define all other selectivities $f_{i,j} = 1$ for $|i - j| \neq 1$. They correspond to cross products. Also note that selectivities are "symmetric", i.e. $f_{i,j} = f_{j,i}$.

In this section we consider only left-deep processing trees. Since producing left-deep trees is equivalent to fixing a permutation, we will henceforth identify left-deep trees and permutations. There is also a unique correspondence between consecutive parts of a permutation and segments

[1] otherwise optimization would be trivial

of a left-deep tree. Furthermore, if a segment of a left-deep tree does not contain cross products, it uniquely corresponds to a consecutive part of the chain in the query graph. In this case we also speak of (sub)chains or connected (sub)sequences. We say two relations R_i and R_j are *connected* if they are adjacent in G; more general, two sequences s and t are connected, if there exist relations R_i in s and R_j in t such that R_i and R_j are connected. A sequence of relations s is connected if the join graph induced by the relations in s is connected.

Given a chain query, we ask for a permutation $s = r_1 \dots r_n$ of the n relations (i.e. there is a permutation π with $r_i = R_{\pi(i)}$ for $1 \leq i \leq n$) such that for some cost function C_x for the binary join operator the total cost defined as

$$C(s) \quad := \quad \sum_{i=2}^{n} C_x(|r_1 \dots r_{i-1}|, r_i) \tag{3.1}$$

is minimized. By $|r_1 \dots r_i|$, we denote the intermediate result size (cardinality) of joining the relations $r_1, \dots, r_i$. For a single relation r_i, we also write n_{r_i} or simply n_i instead of $|r_i|$ in order to denote its size. In this section, we use the size of the result of a join operator as its cost. That is

$$cost(R_i \bowtie R_j) \quad = \quad f_{i,j} * |R_i| * |R_j|$$

Based on this we can now define the general cost function C_{out} which computes the cost of a join between two relations even if they are intermediate relations.

$$C_{out}(|S|, |T|) \quad = \quad f_{S,T} * |S| * |T|$$

where $f_{S,T}$ denotes the product of all selectivities between relations in S and relations in T, i.e.

$$f_{S,T} := \prod_{R_i \in S, R_j \in T} f_{R_i,R_j}$$

For $C_x \equiv C_{out}$, expression (3.1) reads

$$\begin{aligned} C(r_1 \dots r_n) \quad &= \quad \sum_{i=2}^{n} \prod_{j=1}^{i} n_{r_j} \prod_{k<j} f_{r_k,r_j} \\ &= \quad n_{s_1} \cdot n_{s_2} f_{s_1,s_2} (1 + n_{s_3} f_{s_1,s_3} f_{s_2,s_3} (1 + \dots (1 + n_{s_n} \prod_{j=1}^{n-1} f_{s_j,s_n}) \dots)) \end{aligned}$$

The cost function used in this section is the function that sums up all the intermediate result sizes. This cost function is reasonable if one assumes the intermediate results being written to disk, since then the costs for accessing the disk clearly surpass the CPU costs for checking the join predicate. For a further discussion of the relevance of this cost function see [CM95].

As noted in [OL90] the dynamic programming approach considers $n2^{n-1} - n(n+1)/2$ alternatives for left-deep processing trees with cross products—independently of the query graph and the cost function. The question arises, whether it is possible to lower the complexity in case of simple chain queries and the above type of cost function.

The well-known approach in [IK84, KBZ86] for computing optimal left-deep trees without cross products for acyclic queries is based on the *ASI*[2] *property* of cost functions introduced in [MS79]. Although the cost functions in [IK84] and [KBZ86] do not have the ASI property, the authors decompose the problem into polynomially many subproblems which are subject to tree-like precedence constraints. The precedence constraints ensure that the cost functions of the

[2]Adjacent Sequence Interchange

subproblems now have the ASI property. The remaining problem is to optimize the constrained subproblems under the simpler cost function. Unfortunately, this approach does not work in our case, since no such decomposition seems to exist.

In order to extend the approach of [IK84, KBZ86] to our problem, we first generalize the rank function to a relativized rank. We start by relativizing the cost function. The costs of a sequence s *relative* to a sequence u are defined as follows.

Definition 3.1.1

$$C_u(s) = \begin{cases} 0 & \text{if } u = \epsilon \text{ or } s = \epsilon \\ n_i \prod_{R_j \in u} f_{j,i} & \text{if } u \neq \epsilon \text{ and } s = R_i \\ C_u(s_1) + T_u(s_1) * C_{us_1}(s_2) & \text{if } s = s_1 s_2 \text{ and } s_1 \neq \epsilon,\ s_2 \neq \epsilon \end{cases}$$

where

$$T_u(s) = \prod_{R_i \in s} \Big(\prod_{R_j <_{us} R_i} f_{j,i} \Big) * n_i$$

Here, R_i, R_j denote single relations and s_1, s_2, s, u denote sequences of relations. ϵ is the empty sequence. $R_i <_s R_j$ denotes the predicate that is true if and only if R_i precedes R_j in the sequence s. In the sum above, $R_j <_{us} R_i$ is a shorthand notation for $R_j \in us$, $R_j <_{us} R_i$. As usual, empty products evaluate to 1, consequently $T_u(\epsilon) = 1$.

First, we show that C_ϵ is well-defined:

Lemma 3.1.1 *For all sequences s we have $C_\epsilon(s) = C_{out}(s)$*

Proof We shall use induction on the length of s.
For $s = \epsilon$, we have $C_{out}(\epsilon) = 0 = C_\epsilon(\epsilon)$. For $s = R_i$, we have $C_{out}(R_i) = 0 = C_{R_i}(R_i)$.
Let $s = s'R_i$ with $||s'|| > 1$, then

$$\begin{aligned} C_{out}(s'R_i) &= C_{out}(s') + |s'|\Big(\prod_{R_j <_{s'R_i} R_i} f_{j,i} n_i \Big) \\ &= C_\epsilon(s') + T_\epsilon(s') C_{s'}(R_i) \\ &= C_\epsilon(s'R_i) \end{aligned}$$

□

A couple of things should be noted. First, $T_\epsilon(R_i) = |R_i|$ and $T_\epsilon(s) = |s|$. That is, T_u generalizes the size of a single relation or of a sequence of relations. Second, $C_u(R_i) = T_u(R_i)$ for any single relation R_i. Third, note that $C_u(\epsilon) = 0$ for all u but $C_\epsilon(s) = 0$ only if s does not contain more than one relation. The special case that $C_\epsilon(R) = 0$ for a single relation R causes some problems in the homogeneity of definitions and proofs. Hence, we abandon this case from all definitions and lemmata of this section. This will not be repeated in every definition and lemma but will be implicitly assumed. Further, our two algorithms will be presented in two versions. The first version is simpler and relies on a modified cost function C' and only the second version will apply to the original cost function C. As we will see, C' differs from C exactly in the problematic case in which it is defined as $C'_u(R_i) := |R_i|$. Now, $C'_\epsilon(s) = 0$ holds if and only if $s = \epsilon$ holds. Within subsequent definitions and lemmata, C can also be replaced by C' without changing their validity. Last, we abbreviate C_ϵ by C for convenience.
Next we state some useful properties of the functions C and T.

Lemma 3.1.2 *Let u, v, s be sequences of relations. Then*

$$T_{uv}(s) = f_{v,s} * T_u(s)$$

where $f_{v,s}$ is defined as $f_{v,s} := \prod_{R_i \in v,\, R_j \in s} f_{i,j}$

Proof

$$\begin{aligned} T_{uv}(s) &= \prod_{R_i \in s} \Big(\prod_{R_j <_{uvs} R_i} f_{j,i} \Big) * n_i \\ &\quad \prod_{R_i \in s} \Big(\prod_{R_j <_{us} R_i} f_{j,i} \Big) \Big(\prod_{R_j \in v} f_{j,i} \Big) * n_i \\ &= \prod_{R_i \in s} \Big(\prod_{R_j \in v} f_{j,i} \Big) * \prod_{R_i \in s} \Big(\prod_{R_j <_{us} R_i} f_{j,i} \Big) * n_i \\ &= f_{u,v} * T_u(s) \end{aligned}$$

□

Lemma 3.1.3 *Let $r_1, \ldots, r_n$ be single relations. Then*

$$T_u(r_1 \ldots r_n) = \prod_{i=1}^{n} T_{ur_1 \ldots r_{i-1}}(r_i)$$

Proof Induction on n. For $n = 1$ the assertion being trivial, let $n > 1$ and suppose the result is true for smaller values of n. We have

$$\begin{aligned} T_u(r_1 \ldots r_n) &= \prod_{i=1}^{n} \Big(\prod_{r_j <_{ur_1 \ldots r_n} r_i} f_{r_j,r_i} \Big) * |r_i| \\ &= \prod_{i=1}^{n-1} \Big(\prod_{r_j <_{ur_1 \ldots r_n} r_i} f_{r_j,r_i} \Big) * |r_i| * \Big(\prod_{r_j <_{ur_1 \ldots r_n r_n}} f_{r_j,r_n} \Big) * |r_n| \\ &= \prod_{i=1}^{n-1} T_{ur_1 \ldots r_{i-1}}(r_i) * T_{ur_1 \ldots r_{n-1}}(R_n) \quad \text{(induction hypothesis)} \\ &= \prod_{i=1}^{n} T_{ur_1 \ldots r_{i-1}}(r_i) \end{aligned}$$

□

Corollary 3.1.1 *Let u, r and s be sequences. Then*

$$T_u(rs) = T_u(r) * T_{ur}(s)$$

Lemma 3.1.4 *Let $r_1, \ldots, r_n$ be single relations. Then*

$$C_u(r_1 \ldots r_n) = \sum_{i=1}^{n} T_u(r_1 \ldots r_i)$$

Proof We use induction on n. For $n = 1$ the claim is true as $C_u(r_1) = T_u(r_1)$. Now let $n > 1$ and suppose the result is true for smaller values of n. According to Definition 3.1.1 we have

$$\begin{aligned} C_r(r_1 \ldots r_{n-1} r_n) &= C_r(r_1 \ldots r_{n-1}) + T_u(r_1 \ldots r_{n-1}) C_{ur_1 \ldots r_{n-1}}(r_n) \\ &= C_r(r_1 \ldots r_{n-1}) + T_u(r_1 \ldots r_{n-1}) T_{ur_1 \ldots r_{n-1}}(r_n) \\ &= C_r(r_1 \ldots r_{n-1}) + T_u(r_1 \ldots r_{n-1} r_n) \text{ (Lemma 3.1.1)} \\ &= \sum_{i=1}^{n-1} T_u(r_1 \ldots r_{i-1} r_i) + T_u(r_1 \ldots r_{n-1} r_n) \text{ (induction hypothesis)} \\ &= \sum_{i=1}^{n} T_u(r_1 \ldots r_{i-1} r_i) \end{aligned}$$

□

Lemma 3.1.5 *Let u, v be sequences of relations. If u and v are permutations of each other, then*

$$T_u(t) = T_v(t) \text{ and } C_u(t) = C_v(t)$$

Proof (Sketch) We only prove the statement for two sequences differing exactly in two adjacent relations. The claim follows then by induction on the number of pairs of relations with different relative order in the two relations.

$$\begin{aligned}
&T_{r_1 \dots r_{k-1} r_{k+1} r_k r_{k+2} \dots r_n}(t) \\
&= \prod_{1 \le i \le k-1} (\prod_{j<i} f_{j,i}) * n_i * (\prod_{j<k-1} f_{j,k+1}) * f_{k-1,k+1} * n_{k+1} * \\
&\quad (\prod_{j<k-1} f_{j,k}) * f_{k-1,k} * f_{k+1,k} * n_k * \prod_{k+2 \le i \le n} (\prod_{j<i} f_{j,i}) * n_i \\
&= \prod_{1 \le i \le k-1} (\prod_{j<i} f_{j,i}) * n_i * (\prod_{j<k-1} f_{j,k}) * f_{k-1,k} * n_k * \\
&\quad (\prod_{j<k-1} f_{j,k+1}) * f_{k-1,k+1} * f_{k,k+1} * n_{k+1} * \prod_{k+2 \le i \le n} (\prod_{j<i} f_{j,i}) * n_i \\
&= \prod_{1 \le i \le n} (\prod_{j<i} f_{j,i}) * n_i \\
&= T_{r_1 \dots r_k r_{k+1} \dots r_n}(t)
\end{aligned}$$

We can use this identity to prove an analogue identity for C.

$$\begin{aligned}
C_{r_1 \dots r_{k-1} r_{k+1} r_k r_{k+2} \dots r_n}(t) &= \sum_{i=1}^{n} T_{r_1 \dots r_{k-1} r_{k+1} r_k r_{k+2} \dots r_n}(t_1 \dots t_i) \\
&= \sum_{i=1}^{n} T_{r_1 \dots r_k r_{k+1} r_n}(t_1 \dots t_i) \\
&= C_{r_1 \dots r_k r_{k+1} \dots r_n}(t)
\end{aligned}$$

Based on the last two identities, the claim follows by induction on the number of pairs (r_i, r_j) such that $r_i <_u r_j$ and $r_j <_v r_i$. □

Lemma 3.1.6 *Let u, v be sequences of relations. If there is no connection between relations in s and t then*

$$\begin{aligned}
T_{us}(t) &= T_u(t) \\
C_{us}(t) &= C_u(t)
\end{aligned}$$

Proof Lemma 3.1.2 tells us that $T_{us}(t) = f_{s,t} * T_u(s)$. Since s is not connected to t we know that $f_{s,t} = 1$ and hence $T_{us}(t) = T_u(t)$. Let $t = t_1 \dots t_m$, then

$$\begin{aligned}
C_{us}(t) &= \sum_{i=1}^{m} T_{ust_1 \dots t_{i-1}}(T_i) \\
&= \sum_{i=1}^{m} T_{ut_1 \dots t_{i-1}}(T_i) \text{ (Lemma 3.1.5, Lemma 3.1.6)} \\
&= C_u(t)
\end{aligned}$$

□

Example 1: Consider a chain query involving the relations R_1, R_2, R_3. The parameters are $|R_1| = 1, |R_2| = 100, |R_3| = 10$ and $f_{1,2} = f_{2,3} = 0.9$. The expected size of the query result is independent of the ordering of the relations. Hence we have

$$T(R_1R_2R_3) = \cdots = T(R_3R_2R_1) = 100 * 10 * 1 * .9 * .9 = 810.$$

There are 6 possible orderings of the relations with the following costs

$$\begin{aligned}
C(R_1R_2R_3) &= 1 * 100 * .9 + 1 * 100 * 10 * .9 \cdot .9 = 900 \\
C(R_1R_3R_2) &= 1 * 10 + 1 * 10 * 100 * .9 * .9 = 820 \\
C(R_2R_3R_1) &= 100 * 10 * .9 + 100 * 10 * 1 * .9 * .9 = 1710 \\
C(R_2R_1R_3) &= C(R_1R_2R_3) \\
C(R_3R_1R_2) &= C(R_1R_3R_2) \\
C(R_3R_2R_1) &= C(R_2R_3R_1)
\end{aligned}$$

Note that the cost function is invariant with respect to the order of the first two relations. The minimum over all costs is 820, and the corresponding optimal join ordering is $R_1R_3R_2$.

□

Using the relativized cost function, we can define the relativized rank.

Definition 3.1.2 (rank) *The* rank *of a sequence s relative to a nonempty sequence u is given by*

$$rank_u(s) \quad := \quad \frac{T_u(s) - 1}{C_u(s)}$$

In the special case that s consists of a single relation R_i, the intuition behind the *rank* function becomes transparent. Let f_i be the product of the selectivities between relations in u and R_i. Then $rank_u(R_i) = \frac{f_i|R_i|-1}{f_i|R_i|}$. Hence, the *rank* becomes a function of the form $f(x) = \frac{x-1}{x}$. This function is monotonously increasing in x for $x > 0$. The argument to the function $f(x)$ is (for the computation of the size of a single relation R_i) $f_i|R_i|$. But this is the factor by which the next intermediate result will increase (or decrease). Since we sum up intermediate results, this is an essential number. Furthermore, from the monotonicity of $f(x)$ it follows that $rank_u(R_i) \leq rank_u(R_j)$ if and only if $f_i|R_i| \leq f_j|R_j|$ where f_j is the product of all selectivities between R_j and relations in u.
Note that relations which are not connected to a sequence do not influence the rank of the sequence.

Lemma 3.1.7 *Let u, v, s be sequences of relations where v is not connected to s. Then*

$$rank_{uv}(s) = rank_u(s)$$

Proof The claim is a direct consequence of Lemma 3.1.6. □

Example 1 (cont'd): Given the query in Example 1, the optimal sequence $R_1R_3R_2$ gives rise to the following ranks.

$$\begin{aligned}
rank_{R_1}(R_2) &= \tfrac{T_{R_1}(R_2)-1}{C_{R_1}(R_2)} = \tfrac{100*.9-1}{100*.9} \approx 0.9888 \\
rank_{R_1}(R_3) &= \tfrac{T_{R_1}(R_3)-1}{C_{R_1}(R_3)} = \tfrac{10*1.0-1}{10*1.0} = 0.9 \\
rank_{R_1R_3}(R_2) &= \tfrac{T_{R_1R_3}(R_2)-1}{C_{R_1R_3}(R_2)} = \tfrac{100*.9*.9-1}{100*.9*.9} \approx 0.9877
\end{aligned}$$

Hence, within the optimal sequence, the relation with the smallest rank (here R_3, since $rank_{R_1}(R_3)$ $< rank_{R_1}(R_2)$) is preferred. As the next lemma will show, this is no accident.

□

Using the rank function, one can prove the following lemma.

Lemma 3.1.8 *For sequences*

$$\begin{aligned} S &= r_1 \cdots r_{k-1} r_k r_{k+1} r_{k+2} \cdots r_n \\ S' &= r_1 \cdots r_{k-1} r_{k+1} r_k r_{k+2} \cdots r_n \end{aligned}$$

the following holds:

$$C(S) \leq C(S') \Leftrightarrow rank_u(r_k) \leq rank_u(r_{k+1})$$

Here, $u = r_1 \cdots r_{k-1}$. Equality only holds if it holds on both sides.

Proof Let $u = r_1 \ldots r_{k-1}$ and $v = r_{k+2} \ldots r_n$. According to Definition 3.1.1 and Lemma 3.1.4,

$$\begin{aligned} C(ur_k r_{k+1} v) &= C(u) + T(u) C_u(r_k r_{k+1}) + T(ur_k r_{k+1}) C_{ur_k r_{k+1}}(v) \\ &= C(u) + T(u)[T_u(r_k) + T_u(r_k r_{k+1})] + T(ur_k r_{k+1}) C_{ur_k r_{k+1}}(v) \end{aligned}$$

and

$$\begin{aligned} C(ur_{k+1} r_k v) &= C(u) + T(u) C_u(r_{k+1} r_k) + T(ur_{k+1} r_k) C_{ur_{k+1} r_k}(v) \\ &= C(u) + T(u)[T_u(r_{k+1}) + T_u(r_{k+1} r_k)] + T(ur_{k+1} r_k) C_{ur_{k+1} r_k}(v) \end{aligned}$$

Using Lemma 3.1.5 we have

$$\begin{aligned} C(S) \leq C(S') &\Leftrightarrow C(ur_k r_{k+1} v) \leq C(ur_{k+1} r_k v) \\ &\Leftrightarrow T_u(r_k) \leq T_u(r_{k+1}) \\ &\Leftrightarrow \frac{T_u(r_k) - 1}{T_u(r_k)} \leq \frac{T_u(r_{k+1}) - 1}{T_u(r_{k+1})} \\ &\Leftrightarrow \frac{T_u(r_k) - 1}{C_u(r_k)} \leq \frac{T_u(r_{k+1}) - 1}{C_u(r_{k+1})} \\ &\Leftrightarrow rank_u(r_k) \leq rank_u(r_{k+1}) \end{aligned}$$

□

Example 1 (cont'd): Since the ranks of the relations in Example 1 are ordered with ascending ranks, Lemma 3.1.8 states that, whenever we exchange two adjacent relations, the costs cannot decrease. In fact, we observe that $C(R_1 R_3 R_2) \leq C(R_1 R_2 R_3)$. □

Lemma 3.1.9 *Let u, x and y be subchains, x, y not empty. Then,*

$$C_{ux}(y) \leq f_{x,y} C_u(y).$$

Proof We shall perform induction on the number of connections m between x and y.
If there are no connections between x and y, we have $f_{x,y} = 1$. Hence, by Lemma 3.1.6 $C_{ux}(y) = C_u(y) = f_{x,y} C_u(y)$.
Now, assume that there are exactly m connections between x and y and suppose the claim is true for smaller values of m. Let $y = vrw$ where r is a single relation and v and w are subchains. There is one connection between r and x and there are $m - 1$ connections between x and w. x is not connected to v. We have

$$
\begin{array}{rcll}
C_{ux}(y) & = & C_{ux}(vrw) & \\
 & = & C_{ux}(v) + T_{ux}(v)C_{uxv}(rw) & \text{(Def. 3.1.1)} \\
 & = & C_{ux}(v) + T_{ux}(v)C_{uxv}(r) + T_{ux}(v)T_{uxv}(r)C_{uxvr}(w) & \text{(Def. 3.1.1)} \\
 & = & C_u(v) + T_u(v)C_{uv}(r)f_{x,r} + T_u(v)T_{uv}(r)f_{x,r}C_{uxvr}(w) & \text{(Lemma 3.1.2)} \\
 & \leq & C_u(v) + f_{x,r}T_u(v)[C_{uv}(r) + T_{uv}(r)C_{uvr}(w)f_{x,w}] & \text{(induction hypothesis)} \\
 & \leq & C_u(v) + f_{x,r}f_{x,w}T_u(v)C_{uv}(rw) & \text{(Def. 3.1.1)} \\
 & = & C_u(v) + f_{x,y}T_u(v)C_{uv}(rw) & (f_{x,v} = 1) \\
 & \leq & f_{x,y}[C_u(v) + T_u(v)C_{uv}(rw)] & \\
 & = & f_{x,y}C_u(vrw) & \text{(Def. 3.1.1)} \\
 & = & f_{x,y}C_u(y) &
\end{array}
$$

This proves the claim. □

The next lemma provides a condition to decide whether it is safe to interchange two adjacent subchains.

Lemma 3.1.10 *Let u, x and y be subchains, x, y not empty. Then we have*

$$rank_u(x) \leq rank_{ux}(y) \Rightarrow C_u(xy) \leq C_u(yx).$$

Furthermore, if x and y are not interconnected, the reverse direction also holds, i.e.

$$rank_u(x) \leq rank_u(y) \Leftrightarrow C_u(xy) \leq C_u(yx).$$

Proof We have

$$
\begin{array}{rcl}
C(uxy) \leq C(uyx) & \Leftrightarrow & C_u(xy) \leq C_u(yx) \\
 & \Leftrightarrow & C_u(x) + T_u(x)C_{ux}(y) \leq C_u(y) + T_u(y)C_{uy}(x)
\end{array} \tag{3.2}
$$

and

$$
\begin{array}{rcl}
rank_u(x) \leq rank_{ux}(y) & \Leftrightarrow & \dfrac{T_u(x) - 1}{C_u(x)} \leq \dfrac{T_{ux}(y) - 1}{C_{ux}(y)} \\
 & \Leftrightarrow & C_u(x) + T_u(x)C_{ux}(y) \leq C_{ux}(y) + T_{ux}(y)C_u(x).
\end{array} \tag{3.3}
$$

First consider the case where x is not connected to y. Using Lemma 3.1.6, the inequalities (3.2) and (3.3) simplify to

$$
\begin{array}{rcl}
C(uxy) \leq C(uyx) & \Leftrightarrow & C_u(x) + T_u(x)C_u(y) \leq C_u(y) + T_u(y)C_u(x) \\
rank_u(x) \leq rank_{ux}(y) & \Leftrightarrow & C_u(x) + T_u(x)C_u(y) \leq C_u(y) + T_u(y)C_u(x),
\end{array}
$$

and the claim follows.
Now, let x be connected to y, and assume that $rank_u(x) \leq rank_{ux}(y)$. By Lemma 3.1.9,

$$
\begin{array}{rcl}
C_{ux}(y) & \leq & f_{x,y}C_u(y) \\
 & \leq & C_u(y)
\end{array}
$$

and

$$C_{uy}(x) \leq f_{y,x}C_u x.$$

Hence

$$\begin{aligned} T_{ux}(y)C_u(x) &= f_{x,y}T_u(y)C_u(x) \\ &= T_u(y)f_{y,x}C_u(x) \\ &\leq T_u(y)C_{uy}(x). \end{aligned}$$

So

$$C_u(x) + T_u(x)C_{ux}(y) \leq C_u(y) + T_u(y)C_{uy}(x)$$

holds, from which follows

$$C_u(xy) \leq C_u(yx).$$

□

Next we define the notion of a contradictory chain which will be essential to our algorithms. The subsequent lemmata will allow us to cut down the search space to be explored by any optimization algorithm. For the lemmata, we need the essential definition of *contradictory chains*.

Definition 3.1.3 (contradictory pair of subchains) *Let u, x, y be nonempty sequences. We call (x, y) a* contradictory pair of subchains *if and only if*

$$C_u(xy) \leq C_u(yx) \text{ and } rank_u(x) > rank_{ux}(y)$$

A special case occurs when x and y are single relations. Then the above condition simplifies to

$$rank_{ux}(y) < rank_u(x) \leq rank_u(y)$$

To explain the intuition behind the definition of contradictory subchains we need another example.

Example 2: Suppose a chain query involving R_1, R_2, R_3 is given. The relation sizes are $|R_1| = 1, |R_2| = |R_3| = 10$ and the selectivities are $f_{1,2} = 0.5, f_{2,3} = 0.2$. Consider the sequences $R_1R_2R_3$ and $R_1R_3R_2$ which differ in the order of the last two relations. We have

$$\begin{aligned} rank_{R_1}(R_2) &= 0.8 \\ rank_{R_1R_2}(R_3) &= 0.5 \\ rank_{R_1}(R_3) &= 0.9 \\ rank_{R_1R_3}(R_2) &= 0 \end{aligned}$$

and

$$\begin{aligned} C(R_1R_2R_3) &= 15 \\ C(R_1R_3R_2) &= 20 \end{aligned}$$

Hence,

$$\begin{aligned} rank_{R_1}(R_2) &> rank_{R_1R_2}(R_3) \\ rank_{R_1}(R_3) &> rank_{R_1R_3}(R_2) \\ C(R_1R_2R_3) &< C(R_1R_3R_2) \end{aligned}$$

and (R_2, R_3) is a contradictory pair within $R_1R_2R_3$. □

The next (obvious) lemma states that contradictory chains are necessarily connected.

Lemma 3.1.11 *If there is no connection between two subchains x and y, then they cannot build a contradictory pair (x, y).*

Proof Assume that (x, y) is a contradictory pair preceded by the relations u. According to Definition 3.1.3, $C_u(xy) \leq C_u(yx)$ and $rank_u(x) > rank_{ux}(y)$. Since x is not connected to y, we have $rank_{ux}(y) = rank_u(y)$ and therefore $rank_u(x) > rank_u(y)$. Now, Lemma 3.1.10 yields, $C(uxy) = C(u) + T(u)C_u(xy) > C(uyx) = C(u) + T(u) * C_u(yx)$, a contradiction. □

Now we present the fact that between a contradictory pair of relations there cannot be any other relation not connected to them between them without increasing cost.

Lemma 3.1.12 *Let $S = usvtw$ be a sequence. If there is no connection between relations in s and v and relations in v and t, and $rank_u(s) \geq rank_{us}(t)$, then there exists a sequence S' of not higher cost, where s immediately precedes t.*

Proof If $rank_u(v) \leq rank_u(s)$, we can safely exchange s and v (Lemma 3.1.10). If $rank_u(v) > rank_u(s)$, then

$$rank_{us}(v) = rank_u(v) > rank_u(s) \geq rank_u(t) \geq rank_{us}(t)$$

Hence, we can exchange v and t without increasing the costs (Lemma 3.1.10) □

Example 3: Consider five relations $R_1, \ldots, R_5$. The relation sizes are $|R_1| = 1$, $|R_2| = |R_3| = |R_4| = 8$, and $|R_5| = 2$. The selectivities are $f_{1,2} = \frac{1}{2}$, $f_{2,3} = \frac{1}{4}$, $f_{3,4} = \frac{1}{8}$, and $f_{4,5} = \frac{1}{2}$. Relation R_5 is not connected to relations R_2 and R_3. Further, within the sequence $R_1R_2R_5R_3R_4$ relations R_2 and R_3 have contradictory ranks: $rank_{R_1}(R_2) = \frac{4-1}{4} = \frac{3}{4}$ and $rank_{R_1R_2R_5}(R_3) = \frac{2-1}{2} = \frac{1}{2}$. Hence, at least one of $R_1R_5R_2R_3R_4$ and $R_1R_2R_3R_5R_4$ must be of no greater cost than $R_1R_2R_5R_3R_4$. This is indeed the case:

$$\begin{array}{lcccl} C(R_1R_2R_3R_5R_4) & = & 4+8+16+8 & = & 36 \\ C(R_1R_2R_5R_3R_4) & = & 4+8+16+8 & = & 36 \\ C(R_1R_5R_2R_3R_4) & = & 2+8+16+8 & = & 34 \end{array}$$

□

The next lemma shows that, if there exist two sequences of single rank-sorted relations, then their costs as well as their ranks are necessarily equal.

Lemma 3.1.13 *Let $S = x_1 \cdots x_n$ and $S' = y_1 \cdots y_n$ be two different rank-sorted chains containing exactly the relations $R_1, \ldots, R_n$, i.e.*

$$\begin{array}{rcl} rank_{x_1 \cdots x_{i-1}}(x_i) & \leq & rank_{x_1 \cdots x_i}(x_{i+1}) \quad \textit{for all } 1 \leq i \leq n, \\ rank_{y_1 \cdots y_{i-1}}(y_i) & \leq & rank_{y_1 \cdots y_i}(y_{i+1}) \quad \textit{for all } 1 \leq i \leq n, \end{array}$$

then S and S' have equal costs and, furthermore

$$rank_{x_1 \cdots x_{i-1}}(x_i) \quad = \quad rank_{y_1 \cdots y_{i-1}}(y_i) \quad \textit{for all } 1 < i \leq n$$

Proof We shall use induction on the length of the subsequence on which S and S' differ.
If S and S' do not differ the claim is trivially true. Otherwise, S and S' can be represented as

$$S = x\mathbf{a}v\mathbf{b}v'y$$

$$S' = x\mathbf{b}w\mathbf{a}w'y$$

Here, x denotes a maximal common prefix of S and S' and y denotes a maximal common suffix of S and S'. a and b are two different single relations and the subsequence vv' is a permutation of the subsequence ww'. Due to the ascending local ranks, the following inequalities hold

$$rank_x(a) \leq \cdots \leq rank_{xav}(b)$$

$$rank_x(b) \leq \cdots \leq rank_{xbw}(a)$$

and hence

$$f_{x,a}n_a \leq f_{xav,b}n_b$$
$$f_{x,b}n_b \leq f_{xbw,a}n_a.$$

Combining the two inequalities yields

$$f_{x,a}n_a \leq f_{x,b}f_{av,b}n_b \leq f_{av,b}f_{xbw,a}n_a,$$

from which we can deduce

$$f_{av,b} = f_{a,bw} = 1.$$

So av is not connected to b and bw is not connected to a. Furthermore, the latter inequalities reduce to

$$f_{x,a}n_a = f_{x,b}n_b$$

yielding

$$rank_{xbw}(a) = rank_x(a) = rank_x(b) = rank_{xav}(b).$$

Now, if neither v nor w is the empty sequence and $v = v_1 \cdots v_r$, $w = w_1 \cdots w_s$, we have

$$rank_x(a) = rank_{xa}(v_1) = rank_{xav_1}(v_2) = \cdots = rank_{xav}(b),$$

$$rank_x(b) = rank_{xb}(w_1) = rank_{xbw_1}(w_2) = \cdots = rank_{xbw}(a),$$

and since the relations in $av_1 \cdots v_r b$ and $bw_1 \cdots w_s a$ have identical local ranks, we can transform them into abv and abw resp., by successively interchanging adjacent relations. Note, that b has no connections to relations in v, a has no connections to relations in w and a and b are unconnected. This transformation does neither change the costs of the two chains nor does it change any local ranks! The new chains have the form

$$S_1 = xabvv'y, \text{ and } S_1' = xabww'y.$$

By the induction hypothesis we know that

$$C(xabvv') = C(xabww'),$$

and that all local ranks of $xabvv'$ and $xabww'$ are pairwise identical. Since the costs and local ranks of S and S_1 and of S' and S_1' do not differ, $Cost(S) = Cost(S')$ as claimed. □

Consider the problem of merging two optimal unconnected chains. If we knew that the ranks of relations in an optimal chain are always sorted in ascending order, we could use the classical merge procedure to combine the two chains. The resulting chain would also be rank-sorted in ascending order and according to Lemma 3.1.13 it would be optimal. Unfortunately, this does not work, since there are optimal chains whose ranks are not sorted in ascending order: those containing sequences with contradictory ranks.

Now, as shown in Lemma 3.1.12, between contradictory pairs of relations there cannot be any other relation not connected to them. Hence, in the merging process, we have to take care that we do not merge a contradictory pair of relations with a relation not connected to the pair. In order to achieve this, we just tie the relations of a contradictory subchain together by building a *compound relation*. Assume we tie together relations $r_1, \ldots, r_n$ to a new relation $r_{1,\ldots,n}$. Then we define the size of $r_{1,\ldots,n}$ as $|r_{1,\ldots,n}| = |r_1 \bowtie \ldots \bowtie r_n|$. As we shall see later in this section, compound relations correspond with connected subchains of the join graph. As a consequence, a relation r_{n+1} can have at most one connection to the relations in a compound relation $r_{1,\ldots,n}$. We define the selectivity factor $f_{r_{1,\ldots,n},r_k}$ between r_k and $r_{1,\ldots,n}$ as $f_{r_{1,\ldots,n},r_k} = f_{i,k}$.

If we tie together contradictory pairs, the resulting chain of compound relations does still not have to be rank-sorted with respect to the compound relations. To overcome this, we iterate the

process of tying contradictory pairs of compound relations together until the sequence of compound relations is rank-sorted, which will eventually be the case. This process is called *normalization* in [KBZ86]. Actually, we need a generalized version of normalization which uses relativized costs and ranks. A description of the normalization algorithm is given below.

```
 1  proc normalize (p,s)
 2       while there exist subsequences u, v (u ≠ ε) and
 3            compound relations x, y such that s = uxyv
 4              and C_pu(xy) ≤ C_pu(yx)
 5              and rank_pu(x) > rank_pux(y) do
 6                  replace xy in s by a compound relation (x, y);
 7       od
 8       resolve all but outermost tyings in s
 9       return (p, s);
10  end
```

The compound relations in the result of the procedure `normalize` are called *contradictory chains*. A *maximal contradictory subchain* is a contradictory subchain that cannot be made longer by further tying steps. The cost, size and rank functions can now be extended to sequences containing compound relations in a straightforward way. We define the cost of a sequence containing compound relations to be identical with the cost of the corresponding sequence without any tyings. The size and rank functions are defined analogously. Resolving all (or some) of the tyings introduced in the procedure `normalize` is called *de-normalization*. Actually, we are not interested in the particular recursive structure of a contradictory subchain but in the fact that the subsequence is a contradictory subchain. This is why we get rid of all but the outermost tyings (parenthesis) at the end of `normalize`.

Our next milestone will be to prove that the indeterministic procedure `normalize` is well-defined (Theorem 3.1.1), but this will be a long way to go. We start with a simple example application.

Example 4: Consider a chain $S = R_{10}R_5R_4R_8R_7R_6R_9R_3R_1R_2$. For the sake of simplicity, let us assume that S cannot be improved by interchanging two adjacent connected subchains. Furthermore, assume that the conditions

$$\begin{aligned}
rank(R_5) &> rank_{R_5}(R_4)\\
rank(R_8) &> rank_{R_8}(R_7)\\
rank_{R_8}(R_7) &> rank_{R_5,R_7}(R_6)\\
rank_{R_5}(R_8R_7R_6) &> rank_{R_8,R_{10}}(R_9)\\
rank(R_1) &> rank_{R_1,R_3}(R_2)\\
rank_{R_4}(R_3) &> rank_{R_3}(R_1R_2)\\
rank(R_5) &< rank_{R_5,R_7}(R_6)
\end{aligned}$$

hold. A possible computation of `normalize` could look as follows:

$$(R_{10})\,\underline{(R_5)}\,\underline{(R_4)}\,(R_8)\,(R_7)\,(R_6)\,(R_9)\,(R_3)\,(R_1)\,(R_2)$$

$$\Downarrow$$

$$(R_{10})\,(R_5R_4)\,\underline{(R_8)}\,\underline{(R_7)}\,(R_6)\,(R_9)\,(R_3)\,(R_1)\,(R_2)$$

$$\Downarrow$$
$$(R_{10})\,(R_5R_4)\,\underline{(R_8R_7)}\,\underline{(R_6)}\,(R_9)\,(R_3)\,(R_1)\,(R_2)$$
$$\Downarrow$$
$$(R_{10})\,(R_5R_4)\,\underline{((R_8R_7)R_6)}\,\underline{(R_9)}\,(R_3)\,(R_1)\,(R_2)$$
$$\Downarrow$$
$$(R_{10})\,(R_5R_4)\,(((R_8R_7)R_6)R_9)\,(R_3)\,\underline{(R_1)}\,\underline{(R_2)}$$
$$\Downarrow$$
$$(R_{10})\,(R_5R_4)\,(((R_8R_7)R_6)R_9)\,\underline{(R_3)}\,\underline{(R_1R_2)}$$
$$\Downarrow$$
$$(R_{10})\,(R_5R_4)\,(((R_8R_7)R_6)R_9)\,(R_3(R_1R_2))$$

Since $rank(R_5) < rank_{R_5,R_7}(R_6)$ no further tyings are possible. In the last step of `normalize` the chain is de-normalized (flattened), yielding

$$(R_{10})\,(R_5R_4)\,(R_8R_7R_6R_9)\,(R_3R_1R_2).$$

□

The next lemma states that contradictory subchains are always connected.

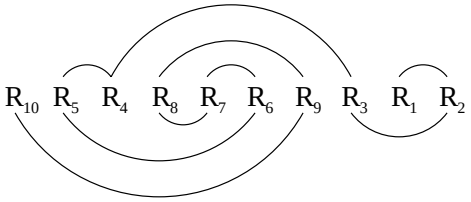
Lemma 3.1.14 *Contradictory subchains correspond to connected subgraphs of the join graph.*

Proof The proof is by induction on the size of the contradictory subchains. The claim is trivially true if the contradictory subchain is a single relation.
Now, suppose that the claim holds for all contradictory subchains with at most n relations. Consider a contradictory subchain s of size $n+1$. Since s is a contradictory subchain, there exist subchains x, y such that $z = xy$ and (x, y) builds a contradictory pair. By the induction assumption, both x and y correspond to connected subgraphs of the join graph. Since, according to Lemma 3.1.11, x is connected to y, s corresponds to a connected subgraph of the join graph. □

The Figure below shows the winding connections of the chain in the last example. Due to the nested structure of contradictory subchains, no contradictory subchain s can have the property that there exists an $i \in \{1, \ldots, n-4\}$ such that

$$R_{i+1} \prec_s R_{i+3} \prec_s R_i \prec_s R_{i+2} \quad \text{or}$$
$$R_{i+2} \prec_s R_i \prec_s R_{i+3} \prec_s R_{i+1}.$$

Here, $R \prec_s R'$ means that relation R precedes relation R' in s.



The following simple observation is central to our algorithms: every chain can be decomposed into a sequence of adjacent maximal contradictory subchains. For convenience, we often speak

of chains instead of subchains and of contradictory chains instead of maximal contradictory subchains. The meaning should be clear from the context. Further, we note that the decomposition into adjacent maximal contradictory subchains is not unique. For example, consider an optimal subchain $r_1r_2r_3$ and a sequence u of preceding relations. If

$$rank_u(r_1) > rank_{ur_1}(r_2) > rank_{ur_1r_2}(r_3)$$

one can easily show that both $(r_1,(r_2,r_3))$ and $((r_1,r_2),r_3)$ are contradictory subchains. In the following we are only interested in contradictory subchains which are *optimal* and in this case the condition $C_u(xy) \leq C_u(yx)$ of Def. 3.1.3 (contradictory pair of subchains) is certainly true[3] and can therefore be neglected.

Example 5: Suppose we want to normalize the chain $R_1R_2R_3R_4$ with respect to the preceding relation R_0. Let the parameters of the chain query problem be $R_0 = 40, R_1 = 90, R_2 = 40, R_3 = 90, R_4 = 30$ and $f_{0,1} = 0.4, f_{1,2} = 0.7, f_{2,3} = 0.6, f_{3,4} = 0.5$. Before entering the while-loop of the normalizing procedure, each relation in the input sequence is replaced by a corresponding compound relation, i.e. $R_1R_2R_3R_4$ becomes $(R_1)(R_2)(R_3)(R_4)$. The ranks of the relations are

$$\begin{aligned} rank_{R_0}(R_1) &\approx 0.9722, \\ rank_{R_1}(R_2) &\approx 0.9643, \\ rank_{R_2}(R_3) &\approx 0.9815, \\ rank_{R_3}(R_4) &\approx 0.9333 \end{aligned}$$

Hence, $rank_{R_0}(R_1) > rank_{R_1}(R_2) < rank_{R_2}(R_3) > rank_{R_3}(R_4)$ and possible candidates for contradictory pairs are (R_1,R_2) and (R_3,R_4). Since $C_{R_0}(R_1R_2) = 1044.0 < 1048.0 = C_{R_0}(R_2R_1)$ and $C_{R_2}(R_3R_4) = 864.0 > 840.0 = C_{R_2}(R_4R_3)$, only (R_1,R_2) is an actual contradictory pair. After the first pass of the while-loop the sequence is replaced by the new sequence $((R_1)(R_2))(R_3)(R_4)$ with the new ranks being

$$\begin{aligned} rank_{R_0}(R_1R_2) &\approx 0.9646, \\ rank_{R_2}(R_3) &\approx 0.9815, \\ rank_{R_3}(R_4) &\approx 0.9333 \end{aligned}$$

There are no new candidates for contradictory pairs and the while-loop terminates. The next step is de-normalization which removes all but the outermost brackets in compound relations. The result of the normalization algorithm is $(R_1R_2)(R_3)(R_4)$.

Now suppose we want to normalize the chain $R_1R_2R_4R_3$. As the reader may verify, the sequence $R_1R_2R_4R_3$ has minimal cost among all sequences starting with R_0 and involving the relations $\{R_1,R_2,R_3,R_4\}$. Note that when using this sequence we need not check the cost condition of contradictory pairs since any subsequence of an optimal sequence has to be optimal too. The local ranks are

$$\begin{aligned} rank_{R_0}(R_1) &\approx 0.9722, \\ rank_{R_1}(R_2) &\approx 0.9643, \\ rank(R_4) &\approx 0.9667, \\ rank_{R_2R_4}(R_3) &\approx 0.9630 \end{aligned}$$

Therefore, $(R1,R_2)$ and (R_4,R_3) are contradictory pairs and the result of the first pass of the loop is $((R_1)(R_2))((R_4)(R_3))$. The new ranks are

$$\begin{aligned} rank_{R_0}(R_1R_2) &\approx 0.9646, \\ rank_{R_2}(R_4R_3) &\approx 0.9631 \end{aligned}$$

[3]Otherwise uyx would be cheaper than uxy, a contradiction to the optimality of uxy.

There is exactly one new contradictory pair and the result of the second pass is $(((R_1)(R_2))((R_4)(R_3)))$. After the second pass the loop terminates since there are no more relations to group. After resolving all but the outermost tyings the result of the normalization algorithm is $(R_1R_2R_4R_3)$. □

Next we will show that for the case of optimal subchains xy, where the cost condition $C_u(xy) \leq C_u(yx)$ is obviously satisfied, the indeterministically defined normalization process is well-defined, that is, if S is optimal, `normalize(P,S)` will always terminate with a unique "flat" decomposition of S into maximal contradictory subchains (flat means that we remove all but the outermost parenthesis, e.g. $(R_1R_2)(((R_5R_4)R_3)R_6)$ becomes $(R_1R_2)(R_5R_4R_3R_6)$). In order to show that the normalization process is well-defined we first need some results concerning the ranks of contradictory subchains.

Lemma 3.1.15
Let r, s and u be nonempty sequences of relations. Then, we have

$$\min(rank_u(r), rank_{ur}(s)) \;\leq\; rank_u(rs) \;\leq\; \max(rank_u(r), rank_{ur}(s))$$

Proof Using Def. 3.1.1 and Def. 3.1.2 we have

$$\begin{aligned}
rank_u(rs) &\leq rank_u(r)\\
\Leftrightarrow \frac{T_u(r)T_{ur}(s)-1}{C_u(r)+T_u(r)C_{ur}(s)} &\leq \frac{T_u(r)-1}{C_u(r)}\\
\Leftrightarrow T_u(r)T_{ur}(s)C_u(r)-C_u(r) &\leq T_u(r)C_u(r)+T_u^2(r)C_{ur}(s)-C_u(r)-T_u(r)C_{ur}(s)\\
\Leftrightarrow T_{ur}(s)C_u(r)-C_u(r) &\leq T_u(r)C_{ur}(s)-C_{ur}(s)\\
\Leftrightarrow \frac{T_{ur}(s)-1}{C_{ur}(s)} &\leq \frac{T_u(r)-1}{C_u(r)}\\
\Leftrightarrow rank_{ur}(s) &\leq rank_u(r)
\end{aligned}$$

Note that $C_u(r) > 0$ and $C_{ur}(s) > 0$. On the other hand, we have

$$\begin{aligned}
rank_u(rs) &\geq rank_u(r)\\
\Leftrightarrow \frac{T_u(r)T_{ur}(s)-1}{C_u(r)+T_u(r)C_{ur}(s)} &\geq \frac{T_{ur}(s)-1}{C_{ur}(s)}\\
\Leftrightarrow T_u(r)T_{ur}(s)C_{ur}(s)-C_{ur}(s) &\geq C_u(r)T_{ur}(s)+T_u(r)T_{ur}(s)C_{ur}(s)\\
&\quad -C_u(r)-T_u(r)C_{ur}(s)\\
\Leftrightarrow T_u(r)C_{ur}(s)-C_{ur}(s) &\geq T_{ur}(s)C_u(r)-C_u(r)\\
\Leftrightarrow \frac{T_u(r)-1}{C_u(r)} &\geq \frac{T_{ur}(s)-1}{C_{ur}(s)}\\
\Leftrightarrow rank_u(r) &\geq rank_{ur}(s)
\end{aligned}$$

This proves the claim. □

Let us introduce the notion of a *decomposition tree*.

Definition 3.1.4 (decomposition tree) *A* decomposition tree T *for a chain s and a prefix u is inductively defined as follows. The nodes of T are labeled with pairs (x, y), where x is a prefix and y is a subchain. The smallest decomposition tree consists of a single node labeled with (u, s). A decomposition tree can be enlarged by selecting an arbitrary leaf node with label (u', s') such that $|s'| > 1$ and adding a left successor node labeled with (u, s_1) and a right successor node labeled with (us_1, s_2), where $s' = s_1s_2$. If $(u_1, s_1), \ldots, (u_k, s_k)$ are the leaf labels of T in left-to-right order, we call $s = s_1s_2 \ldots s_k$ the decomposition of s (into adjacent subchains) defined by T.*

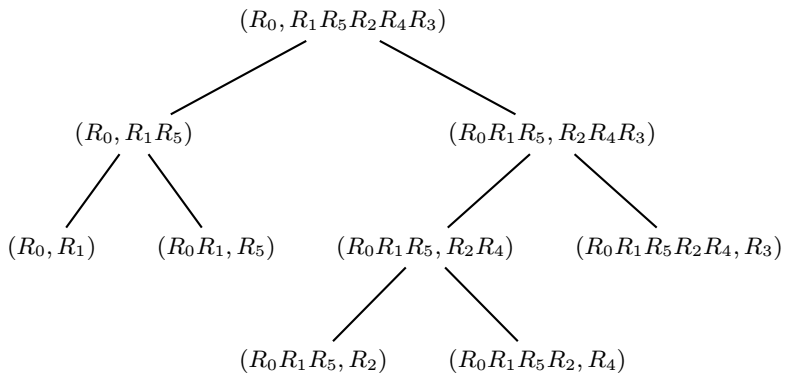


Figure 3.1: A decompostion tree of the chain $R_1R_5R_2R_4R_3$ and the prefix R_0.

The next lemma states that the rank of any chain s is bounded by the extreme values of the ranks of the subchains in a decomposition of s into adjacent subchains.

Lemma 3.1.16 *Let s be a chain and u be an associated prefix. If $s_1 \dots s_k$ is an arbitrary decomposition of s into adjacent subchains, we have*

$$\min_{1 \le i \le k} rank_{us_1 \dots s_{i-1}}(s_i) \le rank_u(s) \le \max_{1 \le i \le k} rank_{us_1 \dots s_{i-1}}(s_i)$$

Proof Let T be a decomposition tree for the decomposition $s_1 \dots s_k$. The proof is by induction on the height of T. If T has height 0, T consists of one node labeled with (u, s), and the claim is obviously true. Now assume that the claim is true for all decomposition trees of height less than n, for some $n > 0$. Consider a tree T of height n. Let (u, s) be the label of the root of T and let $s_1 \dots s_k$ be the decomposition defined by T. Denote the left and right subtrees of T by T_l and T_r, respectively. Let (u_l, s_l) and (u_r, s_r) be the root labels of T_l and T_r, respectively. Assume that $s_l = s_1 \dots s_j$ and $s_r = s_{j+1} \dots s_k$ are the decompositions of T_l and T_r for some $1 \le j < k$. Since both T_l and T_r have height strictly less than n, we can apply the induction hypothesis, obtaining

$$\begin{aligned}
\min_{1 \le i \le j} rank_{us_1 \dots s_{i-1}}(s_i) \;&\le\; rank_{u_l}(s_l) \;\le\; \max_{1 \le i \le j} rank_{us_1 \dots s_{i-1}}(s_i) \\
\min_{j < i \le k} rank_{us_1 \dots s_{i-1}}(s_i) \;&\le\; rank_{u_r}(s_r) \;\le\; \max_{j < i \le k} rank_{us_1 \dots s_{i-1}}(s_i).
\end{aligned}$$

By Lemma 3.1.15, we have

$$\min(rank_{u_l}(s_l), rank_{u_r}(s_r)) \;\le\; rank_u(s) \;\le\; \max(rank_{u_l}(s_l), rank_{u_r}(s_r))$$

and hence

$$\begin{aligned}
&\min(\min_{1 \le i \le j} rank_{us_1 \dots s_{i-1}}(s_i), \min_{j < i \le k} rank_{us_1 \dots s_{i-1}}(s_i)) \;\le\; rank_u(s) \;\le \\
&\max(\max_{1 \le i \le j} rank_{us_1 \dots s_{i-1}}(s_i), \max_{j < i \le k} rank_{us_1 \dots s_{i-1}}(s_i))
\end{aligned}$$

or

$$\min_{1 \le i \le k} rank_{us_1 \dots s_{i-1}}(s_i) \;\le\; rank_u(s) \;\le\; \max_{1 \le i \le k} rank_{us_1 \dots s_{i-1}}(s_i).$$

□

Corollary 3.1.2 *Let u and s be subchains and let T be a decomposition tree for an arbitrary decomposition of s with respect to the prefix u. Let $s_1 \dots s_k$ be the decomposition of s defined by*

T. If (v, w) is the label of an arbitrary node of T and $w = s_i \dots s_j$ for some $1 \leq i \leq k$, then we have

$$\min_{i \leq e \leq j} rank_{us_1 \dots s_{e-1}}(s_e) \;\leq\; rank_v(w) \;\leq\; \max_{i \leq e \leq j} rank_{us_1 \dots s_{e-1}}(s_e)$$

Lemma 3.1.17
Let u, r, x, y and s be sequences of relations. If (x, y) is a contradictory pair in an optimal sequence $urxys$ and both (r, x) and (y, s) are contradictory pairs, then (rx, ys) is a contradictory pair too.

Proof By Lemma 3.1.15 we have

$$rank_u(rx) \geq rank_{ur}(x) > rank_{urx}(y) \geq rank_{urx}(ys).$$

Furthermore, since $urxys$ is an optimal sequence, $C_u(rx) \leq C_{urx}(ys)$ must hold, and the claim follows. □

Lemma 3.1.17 essentially says that the possibility to group a pair of contradictory subchains does not vanish if we extend the left or right subchain.

In a sequence of relations $r = r_1 \dots r_n$, each possible grouping of two adjacent contradictory subchains uniquely corresponds to a pair of adjacent relations $r_i r_{i+1}$ in r. The following definition introduces a name to such a pairs of relations.

Definition 3.1.5 *(connecting point, cp)*
Let $s = uxyw$ be a sequence of relations such that (x, y) is a contradictory pair of subchains in s. Let r_1 denote the last relation of x and r_2 denote the first relation of y, respectively. Then we call the pair (r_1, r_2) a connecting point *in s. Two connecting points (r_1, r_2) and (r_3, r_4)* overlap *if and only if $r_1 = r_4$ or $r_2 = r_3$.*

The nondeterministic computation of the procedure `normalize` can be visualized in form of a *computation tree* which we define next.

Definition 3.1.6 *(computation tree)*
A computation tree *for a subchain s is a rooted tree whose edges are labeled with connection points. The nodes of the tree uniquely correspond to intermediate results of the normalization process as follows. The root of the computation tree corresponds to the original subchain s. For every node v corresponding to a sequence of compound relations $r_1, \dots, r_k$ there is an edge labeled α leaving v if and only if (r_i, r_{i+1}) builds a contradictory pair for some $1 \leq i < k$, and α is the cp of (r_i, r_{i+1}). By a* path *in the computation tree we mean a path starting at the root of the tree. If the path ends in a leaf, we call it a* maximal path.

Since there are at most $n - 1$ different cp's and all the cp's on a path are different, a node of depth l has at most $n - l$ successor nodes. Figure 3.2 shows an example of a computation tree.

Next we summarize some useful facts about sequences of cp's and decompositions into contradictory subchains. For this let ϕ_1 and ϕ_2 be paths consisting of m cp's for a chain c of n relations S. We denote a prefix of length k of ϕ_i $(i = 1, 2)$ by $pref_k(\phi_i)$ and the set of cp's in $pref_k(\phi_i)$ by $cp(pref_k(\phi_i))$. Then, the following facts hold.

Fact 3.1.1 *$pref_k(\phi_1)$ uniquely corresponds to a decomposition of c into $n - k$ contradictory subchains.*

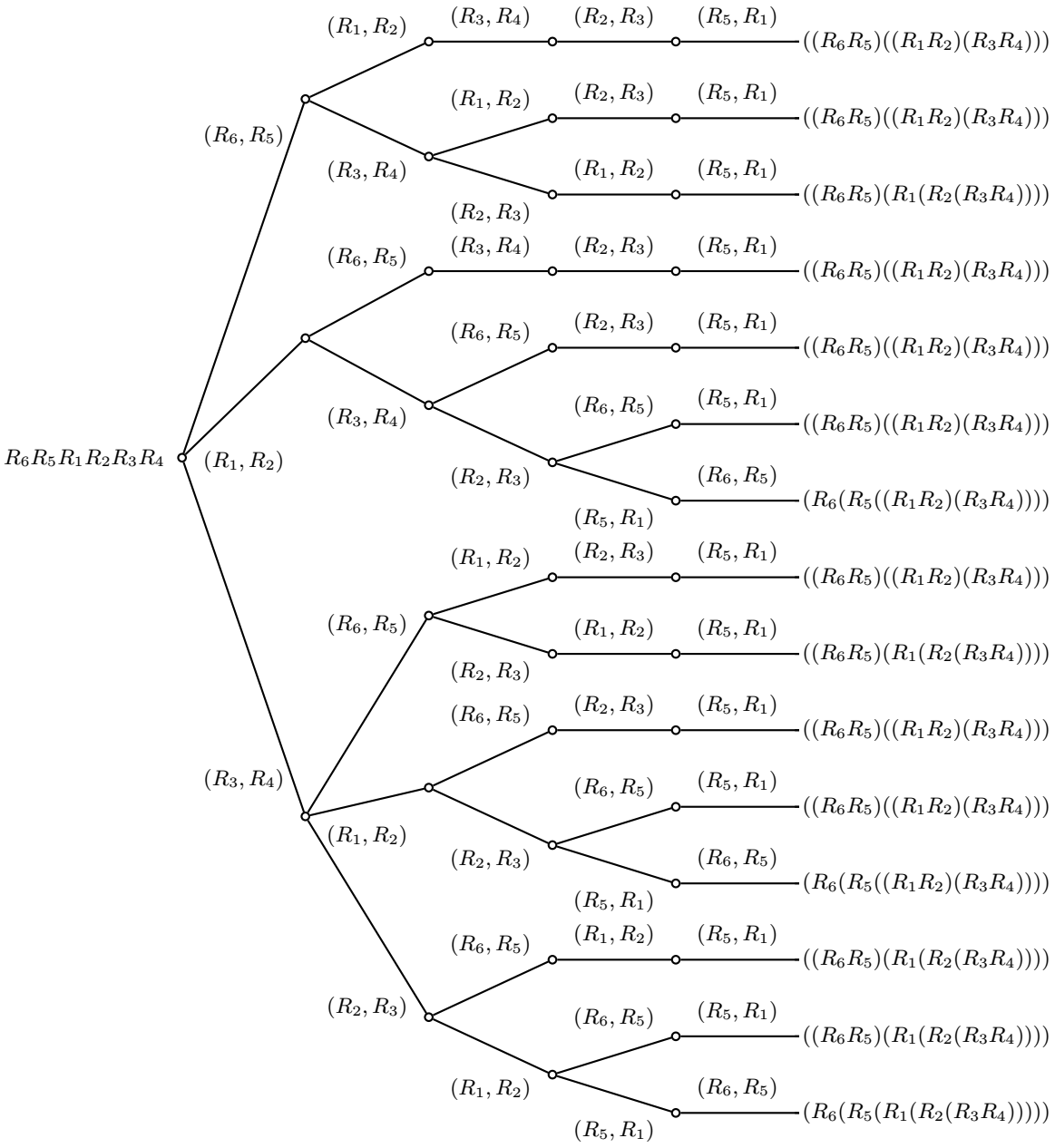


Figure 3.2: A computation tree.

Fact 3.1.2 *$cp(pref_k(\phi_1))$ uniquely corresponds to a partition of S into $n-k$ disjoint subsets.*

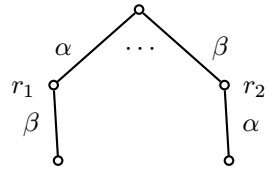
Fact 3.1.3 *If $cp(pref_k(\phi_1)) = cp(pref_k(\phi_2))$ then the subtrees rooted at the end of $pref_k(\phi_1)$ and $pref_k(\phi_2)$ are copies of each other.*

The last fact needs further explanation. Note that if $cp(pref_k(\phi_1)) = cp(pref_k(\phi_2))$ the partitions induced by $pref_k(\phi_1)$ and by $pref_k(\phi_2)$ must be identical (Fact 3.1.2). Also note that in the normalization process the possibility to group two compound relations x, y is not affected by the structure of inner tyings of x and y. Consequently, the subtrees rooted at the end of $pref_k(\phi_1)$ and $pref_k(\phi_2)$ must be copies of each other.

The next lemma says that whenever we have the choice between two different cp's in the normalization process, using one cp does not destroy the other cp.

Lemma 3.1.18 *Let T be a subtree of the computation tree of the procedure* `normalize` *for an optimal sequence of relations s. If there is an edge e_1 with label α and an edge e_2 with label β ($\beta \neq \alpha$) leaving the root of T, then there exists an edge with label β following e_1 and an edge with label α following e_2.*

Proof We denote the node at the end of edge e_1 with r_1 and the node at the end of edge e_2 with r_2, respectively. Let the root of T correspond to a sequence of compound relations $c_1 \dots c_k$. Let α correspond to the contradictory pair (c_i, c_{i+1}) $(1 \leq i < k)$ and β correspond to the contradictory pair (c_j, c_{j+1}) $(1 \leq j < k)$. Without loss of generality assume that $i < j$. We distinguish two cases.

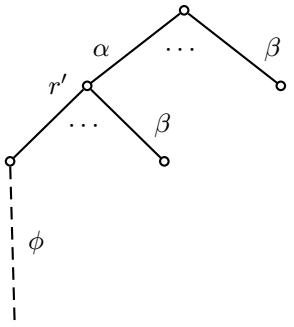


In the first case we have $i + 1 < j$, i.e. the contradictory pairs do not overlap. Obviously, tying c_i and c_{i+1} together does not destroy the contradictory pair (c_j, c_{j+1}). Since (c_j, c_{j+1}) still exists, there must be an edge leaving r_1 labeled with β.
In the second case we have $i + 1 = j$, i.e. the contradictory pairs overlap. According to Lemma 3.1.17, $(c_i c_{i+1}, c_{i+2})$ is a contradictory pair too. Note that $(c_i c_{i+1}, c_{i+2})$ corresponds to the cp β. Hence, after the grouping of c_i and c_{i+1} the cp β still exists. Consequently, there must be an edge leaving r_1 labeled with β. The claim that there is an edge labeled α leaving r_2 follows by symmetry. □

The next lemma says that once an edge with label β leaves a node r, then β occurs on every path starting at r.

Lemma 3.1.19 *Let T be a subtree of the computation tree of the procedure* `normalize` *for an optimal sequence of relations s. Then, if an edge leaving the root of T is labeled with β, β occurs on any path from the root of T to a leaf.*

Proof We perform induction on the height of the subtree T. If T has height 0 the claim is obviously true.
Now assume that the claim is true for all subtrees T with height$(T) \leq n$ for some $n \geq 0$. Consider a tree T of height $n + 1$. Let ϕ be an arbitrary path in T. If the first label of ϕ is β the claim is obviously true. Hence, assume that the first label of ϕ is different from β. Let us denote the second node in ϕ with r' and the subtree rooted at r' with T'. According to Lemma 3.1.18 there must be an edge leaving r' labeled with β. By an application of the induction hypothesis to the subtree T' we know that β occurs on every path in T'. Therefore β also occurs on ϕ. This proves the claim. □

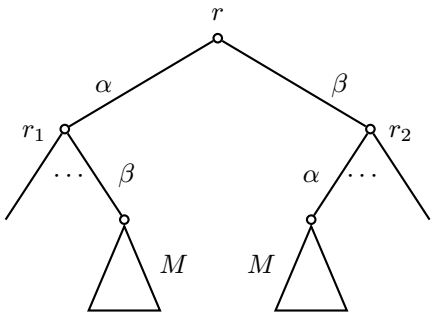


Corollary 3.1.3 *Let T be a subtree of the computation tree for the procedure* normalize *applied to an optimal sequence of relations s. Let r be the root node of T and Γ be the set of all cp's of edges to successor nodes of r. Then, for each path ϕ from r to a leaf of T, Γ is a subset of all the cp's in ϕ.*

Lemma 3.1.20 *Let T be a subtree of the computation tree for the procedure* normalize *applied to an optimal sequence of relations s. Then any two paths in T from the root to a leave are labeled with the same set of connecting points.*

Proof The proof is by induction on the height of the tree T. If height$(T) = 0$ the claim is trivially satisfied. We assume that the claim is true for all subtrees T with height$(T) \leq n$ for some $n \geq 0$.
Consider a subtree of height $n + 1$. If the root of T has only one successor the claim follows immediately from the induction assumption. Hence, let us assume that T has at least two successor nodes.

Consequently, there exist at least two paths from the root of T to leaves of T. We consider two such paths and denote them by ϕ_1 and ϕ_2, respectively. Furthermore, let α be the first cp of ϕ_1 and β the first cp of ϕ_2, respectively. Let r_1 be the node that is reached from r via α and r_2 be the node that is reached from r via β (see figure on the right). The corresponding subtrees are denoted by T_1 and T_2, respectively. Let r correspond to the sequence of contradictory chains $c_1, \ldots, c_k$. Further, let α correspond to the contradictory pair (c_i, c_{i+1}) and β correspond to the contradictory pair (c_j, c_{j+1}). Without loss of generality we assume that $1 \leq i < j < n$.



Due to Lemma 3.1.18 we know that there exists an edge labeled β leaving r_1 and an edge labeled α leaving r_2. Now, tying c_i and c_{i+1} together leaves the contradictory pair $(c_i c_{i+1}, c_{j+1})$ whereas tying c_j and c_{j+1} together leaves the contradictory pair $(c_i, c_j c_{j+1})$, i.e. $\alpha\beta$ and $\beta\alpha$ lead to the same sequence of contradictory subchains and the corresponding subtrees are copies of each other (Fact 3.1.3). Let us denote these two subtrees by T_1' and T_2', respectively. Applying the induction hypothesis to T_1' and T_2' we know that the set of cp's on all paths in T_1' or T_2' is the same, say M. Consequently, the set of cp's of ϕ_1 and ϕ_2 are both equal to $M \cup \{\alpha, \beta\}$ which proves the claim. □

The next lemma is an immediate consequence of Lemma 3.1.20.

Lemma 3.1.21 *Let T be a subtree of the computation tree for the procedure* normalize *applied to an optimal sequence of relations s. Then all paths in T from the root to a leave are labeled with the same set of connecting points.*

Proof Assume that the claim is wrong, i.e. there exist two paths whose sets of connecting points differ. However, this is a contradiction to Lemma 3.1.20. □

Theorem 3.1.1
The indeterministic normalization process is well-defined for chains consisting of optimal contradictory subchains, i.e. for every computation of the procedure normalize, the partition of the set of all n relations into maximal contradictory subchains is unique as long as all contradictory subchains in the decomposition are optimal.

Proof Assume the contrary is true, i.e. there exist two computations for an optimal sequence of relations that yield different partitions into contradictory subchains. Since these two computations correspond to paths in the computation tree with different sets of cp's we have a contradiction to Lemma 3.1.20. □

In [MS79], Monma and Sidney describe an elegant recursive algorithm to determine the optimal sequencing of n jobs with series-parallel precedence constraints. Ibaraki and Kameda [IK84] adapted this algorithm to tree queries without cross products; however, this is not possible here,

since we have an *unconstrained* problem where there is no precedence relation along which we could merge. However, the next two lemmata and the conjecture show a possible way to overcome this problem. The next lemma is a direct consequence of the normalization procedure.

Lemma 3.1.22 *Let $S = s_1 \dots s_m$ be an optimal chain consisting of the maximal contradictory subchains $s_1, \dots, s_m$ (as determined by the function* `normalize`*). Then*

$$rank(s_1) \leq rank_{s_1}(s_2) \leq rank_{s_1 s_2}(s_3) \leq \dots \leq rank_{s_1 \dots s_{m-1}}(s_m),$$

in other words, the (maximal) contradictory subchains in an optimal chain are always sorted by non-decreasing ranks.

Proof Assume that there exist two adjacent contradictory subchains s_i and s_{i+1} with $rank_{s_1 \dots s_{i-1}}(s_i) > rank_{s_1 \dots s_i}(s_{i+1})$. If the subchains are not connected we can interchange them, and according to Lemma 3.1.10 the resulting chain would have fewer costs, a contradiction! On the other hand, if they were connected they would have been tied together in the normalization process, again a contradiction. Hence, we conclude that such a pair of adjacent contradictory subchains with contradicting local ranks cannot exist. □

The next result shows how to build an optimal sequence from two unique rank-sorted non-interconnected sequences.

Lemma 3.1.23 *Let x and y be two unique rank-sorted sequences of maximal contradictory subchains for the disjoint and unconnected sets of relations R_x and R_y, respectively. Then the sequence obtained by merging the contradictory subchains in x and y (as obtained by* `normalize`*) according to their non-decreasing ranks is optimal.*

Proof Assume that the claim is wrong. Then there exists a sequence $c_1 \dots c_k$ of rank-sorted optimal contradictory subchains for the set of relations $R_x \cup R_y$. Since contradictory subchains are connected, each c_i refers either to relations in R_x or to relations in R_y. By eliminating the contradictory subchains in $c_1 \dots c_k$ that refer to R_y we would obtain a sequence of rank-sorted contradictory subchains for R_x, a contradiction to the assumed uniqueness of x. □

Merging two sequences in the way described in Lemma 3.1.23 is a fundamental process. We henceforth refer to it by simply saying that we *merge by the ranks*.

We strongly conjecture that the following generalization of the first part of Lemma 3.1.13 is true, although we could not yet prove it. It uses the notion of *optimal recursively decomposable subchains* defined in the next subsection.

Conjecture 3.1.1 *Consider two sequences S and T containing exactly the same relations R_1, ... ,R_n. Let $S = s_1 \dots s_k$ and $T = t_1 \dots t_l$ be decompositions of S and T into maximal contradictory subchains such that the subchains s_i $(i = 1, \dots, k)$ and s_i $(i = 1, \dots, l)$ are all optimal recursively decomposable with respect to the respective prefixes $s_1 \dots s_{i-1}$ and $t_1 \dots t_{j-1}$, respectively. Then S and T have equal costs.*

3.1.2 The First Algorithm

We first use a slightly modified cost function C' which additionally respects the size of the first relation in the sequence, i.e. C and C' relate via

$$C'_u(s) = \begin{cases} C(s) + |n_R|, & \text{if } u = \epsilon \text{ and } s = Rs' \\ C_u(s), & \text{otherwise} \end{cases}$$

This cost function can be treated in a more elegant way than C. The new rank function is now defined as

$$rank_u(s) := \frac{T_u(s) - 1}{C'_u(s)}.$$

Note that the rank function is now defined even if $u = \epsilon$ and s is a single relation. The size function remains unchanged. At the end of this subsection, we describe how our results can be adapted to the original cost function C.

The rank of a contradictory chain depends on the relative position of the relations that are directly connected to it. For example the rank of the contradictory subchain $(R_5R_3R_4R_2)$ depends on the position of the neighboring relations R_1 and R_6 relative to $(R_5R_3R_4R_2)$, that is whether they appear before or after the sequence $(R_5R_3R_4R_2)$. We therefore introduce the following fundamental definitions:

Definition 3.1.7 (neighborhood) *We call the set of relations that are directly connected to a subchain (with respect to the query graph G) the* complete neighborhood *of that subchain. A* neighborhood *is a subset of the complete neighborhood. The* complement *of a neighborhood u of a subchain s is defined as $v - u$, where v denotes the complete neighborhood of s.*

Note that the neighborhood of a subchain s within a larger chain us is uniquely determined by the subsequence u of relations preceding it. We henceforth denote a pair consisting of a connected sequence s and a neighborhood u by $[s]_u$.

Definition 3.1.8 (contradictory subchain, extent) *A* contradictory subchain $[s]_u$ *is inductively defined as follows.*

1. *For a single relation s, $[s]_u$ is a contradictory subchain.*
2. *There is a decomposition $s = vw$ such that (v, w) is a contradictory pair with respect to the preceding subsequence u and both $[v]_u$ and $[w]_{uv}$ are themselves contradictory subchains.*

The extent *of a contradictory chain $[s]_u$ is defined to be the pair consisting of the neighborhood u and the set of relations occurring in s. Since contradictory subchains are connected, the set of occurring relations has always the form $\{R_i, R_{i+1}, \ldots, R_{i+l}\}$ for some $1 \leq i \leq n,\ 0 \leq l \leq n - i$. An* optimal contradictory subchain *to a given extent is a contradictory subchain with lowest cost among all contradictory subchains of the same extent.*

Note that optimal contradictory subchains are only optimal with respect to the underlying recursive building scheme and hence they are not necessarily optimal when viewed as a sequence of relations.

Lemma 3.1.24 *The number of different extents of a chain of n relations is*

$$2n^2 - 2n + 1 = O(n^2).$$

Proof The set of possible extents of a chain $R_1 \ldots R_n$ can be written as

$$\begin{aligned}
&\{(\emptyset, \{R_1, \ldots, R_n\})\} &\uplus\\
&\{(u, \{R_1, \ldots, R_i\}) \mid 1 \leq i < n,\ u \in \{\emptyset, \{i+1\}\}\} &\uplus\\
&\{(u, \{R_i, \ldots, R_n\}) \mid 1 < i \leq n,\ u \in \{\emptyset, \{i-1\}\}\} &\uplus\\
&\{(u, \{R_i, \ldots, R_j\}) \mid 1 < i < j < n,\ u \in \{\emptyset, \{i-1\}, \{j+1\}, \{i-1, j+1\}\}\}.
\end{aligned}$$

Hence, the number of extents is

$$1 + 2 * (n-1) + 2 * (n-1) + 4 * \sum_{k=1}^{n-2} (n-k-1) \quad = \quad 2n^2 - 2n + 1.$$

□

Each contradictory chain can be completely recursively decomposed into adjacent pairs of connected subchains. Subchains with this property are defined next (similar types of decompositions occur in [HC93, SS91]).

Definition 3.1.9 ((optimal) recursively decomposable subchain) *A* recursively decomposable subchain $[s]_u$ *is inductively defined as follows.*

1. *If s is a single relation then $[s]_u$ is recursively decomposable.*
2. *There is a decomposition $s = vw$ such that v is connected to w and both $[v]_u$ and $[w]_{uv}$ are recursively decomposable subchains.*

A recursively decomposable subchain $[s]_u$ with extent (U, S) is called optimal recursively decomposable *if there is no other recursively decomposable chain $[t]_v$ with the same extent (U, S) and $C_v(t) < C_u(s)$.*

The extent of a recursively decomposable chain is defined in the same way as for contradictory chains. Note that every contradictory subchain is recursively decomposable. Consequently, the set of all contradictory subchains for a certain extent is a subset of all recursively decomposable subchains of the same extent.

Example 6: Consider the sequence of relations

$$s = R_2R_4R_3R_6R_5R_1.$$

Using parenthesis to indicate the recursive decompositions we have the following two possibilities

$$(((R_2(R_4R_3))(R_6R_5))R_1)$$

$$((R_2((R_4R_3)(R_6R_5)))R_1)$$

The extent of the recursively decomposable subchain $R_4R_3R_6R_5$ of s is $(\{R_2\}, \{R_3, R_4, R_5, R_6\})$. $R_3R_1R_4R_2$ is an example of a chain which is not recursively decomposable. □

The number of different recursively decomposable chains involving the relations $R_1, \ldots, R_n$ is the n-th Schröder number r_n [SS91]. It can be shown that $r_n \sim \frac{C(3+\sqrt{8})^n}{n^{3/2}}$ with $C = \frac{1}{2}\sqrt{\frac{3\sqrt{2}-4}{\pi}}$. By Stirling's Formula, $n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$ and therefore $\lim_{n\to\infty} \frac{r_n}{n!} = 0$, i.e. the probability of a random permutation being recursively decomposable strives to zero for large n.

There is an obvious dynamic programming algorithm to compute optimal recursively decomposable subchains. It is not hard to see that *Bellman's optimality principle* [Min86, CLR90] holds and every optimal recursively decomposable subchain can be decomposed into smaller optimal recursively decomposable subchains.

Example 7: In order to compute an optimal recursively decomposable subchain for the extent

$$(\{R_2, R_7\}, \{R_3, R_4, R_5, R_6\})$$

the algorithm makes use of optimal recursively decomposable subchains for the extents

$$\begin{array}{rl} (\{R_2\}, \{R_3\}) & (\{R_7, R_3\}, \{R_4, R_5, R_6\}) \\ (\{R_2\}, \{R_3, R_4\}) & (\{R_7, R_4\}, \{R_5, R_6\}) \\ (\{R_2\}, \{R_3, R_4, R_5\}) & (\{R_5, R_7\}, \{R_6\}) \\ (\{R_7\}, \{R_4, R_5, R_6\}) & (\{R_2, R_4\}, \{R_3\}) \\ (\{R_7\}, \{R_5, R_6) & (\{R_2, R_5\}, \{R_3, R_4\}) \\ (\{R_7\}, \{R_6\}) & (\{R_2, R_6\}, \{R_3, R_4, R_5\}) \end{array}$$

which have been computed in earlier steps. A similar dynamic programming algorithm can be used to determine optimal contradictory subchains. □

Let E be the set of all possible extents. We define the following partial order $\mathcal{P} = (E, \prec)$ on E. For all extents $e_1, e_2 \in E$, we have $e_1 \prec e_2$ if and only if e_1 can be obtained by splitting the extent e_2. For example, $(\{R_7\}, \{R_5, R_6\}) \prec (\{R_2, R_7\}, \{R_3, R_4, R_5, R_6\})$. The set of maximal extents M then corresponds to a set of incomparable elements (antichain) in $\mathcal{P}$ such that for all extents e enumerated so far, there is an extent $e' \in M$ with $e \prec e'$.

Now, since every optimal join sequence has a representation as a sequence of contradictory subchains, we only have to determine this representation. Consider a contradictory subchain c in an optimal join sequence s. What can we say about c? Obviously, c has to be optimal with respect to the neighborhood defined by the relations preceding c in s. Unfortunately, identifying contradictory subchains that are optimal sequences seems to be as hard as the whole problem of optimizing chain queries. Therefore we content ourselves with the following weaker condition which may lead to multiple representations. Nevertheless it seems to be the strongest condition for which all subchains satisfying the condition can be computed in polynomial time. The condition says that s should be optimal both with respect to all contradictory chains of the same extent as s and with respect to all recursively decomposable subchains of the same extent. So far it is not clear whether these conditions lead to multiple representations, therefore we have no choice but to enumerate all possible representations and select the one with minimal costs. If conjecture 3.1.1 holds, we know that multiple representations have identical costs, and we do not have to enumerate them all. Next we describe our first algorithm.

Algorithm I':

1. Use dynamic programming to determine all contradictory subchains. That is, for each possible extent keep track of the cheapest contradictory subchain if one exists. Also keep track of the set M of all maximal extents with respect to the partial order induced by splitting extents.
2. Use dynamic programming to determine all optimal recursively decomposable subchains for all extents included in some maximal extent in M.
3. Compare the results from steps 1 and 2 and retain only matching subchains.
4. Sort the contradictory subchains according to their ranks.
5. Eliminate contradictory subchains that cannot be part of a solution.
6. Use backtracking to enumerate all sequences of rank-ordered optimal contradictory subchains and keep track of the sequence with lowest cost.

In step 5 of the algorithm we eliminate contradictory subchains that do not contribute to a solution. Note that the contradictory subchains in an optimal sequence are characterized by the following two conditions.

1. The neighborhood of the first contradictory subchain in the sequence is minimal (empty set) whereas the neighborhood of the last contradictory subchain is maximal.
2. The extents of all contradictory subchains in the representation build a partition of the set of all relations.
3. The neighborhoods of all contradictory subchains are consistent with the relations occurring at earlier and later positions in the sequence.

Note that any contradictory subchain occurring in the optimal sequence (except at the first and last positions) necessarily has matching contradictory subchains preceding and succeeding it in the list. In fact, every contradictory subchain X with neighborhood P occurring in the optimal join sequence must meet the following two conditions.

1. For every relation R in the neighborhood of X, there exists a contradictory subchain Y with neighborhood P' at an earlier position in the list (i.e. with smaller rank) which itself fulfills condition 1, such that R occurs in Y, and Y can be followed by X (i.e. $X \cap Y = \emptyset$ and $X \cap P' = \emptyset$).
2. For every relation R in the complementary neighborhood of X, there exists a contradictory subchain Y at a later position in the list (i.e. with larger rank) which itself fulfills condition 2, such that R occurs in the neighborhood of Y, and X can be followed by Y (i.e. $X \cap Y = \emptyset$ and $Y \cap P = \emptyset$).

Using these two conditions, we can eliminate "useless" contradictory chains from the rank-ordered list by performing a reachability[4] algorithm for each of the DAGs defined by the conditions 1 and 2. In the last step of our algorithm backtracking is used to enumerate all representations. Example 8 shows the subchains examined by Algorithm I' at various intermediate stages.

Lemma 3.1.25 *The first algorithm is correct.*

Correctness: First, note that the algorithm systematically enumerates all contradictory subchains. Contradictory subchains that turn out to be suboptimal are eliminated. From the resulting set of contradictory subchains, all sequences of rank-ordered contradictory subchains are constructed and the cheapest one is selected. Since an optimal chain can be proved to be a rank-ordered sequence of (optimal) contradictory subchains (Lemma 3.1.22) the optimal chain has to be among the enumerated rank-ordered sequences of contradictory subchains.

Complexity: Let us analyze the worst case time complexity of the algorithm. The two dynamic programming steps both iterate over $O(n^2)$ different extents (Lemma 3.1.24) and each extent gives rise to $O(n)$ splittings. Moreover, for each extent one normalization is necessary, which requires linear time (cost, size and rank can be computed in constant time using recurrences). Therefore the complexity of the two dynamic programming steps is $O(n^4)$. Sorting $O(n^2)$ contradictory chains can be done in time $O(n^2 \log n)$. The step where all "useless" contradictory subchains are eliminated consists of two stages of a reachability algorithm which has complexity $O(n^4)$. If conjecture 3.1.1 is true, the backtracking step requires linear time and the total complexity of the algorithm is $O(n^4)$. Otherwise, if conjecture 3.1.1 is false, the algorithm might exhibit exponential worst case time complexity as the following example indeed shows.

[4]Basically, we compute the transitive closure of a relation $\rightarrow$ and its inverse relation $\leftarrow$. Only those nodes that are reachable from a starting node and from which we can reach a final node are retained. A similar algorithm is used to identify all "useful" symbols in a context-free grammar [HU79].

Consider the following hypothetical[5] set of contradictory subchains:

$$\begin{aligned} X_n &= \bigcup_{i=1}^{n} B_i, \text{ where} \\ B_i &= \bigcup_{j=0}^{i-1} C_{j,i} \\ C_{0,i} &= \{\emptyset, \{R_0, R_1, \ldots, R_i\} \\ C_{j,i} &= \{\{R_{j-1}\}, \{R_j, R_{j+1}, \ldots, R_i\}\} \quad 0 < j \leq i \end{aligned}$$

Assume that the rank ordering is as follows

$$rank(C_{i,j}) \leq rank(C_{k,l}) \iff i < k \text{ or } (i = k \text{ and } j < l) \tag{3.4}$$

The next table shows the set X_n for the case of $n = 6$ in form of a matrix. Rows represent the relations $R_0, \ldots, R_5$ and columns represent the contradictory subchains of X_n in rank-sorted order (ranks increase from left to right). A cross denotes the presence of a relation in a contradictory subchain and a dot denotes relations in the neighborhood.

R_0	×	×	·	×	·		×	·			×	·				×	·				
R_1		×	×	×	×	·	×	×	·		×	×	·			×	×	·			
R_2				×	×	×	×	×	×	·	×	×	×	·		×	×	×	·		
R_3							×	×	×	×	×	×	×	×	·	×	×	×	×	·	
R_4											×	×	×	×	×	×	×	×	×	×	·
R_5																×	×	×	×	×	×

Any possible solution corresponds to a sub-matrix with some columns canceled out such that the following conditions hold. In every row there is exactly one cross. There is a dot at each position P directly above or directly below a block of adjacent crosses in a column if and only if there is a cross to the left of P in the same row.
Now let us compute the number of possible solutions a_n that arise from X_n. By the i-th block of the matrix we mean the sub-matrix which corresponds to the set B_i. First, note that R_{n-1} can only be covered by a contradictory subchain from the last block of the matrix. Second, any solution includes at most one subchain from every block. Third, if the last subchain in a solution is the chain that corresponds to the i-th column ($1 \leq i \leq n$) in block n (the last block), there are a_{i-1} ways to cover the remaining relations $R_0, R_1, \ldots, R_{i-1}$. Hence, a_n satisfies the recurrence

$$a_n = \begin{cases} 1, & n = 0 \\ \sum_{i=0}^{n-1} a_i, & n > 0 \end{cases}$$

Subtracting a_{n+1} from a_n we obtain the simpler recurrence $a_n = 2a_{n-1}$ for $n > 1$ and $a_1 = 1$. Obviously, the solution is $a_n = 2^{n-1}, n \geq 1$.

The following pseudocode is a more detailed description of Algorithm I'. Besides basic types like booleans, integers and floats we make use of arrays, sets and lists. Note that lists are denoted as in Lisp with ordinary brackets (no square brackets). The standard list operations `head()`, `tail()`, `cons()`, `append()` and `reverse()` have the usual semantics. Sometimes we treat lists as sets and apply set operators like $\subseteq$. The base relations $R_0, \ldots, R_{n-1}$ are represented by the integers $0, \ldots, n-1$.

[5]This is only a simple combinatorial argument which ignores properties of the cost function. It is far from obvious that such a set of contradictory subchains can really exist.

```
1 procedure all-contradictory-subchains(F, N, n);
2     ▷ input: n: number of relations in the chain,
3              N[i]: size of relation R_i,
4              F[i, j]: selectivity between relations R_i and R_j
5     ▷ output: a list of all possible optimal recursively decomposable contr. subchains
6               (each element of the list is a pair (p, c), consisting of a neighborhood p
7               and a subchain c)
8         // tables used:
9         // A[i, j, k]: optimal subchain for extent (k, {i, ... i + j − 1})
10        // T[i, j, k]: size of subchain A[i, j, k]
11        // C[i, j, k]: cost of subchain A[i, j, k]
12        // k codes a neighborhood of {i, ... i + j − 1} as follows:
13        // (k & 1) = 1 ⇔ neighborhood contains {i − 1}
14        // (k & 2) = 1 ⇔ neighborhood contains {i + j}
15    for 0 ≤ i < n, 0 ≤ k ≤ 4 do
16        nb ← ∅;   // decode neighborhood
17        if (k & 1) and (i > 0) then nb ← nb ∪ {i − 1};
18        if (k & 2) and (i + j < n − 1) then nb ← nb ∪ {i + j};
19        A[i, i, k] ← (i);
20        T[i, i, k] ← N[i] ∗ ∏_{j∈nb} F[j, i];
21        C[i, i, k] ← T[i, i, k]
22        A1[i, i, k] ← A[i, i, k];
23        C1[i, i, k] ← C[i, i, k];
24        T1[i, i, k] ← T[i, i, k];
25    od;
26    peaks ← nil;
27    for 2 ≤ j ≤ n, 0 ≤ i ≤ n − j, 0 ≤ l < 4 do
28        if i > 0 then w1 ← 1
29                 else w1 ← 0;
30        if i + j < n − 1 then w2 ← 1
31                         else w2 ← 0;
32        for k1 ← 0 to w1 do
33            for k2 ← 0 to w2 do
34                l ← k1 + 2 ∗ k2;   // coded neighborhood
35                left0 ← i;
36                right1 ← i + j − 1;
37                for k ← 1 to l − j do
38                    left1 ← i + k − 1;
39                    right0 ← i + k;
40                    l1 ← l & 1;
41                    l2 ← (l & 2) | 1;
42                    if A[left0, left1, l1] ≠ nil and A[right0, right1, l2] ≠ nil then
43                        subchain ← append(A[left0, left1, l1], A[right0, right1, l2]);
44                        if subchain is a contr. subchain then
45                            size ← T[left0, left1, l1] ∗ T[right0, right1, l2];
46                            cost ← C[left0, left1, l1] + T[left0, left1, l1] ∗ C[right0, right1, l2];
47                            if cost < C[left0, right1, l] or A[left0, right1, l] = nil then
48                                A[left0, right1, l] ← subchain;
49                                C[left0, right1, l] ← cost;
50                                T[left0, right1, l] ← size;
```

```
51                              fi;
52                          fi;
53                          l1 ← l & 2;
54                          l2 ← (l & 1) | 2;
55                          subchain ← append(A[right0, right1, l1], A[left0, left1, l2]);
56                          if subchain is a contr. subchain then
57                              size ← T[right0, right1, l1] * T[left0, left1, l2];
58                              cost ← C[right0, right1, l1] + T[right0, right1, l1] * C[left0, left1, l2];
59                              if cost < C[left0, right1, l] then
60                                  A[left0, right1, l] ← subchain;
61                                  C[left0, right1, l] ← cost;
62                                  T[left0, right1, l] ← size;
63                              fi;
64                          fi;
65                      fi;
66                  fi;
67              od;
68              remove all tuples (l, r) with left0 ≤ l and r ≤ right1 from the list peaks;
69              peaks ← cons((left0, right1), peaks);
70          od;
71          ccl ← nil;
72          for (l, r) in peaks do
73              for 2 ≤ j ≤ r − l + 1, l ≤ i ≤ r − j + 1 and
74                  l a coding of a neighborhood of the block {i, ..., i + j − 1} do
75                  left0 ← i;
76                  right1 ← i + j − 1;
77                  let pref be the neighborhood coded by l;
78                  if A1[left0, right1, l] ≠ nil then
79                      for k ← 1 to l − j do
80                          left1 ← i + k − 1;
81                          right0 ← i + k;
82                          l1 ← l & 1;
83                          l2 ← (l & 2) | 1;
84                          subchain ← append(A1[left0, left1, l1], A1[right0, right1, l2]);
85                          size ← T1[left0, left1, l1] * T1[right0, right1, l2];
86                          cost ← C1[left0, left1, l1] + T1[left0, left1, l1] * C1[right0, right1, l2];
87                          if cost < C1[left0, right1, l] or A1[left0, right1, l] = nil then
88                              A1[left0, right1, l] ← subchain;
89                              T1[left0, right1, l] ← size;
90                              C1[left0, right1, l] ← cost;
91                          fi;
92                          l1 ← l & 2;
93                          l2 ← (l & 1) | 2;
94                          subchain ← append(A1[right0, right1, l1], A1[left0, left1, l2]);
95                          size ← T1[right0, right1, l1] * T1[left0, left1, l2];
96                          cost ← C1[right0, right1, l1] + T1[right0, right1, l1] * C1[left0, left1, l2];
97                          if cost < C1[left0, right1, l] then
98                              A1[left0, right1, l] ← subchain;
99                              T1[left0, right1, l] ← size;
100                             C1[left0, right1, l] ← cost;
101                         fi;
102                     od;
103                     if A[left0, right1, l] ≠ A1[left0, right1, l] then
104                         A[left0, right1, l] ← nil;
```

```
105                     else
106                         if A[left0, right1, l] is contr. subchain then
107                             ccl ← cons(A[left0, right1, l], ccl);
108                         fi;
109                     fi;
110                 od;
111             od;
112         od;
113     return ccl;
114 end
```

```
1   procedure sort-by-rank(chains);
2       ▷ input: a list of subchains; subchains are represented by tuples (p, c)
3                where c is a subsequence and p a suitable neighborhood to c
4       ▷ output: input list chains sorted by non-descending local ranks
5                 (the local rank of a subchain (p, c) is rank_p(c))
6       ...
7   end
```

```
1   procedure group-by-equal-ranks(chains);
2       ▷ input: a list of rank-ordered subchains
3       ▷ output: input list parenthesized (grouped) by chains with equal local ranks
4       ...
5   end
```

```
1   procedure toposort-groups(gcc);
2       ▷ input: grouped list of rank-ordered subchains
3       ▷ output: input list with each of its groups topologically sorted
4                 with respect to precedences implied by the neighborhoods and carrier
5                 sets of the subchains
6       result ← nil;
7       for each grp in gcc do
8           res ← nil;
9           subchains_left ← grp;
10          while subchains_left ≠ nil do
11              grp ← subchains_left;
12              subchains_left ← nil;
13              for (p, c) in grp do
14                  let right be the part of the list grp to the right of
15                    the element (p, c);
16                  if there is a chain (p′, c′) in one of the lists subchains_left or right
17                      such that either p and c′ intersect or p′ and c intersect
18                      then add (p, c) to the end of list subchains_left;
19                      else add (p, c) to the end of list res;
20                  fi;
21              od;
22          od;
23          add res to the end of list result;
24      od;
25      return result;
26  end
```

```
1   procedure reduce(chains);
2       ▷ input: rank-ordered list of subchains
3       ▷ output: input list with irrelevant subchains eliminated
3
4       // remove contradictory subchains from the start (end) of the list
5       // which cannot be the first (last) contr. suchains in any solution
6       delete all elements (p, c) with p = nil from the beginning of the list chains
6       (start at the head of the list and stop at the first element (p, c) with p ≠ nil);
7       delete all elements (p, c) with
8         (carrier(c) = {i, i + 1, ..., j − 1, j} and 0 < i ≤ j < n and length(p) < 2) or
9         (carrier(c) = {i, i + 1, ..., j − 1, j} and i ≤ j and (i = 0 or j = n − 1) and p = nil)
10      from the end of the list chains (start at the end of the
10      list and stop when the condition is false the first time);
11      // remove contr. subchains that cannot be preceded by any subchain
12      new_list ← nil;
13      for each (p, c) in chains do
14          if new_list ≠ nil then
15              ok ← true;
16              for each k in p do
17                  if (k occurs in c′ and neither c′ nor p′ overlaps with c)
18                      for some element (p′, c′) in new_list
19                      then ok ← false;
20                  fi;
21              od;
22          fi;
23          if ok then add (p, c) to the end of new_list;
24      od;
25      // remove contr. subchains that cannot be followed by any subchain
26      new_list ← nil;
27      for each (p, c) in reverse(chains) do
28          if new_list ≠ nil then
29              ok ← true;
30              let p1 be the complementary neighborhood of p (w.r.t. the
31                maximal possible neighborhood);
32              for each k in p1 do
33                  if (k occurs in c′ and c does not overlap with c′ wheras
34                      p′ overlaps with c for some element (p′, c′) in new_list)
35                      then ok ← false;
36                  fi;
37              od;
38          fi;
39          if ok then add (p, c) to the end of new_list;
40      od;
41      return chains;
42  end

1   procedure reachable(prefix, chains);
2       ▷ input: a rank-ordered list of subchains and a prefix
3       ▷ output: a bit vector of length n indicating which relations can be covered
4                 by subsequent contr. subchains from the list chains and which
5                 relations can certainly not be covered
6       reach ← nil;
7       for each (p, c) in chains do
8           compl ← nil;
```

```
9             let {left, ..., right} be the carrier set of the subchain c;
10            if (left ≠ 0) and (left − 1 ∉ p) then
11                  compl ← cons(left − 1, compl);
12            fi;
13            if (right ≠ n − 1) and (right + 1 ∉ p) then
14                  compl ← cons(right + 1, compl);
15            fi;
16            if (prefix ∩ c = ∅) and (prefix ∩ compl = ∅) and (p ⊆ prefix)
16                  and (c is not contained in reach) then
18                  reach ← cons(c, reach);
20            fi;
21      od;
22      return reach;
23  end
```

```
1   procedure find-next-contr-subchain(prefix, chains);
2       ▷ input: a rank-ordered list of subchains and a prefix
3       ▷ output: the first contr. subchain from the list chains that is compatible with the
4                 prefix and where all non-covered relations are reachable
7       rest ← (0, ..., n − 1) − pf;
8       ok ← false;
9       while (chains ≠ nil) and ¬ok do
10            (p, c) ← head(chains);
11            chains ← tail(chains);
13            let {left, ..., right} be the carrier set of c;
14            rest1 ← (rest − c) ∪ (c − rest);
15            ok ← ok and (p ⊆ pf) and (pf ∩ c = ∅) and
16                  ((left = 0) or (left − 1 ∈ p) or (left − 1 ∉ pref)) and
17                  ((right = n − 1) or (right + 1 ∈ p) or (right + 1 ∉ pref)) and
18                  (rest1 ⊆ reachable(prefix, chains));
19      od;
20      if ok then return (p, c);
21          else return nil fi;
22  end
```

```
1   procedure find-first-solution(prefix, chains);
2       ▷ input: a rank-ordered list of subchains and a prefix
3       ▷ output: the first possible sequence of contr. subchains from the list chains that
4                 is compatible with the given prefix and which covers all missing relations.
5       if length(prefix) = n then return prefix;
6         else
7           (p, c) ← find-next-contr-subchain(prefix, chains);
8           return find-first-solution(append(prefix, c), tail(chains));
9       fi;
10  end
```

```
1   procedure optimize(F, N, n);
2       ▷ input: n: number of relations in the chain,
3                N[i]: size of relation R_i (0 ≤ i < n)
4                F[i, j]: selectivity between relations R_i and R_j (0 ≤ i, j < n)
5       ▷ output: a permutation of (0, 1, ..., n − 1) representing the order of relations
6                 in an optimal left-deep processing tree for the join query R_0 ⋈ ··· ⋈ R_{n−1}
```

```
7                 where cross products are allowed.
8       return reduce(sort-by-local-ranks(all-contradictory-subchains(F, N, n)));
9   end
```

We now describe how to reduce the problem for our original cost function C to the problem for the modified cost function C'. One difficulty with the original cost function is that the ranks are defined only for subsequences of *at least two* relations. Hence, for determining the first relation in our solution we do not have sufficient information. An obvious solution to this problem is to try every relation as starting relation, process each of the two resulting chain queries separately and choose the chain with minimum costs. The new complexity will increase by about a factor of n. This first approach is not very efficient, since the dynamic programming computations overlap considerably, e.g. if we perform dynamic programming on the two overlapping chains $R_1R_2R_3R_4R_5R_6$ and $R_2R_3R_4R_5R_6R_7$, for the intersecting chain $R_2R_3R_4R_5R_6$ everything is computed twice. The cue is that we can perform the dynamic programming calculations before we consider a particular starting relation. Hence, the final algorithm can be sketched as follows:

Algorithm I:

1 Compute all contradictory chains by dynamic programming (corresponds to the steps 1-4 of Algorithm I')

2 For each starting relation R_i do the following steps:

- 2.1 Let L_1 be the result of applying steps 5 and 6 of Algorithm I' to all contradictory subchains whose extent (N, M) satisfies $R_i \in N$ and $M \subseteq \{R_1, \ldots, R_i\}$.
- 2.2 Let L_2 be the result of applying steps 5 and 6 of Algorithm I' to all contradictory subchains whose extent (N, M) satisfies $R_i \in N$ and $M \subseteq \{R_i, \ldots, R_n\}$.
- 2.3 For all $(l_1, l_2) \in L_1 \times L_2$ do the following steps:
  - 2.3.1 Let L be the result of merging l_1 and l_2 according to their ranks.
  - 2.3.2 Use R_iL to update the current-best join ordering.

Complexity: Suppose conjecture 3.1.1 is true and we can replace the backtracking part by a search for the first solution. Then the complexity of the step 1 is $O(n^4)$ whereas the complexity of step 2 amounts to $\sum_{i=1}^{n}(O(i^2) + O(n-i)^2 + O(n)) = O(n^3)$. Hence the total complexity would be $O(n^4)$ in the worst case. Of course, if our conjecture is false, the necessary backtracking step might lead to an exponential worst case complexity.

The following pseudocode is a more detailed description of Algorithm I.

```
1  proc optimize();
2       initialize minchain to an arbitrary chain;
3       acc ← all-contradictory-subchains(F, N, n);
4       for i  from 0 to n − 1 do
```

```
 8            lcc ← select-all-ccs(0, i − 1, acc);
 9            rcc ← select-all-ccs(i + 1, n − 1, acc);
10            x ← find-first-chain(reduce(r, sort-by-local-ranks(r, lcc)));
11            y ← find-first-chain(reduce(r, sort-by-local-ranks(r, rcc)));
12            z ← cons(r, convert(merge-by-local-ranks(r, x, y)));
13            use z to update minchain;
14        od
15        return minchain;
16    end
```

```
 1  proc select-all-ccs(l, r, chains);
 2        result ← nil;
 3        if l ≤ r then
 4            for (p, c) in chains do
 5                if (p ∪ c) ⊆ {l, l + 1, ..., r − 1, r} then result ← cons((p, c), result);
 6            od
 7        fi
 8        return result;
 9    end
```

The functions `all-contradictory-subchains()`, `sort-by-local-ranks()`, `reduce()`, `find-first-chain()` and `convert()` have already been described before. The additional parameter R accounts for the starting relation.

Example 8: Consider a chain query with the following parameters:

n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}	n_{11}
820	930	870	160	600	880	760	530	800	990	200	980

$f_{0,1}$	$f_{1,2}$	$f_{2,3}$	$f_{3,4}$	$f_{4,5}$	$f_{5,6}$	$f_{6,7}$	$f_{7,8}$	$f_{8,9}$	$f_{9,10}$	$f_{10,11}$
0.20	0.37	0.18	0.32	0.42	0.10	0.61	0.87	0.43	0.44	0.44

The list of all optimal contradictory subchains is shown below. The ranks are ascending with the topmost subchain having the lowest rank. Each line consists of a neighborhood and a corresponding subchain, e.g. in the first line $\{2, 4\}$ (3) represents the subchain R_3 with neighborhood $\{R_2, R_4\}$.

```
 {2,4} (3)
   {2} (3)
 {4,6} (5)
{9,11} (10)
 {5,7} (6)
 {4,7} (5,6)
   {4} (3)
 {1,3} (2)
   {1} (3,2)
 {0,2} (1)
 {0,3} (2,1)
   {0} (3,2,1)
   {5} (6)
```

```
   {4} (5,6)
 {3,5} (4)
 {3,6} (5,4)
 {3,7} (6,5,4)
   {3} (6,5,4)
   {5} (3,4)
   {6} (5,3,4)
   {7} (6,5,3,4)
    {} (6,5,3,4)
   {6} (5)
   {9} (10)
  {11} (10)
   {7} (6,5)
    {} (6,5)
   {3} (2)
    {} (3,2)
    {} (3)
   {1} (0)
   {2} (1,0)
   {0} (1)
    {} (0,1)
{8,10} (9)
   {8} (10,9)
   {3} (4)
    {} (10)
   {5} (4)
 {6,8} (7)
 {7,9} (8)
 {6,9} (7,8)
{7,10} (9,8)
{6,10} (7,9,8)
   {1} (2)
   {6} (7)
   {9} (8)
   {2} (1)
  {10} (9,8)
    {} (2,1)
   {4} (5)
    {} (4,5)
   {8} (9)
   {7} (8,9)
    {} (8,9)
  {10} (11)
  {10} (9)
   {8} (7)
   {7} (6)
    {} (7,6)
    {} (7)
    {} (4)
   {7} (8)
    {} (6)
    {} (8)
    {} (0)
    {} (2)
```

```
    {} (5)
    {} (1)
    {} (11)
    {} (9)
```

These 71 subchains are the input for the procedure `reduce` which eliminates subchains that can not be part of any solution. In the first part of the reduction process (lines 2-10 of reduce) all subchains from the beginning (end) of the list whose neighborhoods are not minimal (maximal) are removed. In our example this halves the number of subchains, leaving 35 subchains:

```
     {} (6,5,3,4)
    {6} (5)
    {9} (10)
   {11} (10)
    {7} (6,5)
     {} (6,5)
    {3} (2)
     {} (3,2)
     {} (3)
    {1} (0)
    {2} (1,0)
    {0} (1)
     {} (0,1)
 {8,10} (9)
    {8} (10,9)
    {3} (4)
     {} (10)
    {5} (4)
  {6,8} (7)
  {7,9} (8)
  {6,9} (7,8)
 {7,10} (9,8)
 {6,10} (7,9,8)
    {1} (2)
    {6} (7)
    {9} (8)
    {2} (1)
   {10} (9,8)
     {} (2,1)
    {4} (5)
     {} (4,5)
    {8} (9)
    {7} (8,9)
     {} (8,9)
   {10} (11)
```

After the second part of the reduction process only 6 subchains are left.

```
     {} (6,5,3,4)
    {3} (2)
    {2} (1,0)
     {} (10)
 {6,10} (7,9,8)
   {10} (11)
```

Hence we have the following sequence of rank-sorted optimal contradictory subchains:

$$(6\ 5\ 3\ 4)\,(2)\,(1\ 0)\,(10)\,(7\ 9\ 8)\,(11)$$

The corresponding left-deep plan is:

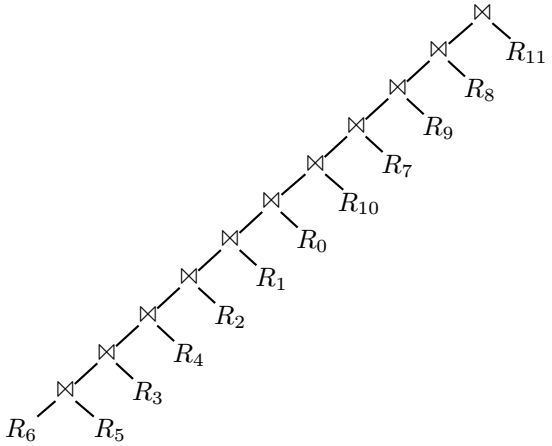


Figure 3.3 shows the previous three tables in graphical form. The representation has been chosen to illustrate the search process for a solution. Each subchain in a list is represented by a row in the table. Crosses represent relations in a subchain whereas dots represent relations in a neighborhood of a subchain, i.e. a dot means that there should be a cross in the same column above. Any solution may be build by successively picking subchains from the table proceeding from top to bottom while avoiding duplicate relations and neighborhoods that do not match with previously picked relations.

3.1.3 The Second Algorithm

The second algorithm is much simpler than the first one but proves to be less efficient in practice. Since the new algorithm is very similar to parts of the old one, we just point out the differences between both algorithms. The new version of the algorithm works as follows.

Algorithm II':

1. Use dynamic programming to compute an optimal recursively decomposable chain for the whole set of relations $\{R_1, \ldots, R_n\}$.
2. Normalize the resulting chain.
3. Reorder the contradictory subchains according to their ranks.
4. De-normalize the sequence.

Step 1 is identical to step 2 of our first algorithm. Note that Lemma 3.1.22 cannot be applied to the sequence in step 2 since an optimal recursively decomposable chain is not necessarily an optimal chain. Therefore the question arises whether step 3 really makes sense. One can show that the partial order defined by the precedence relation among the contradictory subchains has the property that all elements along paths in the partial order are sorted by rank. By computing a greedy topological ordering (greedy with respect to the ranks) we obtain a sequence as requested in step 3.

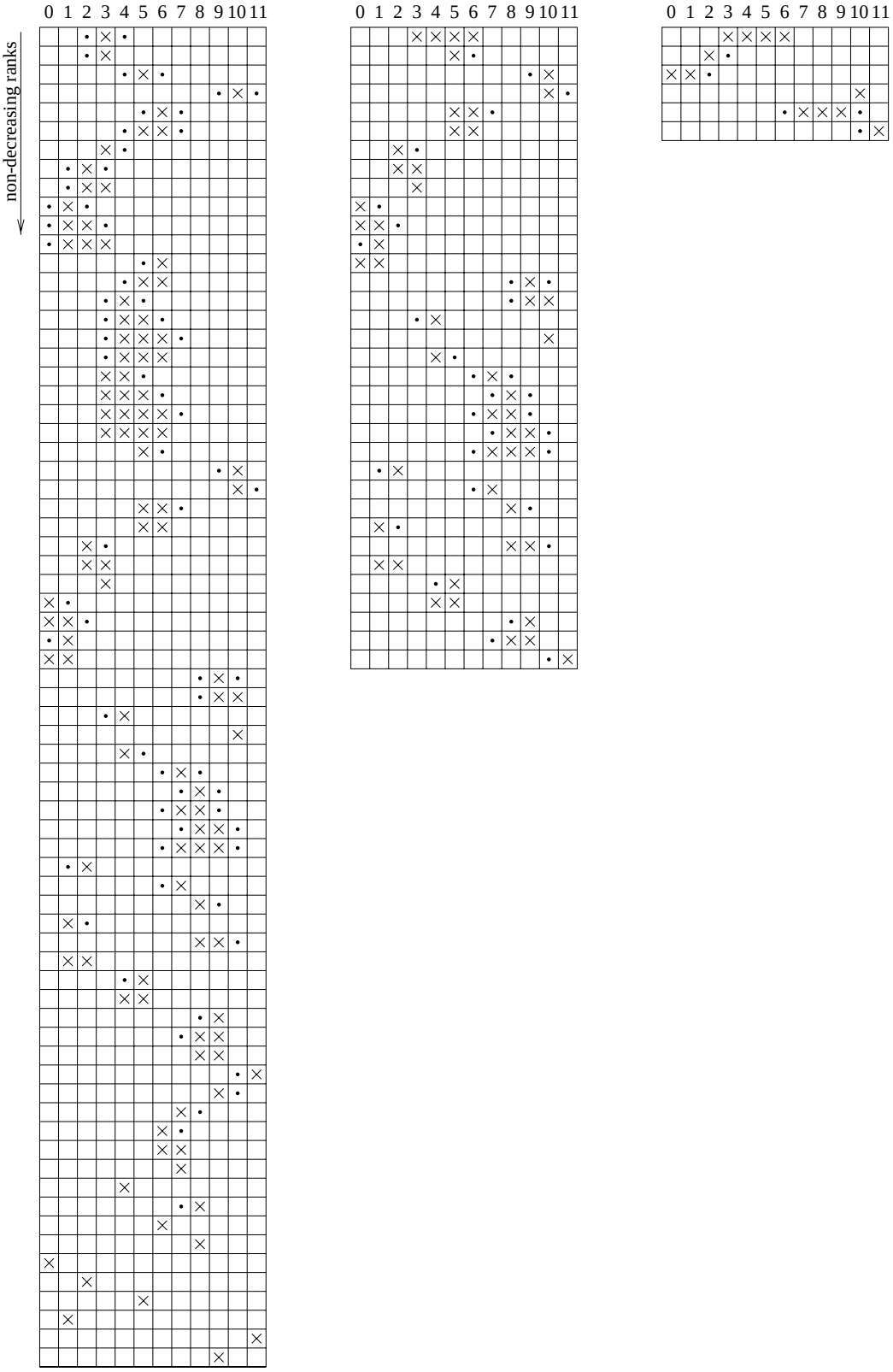


Figure 3.3: Subchains at various stages of Algorithm I

The pseudocode for Algorithm II' reads as follows:

```
1   procedure optimal-rec-decomp-chains(F, N, n);
2       ▷ input: n: number of relations in the chain,
3                N[i]: size of relation R_i,
4                F[i,j]: selectivity between relations R_i and R_j
5       ▷ output: an optimal recursively decomposable subchain
6       // tables used:
6       // A[i,j,k]: optimal subchain for extent (k, {i, ... i+j-1})
6       // T[i,j,k]: size of subchain A[i,j,k]
6       // C[i,j,k]: cost of subchain A[i,j,k]
6       // k codes a neighborhood of {i, ... i+j-1} as follows:
6       // (k & 1) = 1 ⇔ neighborhood contains {i-1}
6       // (k & 2) = 1 ⇔ neighborhood contains {i+j}
6       for 1 ≤ i < n-1, 0 ≤ k < 4 do
6           nb = ∅;   // decode neighborhood
6           if (k & 1) and (i > 0) then nb ← nb ∪ {i-1};
6           if (k & 2) and (i+j < n-1) then nb ← nb ∪ {i+j};
8           A[i,i,k] ← (i);
9           T[i,i,k] ← N[i] * ∏_{j∈nb} F[j,i];
10          C[i,i,k] ← T[i,i,k];
11      od;
12      for 2 ≤ j ≤ n, 0 ≤ i ≤ n-j do
12      if i > 0 then w1 ← 1
12               else w1 ← 0;
12      if i+j < n-1 then w2 ← 1
12                   else w2 ← 0;
12      for k1 ← 0 to w1 do
12          for k2 ← 0 to w2 do
12              l ← k1 + 2*k2;   // coded neighborhood
14              left0 ← i;
15              right1 ← i+j-1;
15              T[left0, right1, l] ← nil;
17              for k ← 1 to l-j do
18                  left1 ← i+k-1;
19                  right0 ← i+k;
20                  l1 ← l & 1;
21                  l2 ← (l & 2) | 1;
22                  l3 ← l & 2;
23                  l4 ← (l & 1) | 2;
24                  newchain1 ← append(A[left0, left1, l1], A[right0, right1, l2]);
25                  newchain2 ← append(A[right0, right1, l3], A[left0, left1, l4]);
26                  if T[left0, right1, l] = nil then
27                      T[left0, right1, l] ← T[left0, left1, l1] * T[right0, right1, l2];
28                  fi;
29                  cost1 ← C[left0, left1, l1] + T[left0, left1, l1] * C[right0, right1, l2];
30                  cost2 ← C[right0, right1, l3] + T[right0, right1, l3] * C[left0, left1, l4];
31                  if cost1 < C[left0, right1, l] then
32                      A[left0, right1, l] ← newchain1;
```

```
33                          C[left0, right1, l] ← cost1;
34                      fi;
35                      if cost2 < C[left0, right1, l] then
36                          A[left0, right1, l] ← newchain2;
37                          C[left0, right1, l] ← cost2;
39                      od;
39                  od;
40              od;
41          return A[0, n − 1, 0];
42      end
```

```
1       procedure contr-subchains(ch);
2           ▷ input: a subchain ch
3           ▷ output: a list of contradictory subchains (with neighborhood)
4                     corresponding to the normalization of ch
5           ...
6       end
```

```
1       procedure join(cc − list);
2           ▷ input: a list of disjoint contradictory subchains (with neighborhood)
3           ▷ output: the corresponding subchain (a flat list)
4           ...
5       end
```

```
1       procedure sort-by-rank(chains);
2           ▷ input: a list of subchains; subchains are represented by tuples (p, c)
3                    where c is a subsequence and p a suitable neighborhood to c
4           ▷ output: the input list chains sorted by non-descending local ranks
5                     (the local rank of a subchain (p, c) is rank_p(c))
6           ...
7       end
```

```
1       procedure optimize(F, N, n);
2           ▷ input: n: number of relations in the chain,
3                    N[i]: size of relation R_i, 0 ≤ i < n
4                    F[i, j]: selectivity between relations R_i and R_j,0 ≤ i, j < n
5           ▷ output: a permutation of 0, 1, ..., n − 1 representing the order of relations
6                     in an optimal left-deep processing tree for the join query R_0 ⋈ ··· ⋈ R_{n−1}
7                     in the case where cross products are allowed.
8           return join(sort-by-local-ranks(contr-subchains(optimal-rec-decomp-chains(F, N, n))));
9       end
```

Complexity: Let us briefly analyze the worst case time complexity of the second algorithm. The first step requires time $O(n^4)$ whereas the second step requires time $O(n^2)$. The third step has complexity $O(n \log n)$. Hence the total complexity is $O(n^4)$.

Algorithm II' is based on the cost function C'. We can now modify the algorithm for the original cost function C as follows.

Algorithm II:

1 Compute all optimal recursively decomposable chains by dynamic programming (corresponds to step 1 of Algorithm II')

2 For each starting relation R_i do the following steps:

- 2.1 Let L_1 be the result of applying the steps 2 and 3 of Algorithm II' to all optimal recursively decomposable subchains whose extent (N, M) satisfies $R_i \in N$ and $M \subseteq \{R_1, \ldots, R_i\}$.
- 2.2 Let L_2 be the result of applying the steps 2 and 3 of Algorithm II' to all optimal recursively decomposable subchains whose extent (N, M) satisfies $R_i\ N$ and $M \subseteq \{R_i, \ldots, R_n\}$.
- 2.3 Let L be the result of merging L_1 and L_2 according to their ranks
- 2.4 De-normalize L
- 2.5 Use R_iL to update the current-best join ordering

Complexity: The complexity of step 1 is $O(n^4)$ whereas the complexity of step 2 amounts to $\sum_{i=1}^{n}(O(i^2) + O(n-i)^2 + O(n)) = O(n^3)$. Hence, the time complexity of Algorithm II is $O(n^4)$.

Algorithm II is identical to Algorithm I except that it calls the function `optimize()` from Algorithm II'.

3.1.4 A Connection Between the Two Algorithms

In this section we investigate how the two algorithms are related. So far we know that the first algorithm is correct but may have exponential time complexity in the worst case. On the other side we know that the second algorithm has polynomial worst case complexity but we can not prove its correctness. Although the two algorithms have similarities it is not clear whether they yield the same results. In the following we prove that if the first algorithm does without backtracking (or if backtracking is not necessary since an optimal solution is unique) the second algorithm is correct.

First we introduce two operations which are useful in the transformation of decomposition trees. We call these operations *plucking* and *grafting*. Plucking removes a subtree from a tree while grafting inserts a new subtree.

Definition 3.1.10 (plucking/grafting) *Let T' be subtree of a decomposition tree T and T_l and T_r be the left and right subtrees of T', respectively. Then,* plucking T_r *is the transformation which replaces T' by T_l . The transformation which replaces T_l by a new node with T_l as its left descendant and T_l' as its right descendant is called* grafting T_l' above T_l from the right*. Similarly, replacing T_l by a new node with T_l' as its left descendant and T_l as its right descendant is called* grafting T_l' above T_l from the left*.*

We shall use these two operations as a single operation transforming one decomposition tree into another decomposition tree: A subtree T_1 is plucked and grafted above another subtree T_2. Note that this transformation maintains recursive decomposability if there is a connection between T_1 and T_2 (see Figure 3.4).

The next lemma states that the sequences of maximal contradictory subchains in an optimal recursive decomposition of a chain are rank-sorted with respect to the decomposition into left and right subchains.

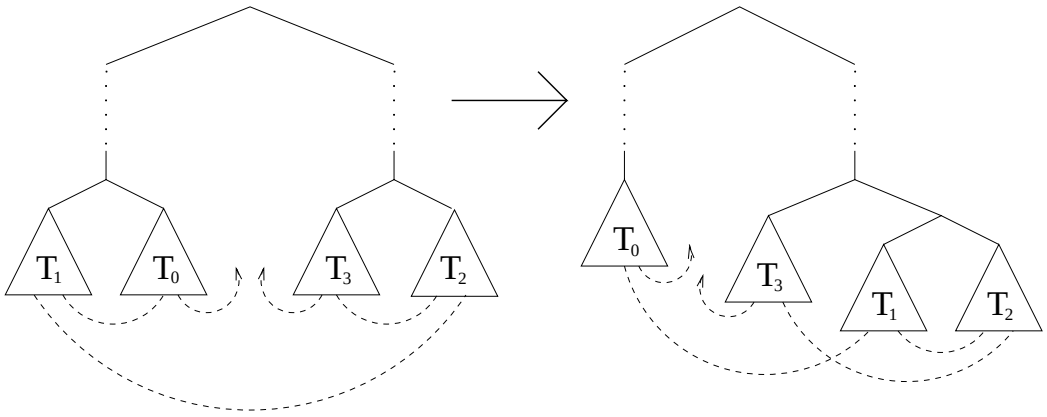


Figure 3.4: Plucking subtree T_1 and grafting it above subtree T_2 from the left maintains recursive decomposability. Dashed lines indicate connections between subchains.

Lemma 3.1.26 *Let s be an optimal recursively decomposable chain with respect to a given prefix u. Let $c_1 \dots c_k$ be a decomposition of s into maximal contradictory subchains (as produced by* `normalize`*), and let T be a decomposition tree of s whose left-to-right sequence of leaf labels is $c_1, \dots, c_k$. Let (u, s) be the label of an arbitrary node v in T, and let (u_l, s_l), (u_r, s_r) be the labels of the left and right sons of v, respectively. Then*

$$rank_{u_l}(s_l) \leq rank_u(s) \leq rank_{u_r}(s_r).$$

Proof Assume that the contrary is true and there exists a subtree T' of the decomposition tree T such that the following holds. Without loss of generality, we assume that $T = T'$. Let (u, s) be the label in the root of T' and let (u_l, s_l), (u_r, s_r) be the label in the root of the left and right subtrees of T, respectively. Let $s_l = c_1 \dots c_j$ and $s_r = c_{j+1} \dots c_k$ for some i, j $(1 \leq i \leq k)$. Finally, assume that $rank_{u_l}(s_l) > rank_{u_r}(s_r)$ holds.

Due to Lemma 3.1.16 we have

$$\min_{1 \leq i \leq j} rank_{u_l c_1 \dots c_{i-1}}(c_i) \quad > \quad \max_{j < i \leq k} rank_{u_r c_{j+1} \dots c_{i-1}}(c_i). \tag{3.5}$$

Let c_{e_1} be connected to c_{f_2} for some e_1, f_2 with $1 \leq e_1 \leq j$, $j+1 \leq f_2 \leq k$. Apart from the connection between c_{e_1} and c_{f_2}, c_{e_1} may have a second connection to the right and c_{f_2} may have a second connection to the left. Let us consider maximal "chains of connections" in either direction. Let T_1 be the smallest subtree of T that contains c_{e_1} and a maximal chain of connections to the right. Similarly, let T_2 be the smallest subtree of T that contains c_{f_2} and a maximal length chain of connections to the left. Let $x := c_{e_1} \dots c_{e_2}$ $(e_1 \leq e_2)$ be the subchain corresponding to T_1 and $y := c_{f_1} \dots c_{f_2}$ $(f_1 \leq f_2)$ the subchain corresponding to T_2. Using the abbreviations $x_l = c_1 \dots c_{e_1 - 1}$, $x_r = c_{e_2+1} \dots c_j$, $y_l = c_{j+1} \dots c_{f_1 - 1}$ and $y_r = c_{f_2+1} \dots c_k$, we have

$$s \quad = \quad x_l x x_r y_l y y_r.$$

We distinguish two cases.

1. Case $x_r y_l \neq \epsilon$. Without loss of generality, assume that $x_r \neq \epsilon$ (the case $y_l \neq \epsilon$ is analogous). Due to Lemma 3.1.16 and inequality (3.5), we have

$$rank_{u_l y_l y}(x_r) \quad > \quad rank_{u_r}(y_l y y_r).$$

Note that x_r is not connected to $y_l y y_r$. By Lemma 3.1.10 we can interchange the two subchains yielding the cheaper chain

$$s' \quad = \quad u_l x_l x y_l y y_r x_r$$

s' can also be obtained by the following transformations of the decomposition tree. First, the subtree T_2 is plucked, and then T_2 is grafted above T_x from the right, where T_x denotes the subtree corresponding to the subchain x. Since x and $y_l y y_r$ are connected, the chain obtained by this transformation is decomposable too. This is a contradiction to the fact that s is optimal among all recursively decomposable chains.

2. Case $x_r y_l = \epsilon$. We distinguish three subcases.

   (a) Case $e_1 = e_2$ and $f_1 = f_2$, i.e. c_j is connected to c_{j+1}. By inequality (3.5),

$$rank_{c_1 \ldots c_{j-1}}(c_j) \quad > \quad rank_{c_1 \ldots c_j}(c_{j+1}).$$

   Furthermore, since s is optimal recursively decomposable, we know that

$$C_{c_1 \ldots c_{j-1}}(c_j c_{j+1}) \quad \leq \quad C_{c_1 \ldots c_{j-1}}(c_{j+1} c_j),$$

   hence (c_j, c_{j+1}) would build a contradictory pair of subchains, a contradiction to the maximality of c_j and c_{j+1}.

   (b) Case $e_1 < e_2$, i.e. c_{e_1} is connected to y. Denote c_{e_1} with x' and $c_{e_1+1} \ldots c_{e_2}$ with x'', i.e. $x = x'x''$. By Lemma 3.1.16 and inequality (3.5),

$$\begin{aligned} rank_{u_l c_1 \ldots c_{e_1}}(x'') &\geq \min_{e_1 < i \leq e_2} rank_{u_l c_1 \ldots c_{i-1}}(c_i) \\ &\geq \min_{1 \leq i \leq j} rank_{u_l c_1 \ldots c_{i-1}}(c_i) \\ &> \max_{j < i \leq k} rank_{u_r c_{j+1} \ldots c_{i-1}}(c_i) \\ &\geq rank_{u_r}(s_r). \end{aligned}$$

   By Lemma 3.1.10 we can obtain a chain with lower costs by interchanging x'' and s_r (note that x'' is not connected to s_r). An equivalent transformation of the decomposition tree is to pluck T_r , and then graft T_r above the subtree corresponding to x' from the right. Since x' is connected to s_r the resulting sequence of relations is recursively decomposable too. This is a contradiction to the optimal recursive decomposability of s.

   (c) Case $f_1 < f_2$. This case is symmetric to case 2b and can be treated in a similar way (interchange y' with x, where $y' := c_{e_1} \ldots c_{f_2 - 1}$).

   All cases lead to a contradiction. Consequently, our assumption was wrong and $rank_{u_l}(s_l) \leq rank_{u_r}(s_r)$. The claim now follows by Lemma 3.1.15. □

The following two lemmata show why contradictory subchains in an optimal recursively decomposable chain can be sorted by rank without violating the precedences imposed by the neighborhoods. The next lemma essentially states that the ordering of two contradictory subchains that are connected agrees with their relative order in the chain.

Lemma 3.1.27 *Let s be an optimal recursively decomposable chain and u an accompanying prefix. Suppose the normalization of s yields the contradictory subchains $c_1 \ldots c_k$. Then, for all $1 \leq i < j \leq k$:*

$$c_i \text{ is connected to } c_j \;\Rightarrow\; rank_{c_1 \ldots c_{i-1}}(c_i) \leq rank_{c_1 \ldots c_{j-1}}(c_j)$$

Proof Let T be a decomposition tree for s. Consider an arbitrary pair c_i, c_j $(1 \leq i < j \leq k)$ of interconnected maximal contradictory subchains in s. Apart from the connection to c_j, c_i may have a second connection to contradictory subchains to the right of c_i. Let us consider a maximal length sequence of such connections to the right until there is the first connection to the left. Let T_1 be the smallest subtree of T that contains c_i and a maximal length sequence of connections to the right. Similarly, c_j may have a sequence of connections to the left. Let T_2 be the smallest subtree of T that contains c_j and a maximal sequence of connections to the left. Let $x = c_i x'$ and $y = y' c_j$ be the subchains corresponding to the subtrees T_1 and T_2, respectively. Note that there is a connection between T_1 and T_2 in T. Denote the subchains left and right of x with x_l and x_r, respectively. The subchains left and right of y are denoted with y_l and y_r, respectively. Hence, we have

$$s \quad = \quad x_l x x_r y_l y y_r$$

Since there is an optimal recursive decomposition that splits x into the left subchain c_i and the right subchain x', Lemma 3.1.15 yields

$$rank_{u_l x_l}(c_i) \quad \leq \quad rank_{u_l x_l}(x) \tag{3.6}$$

By an analogous argumentation

$$rank_{u_l x_l x x_r y_l}(y) \quad \leq \quad rank_{u_l x_l x x_r y_l y'}(c_j). \tag{3.7}$$

Now, we distinguish two cases.

1. Case $x_r y_l = \epsilon$. Let T_3 be a new decomposition tree with T_1 as its left subtree and T_2 as its right subtree. Then T_3 represents an optimal recursive decomposition of the subchain xy (otherwise s would not be optimal recursively decomposable). Hence, by Lemma 3.1.15,

   $$rank_{u_l x_l}(x) \leq rank_{u_l x}(y)$$

   and due to inequalities (3.6) and (3.7),

   $$\begin{aligned} rank_{u_l x_l}(c_i) &\leq rank_{u_l x_l}(x) \\ &\leq rank_{u_l x}(y) \\ &\leq rank_{u_l x x'}(c_j) \end{aligned}$$

2. Now, consider the case $x_r y_l \neq \epsilon$. We assume that

   $$rank_{u_l x_l}(c_i) > rank_{u_l x_l x x_r y_l y'}(c_j) \tag{3.8}$$

   and derive a contradiction. We distinguish two subcases.

   (a) Case $rank_{u_l x_l}(x) > rank_{u_l x_l x}(x_r y_l)$. Since x is not connected to $x_r y_l$ we can apply Lemma 3.1.10 and interchange the subchains in order to lower the costs. The resulting chain

   $$x_l x_r y_l x y y_r$$

   is recursively decomposable too, since it corresponds to the following transformations of the decomposition tree. T_2 is plucked and grafted above T_1 from the left. Note the connection between T_1 and T_2.

   (b) Case

   $$rank_{u_l x_l}(x) \leq rank_{u_l x_l x}(x_r y_l) \tag{3.9}$$

By the inequalities (3.6), (3.7) and (3.8),

$$\begin{aligned} rank_{u_l x_l c_i}(x') &\geq rank_{u_l x_l}(x) \\ &\geq rank_{u_l x_l}(c_i) \\ &> rank_{u_l x_l x x_r y_l y'}(c_j) \\ &\geq rank_{u_l x_l x x_r y_l}(y) \end{aligned}$$

and due to inequality (3.9),

$$\begin{aligned} rank_{u_l x_l x}(x_r y_l) &\geq rank_{u_l x_l}(x) \\ &> rank_{u_l x_l x x_r y_l}(y) \end{aligned}$$

Note that $x_r y_l$ is not connected to y. By Lemma 3.1.10 we can interchange the two subchains yielding the cheaper chain

$$s' = x_l x y x_r y_l y_r,$$

a contradiction to the optimal recursive decomposability of s. Hence our assumption (3.8) was wrong and

$$rank_{u_l x_l}(c_i) \leq rank_{u_l x_l x x_r y_l y'}(c_j).$$

Note that s' is recursively decomposable, since there exists a corresponding transformation of the decomposition tree that maintains the recursive decomposability. The transformation is to pluck T_1, and graft T_1 above T_2 from the right.

In both cases we derived a chain s' which is recursively decomposable and cheaper than s. This is a contradiction to the optimal recursive decomposability of s. Hence our assumption (3.8) was wrong.

□

Lemma 3.1.28 *The maximal contradictory subchains in an optimal recursively decomposable chain s can be sorted according to their ranks without changing the relative order of two connected contradictory subchains. Moreover, the resulting chain s' cannot have larger costs than s.*

Proof The proof is by induction on the number of pairs of contradictory subchains $[c_1]_{u_1}$, $[c_2]_{u_2}$ such that c_1 is not connected to c_2, c_1 precedes c_2, and $rank_{u_1}(c_1) > rank_{u_2}(c_2)$. We shall call such pairs *inversions*.
In the base case, there are no inversions and for every pair of adjacent contradictory subchains the following holds. If the two subchains are connected, then by Lemma 3.1.27 they are rank-sorted. If the two subchains are not connected, they must be rank-sorted (otherwise they would represent an inversion).
Now, assume that the claim is true for all optimal recursively decomposable chains with $n-1$ inversions, for some $n > 0$. Consider an arbitrary recursively decomposable chain s with n inversions. Since s is not rank-sorted, there exist two adjacent maximal contradictory subchains $c_1 c_2$ that are not rank-sorted. Note that c_1 cannot be connected to c_2, hence c_1, c_2 represents an inversion in s. Interchanging c_1 with c_2 decreases the number of inversions by one and does not increase the costs (Lemma 3.1.10). We can now apply the induction hypothesis to the resulting chain and the claim follows. □

The following theorem shows a connection between the number of solutions produced by the first algorithm and the correctness of the second algorithm.

Theorem 3.1.2 *The second Algorithm is correct if the first Algorithm always yields exactly one decomposition into contradictory subchains.*

Proof Suppose that the first algorithm computes only one decomposition X. Since the first algorithm is correct, X must be the unique optimum. Now we have to show that the second algorithm computes X too. Let Y be the decomposition computed by the second algorithm and assume that Y is different from X. According to Lemma 3.1.28 we know that Y consists of a sequence of rank-sorted, optimal recursively decomposable maximal contradictory subchains. This is a contradiction, since Y would also have been enumerated by the first algorithm. Hence our assumption was wrong and $X = Y$. □

We conjecture that the other direction of Theorem 3.1.2 holds too: [6]

Conjecture 3.1.2 *Assuming that there is a unique optimal plan, the first Algorithm yields exactly one decomposition into contradictory subchains if the second Algorithm is correct.*

If conjecture 3.1.2 is true, both algorithms produce optimal left-deep processing trees in polynomial time. Nevertheless, since we could not prove our conjecture, we have to consider both the first algorithm (without backtracking) and the second algorithm merely as good heuristic algorithms.

Due to the lack of hard facts, we ran about 700,000 experiments with random queries of sizes up to 30 relations and fewer experiments for random queries with up to 300 relations to compare the results of our algorithms. For $n \leq 15$ we additionally compared the results with a standard dynamic programming algorithm [SAC+79]. The results of all our experiments can be summarized as follows.

- All algorithms yielded identical results.
- Backtracking always led to exactly one sequence of contradictory chains.
- In the overwhelming majority of cases the first algorithm proved to be faster than the second algorithm.

Whereas the run time of the second algorithm is mainly determined by the number of relations in the query, the run time of the first algorithm also heavily depends on the number of existing optimal contradictory subchains. In the worst case, the first algorithm is slightly inferior to the second algorithm.

[6] In [SM97] this conjecture was formulated as a theorem. Shortly after publication we discovered an error in the proof of the theorem.

3.2 Acyclic Queries with Joins and Selections

The section is dedicated to the optimization of selection-join-queries with expensive predicates. The generally accepted optimization heuristics of pushing selections down does not yield optimal plans in the presence of expensive predicates. Therefore, several researchers have proposed algorithms to compute optimal processing trees with expensive predicates. All these algorithms have exponential run time. For a special case, we propose a polynomial algorithm which—in one integrated step—computes the optimal join order and places expensive predicates optimally within the processing tree. The special case is characterized by the following statements. Only left-deep trees without cross products are considered. Expensive selections can only be placed on the path from the leftmost leaf to the root of the tree. Cheap selections are pushed before-hand, and the cost function has to exhibit the ASI property [IK84].

3.2.1 Preliminaries

A query is represented by a set of query predicates P, where each $p \in P$ is either a selection predicate p_i referring to a single relation R_i or a join predicate $p_{i,j}$ connecting relations R_i and R_j.

Let $R_1, \ldots, R_n$ be the relations involved in the query. Associated with each relation is its size $n_i = |R_i|$. The predicates in the query induce a join graph G whose edges consist of all pairs $\{R_i, R_j\}$ for which there exists a predicate $p_{i,j}$ relating R_i and R_j. We assume that the join graph is *acyclic*. The selectivity of a join predicate $p_{i,j} \in P$ is denoted with $f_{i,j}$, and the selectivity of a selection predicate $p_i \in P$ is denoted with f_i. The evaluation of a join or selection predicate can be of different costs. We denote by $c_{i,j}$ the costs of evaluating predicate $p_{i,j}$ for one tuple of $R_i \times R_j$. Similarly, c_i denotes the per-tuple-costs associated with the selection predicate p_i.

In the following we consider only left-deep processing trees. Every left-deep processing tree can be represented by an algebraic expression of the form

$$(\ldots((R_1\,\psi_1)\,\psi_2)\,\ldots\,\psi_m),$$

where the *unary* operators ψ_i $(i = 1 \ldots n)$ are either selections σ_{p_i} or joins $\bowtie_{p_{i,j}} R_j$. R_1 is called the *starting relation*.

There are different implementations of the join operator, each leading to different cost functions for the join and hence to different cost functions for the whole processing tree. Common implementations of a binary join operator are (cf. [Gra93, ME92, Ull89])

- nested loop join
- hash loop join
- sort merge join

The corresponding cost functions [KBZ86] are

$$\begin{aligned} C_{nl}(R \bowtie S) &= |R| \cdot |S| + |R| \cdot |S| \cdot f_{RS} \\ C_{hl}(R \bowtie S) &= 1.2 \cdot |R| + |R| \cdot |S| \cdot f_{RS} \\ C_{sm}(R \bowtie S) &= (|R| \cdot \log|R| + |S| \cdot \log|S|) + |R| \cdot |S| \cdot f_{RS} \end{aligned}$$

Here we made the important assumption that our database is *memory resident*, i.e. there is no paging to disk during execution of a query. The first summand in the cost functions accounts for

the costs of iterating over the relations and for checking the join predicate. The second sum which is identical for all cost functions, accounts for the costs to construct the intermediate results. The factor 1.2 stands for the average length of the collision list of the hash table.
In order to approximate the costs of n-way joins, we associate with each of the cost functions C_{nl}, C_{hl}, C_{sm} operating on join-expressions a corresponding binary cost function g working on input sizes:

$$\begin{aligned} g_{nl}(r,s) &= r \cdot s + r \cdot s \cdot f_{RS} \\ g_{hl}(r,s) &= 1.2 \cdot r + r \cdot s \cdot f_{RS} \\ g_{sm}(r,s) &= (r \cdot \log r + s \cdot \log s) + r \cdot s \cdot f_{RS} \end{aligned}$$

Since the sizes of the intermediate results play an important role in all cost functions, it is a central problem to determine these sizes. Under the usual assumptions of independent and uniformly distributed attribute values, the following standard approximation holds [Ull89]:

$$|R_1 \bowtie \ldots \bowtie R_k| \approx \prod_{i=1}^{k} |R_i| \prod_{j<i} f_{ij}$$

Hence we can write

$$\begin{aligned} C(R_{\pi(1)} \bowtie \ldots \bowtie R_{\pi(n)}) &= \sum_{k=2}^{n} g_k(|R_{\pi(2)} \bowtie \ldots \bowtie R_{\pi(k)}|, |R_{\pi(k)}|) \\ &= \sum_{k=2}^{n} g_k(\prod_{i=1}^{k} |R_{\pi(i)}| \prod_{j<i} f_{\pi(i)\pi(j)}, |R_{\pi(k)}|) \end{aligned}$$

where g_k is one of the functions C_{nl}, C_{hl} depending on the join algorithm used. Since C_{nl} and C_{hl} are both linear in the first argument, we can "extract" the linear factor and define the unary cost function g as

$$\begin{aligned} g_{nl}(s) &= s + s \cdot f_{RS} \\ g_{hl}(s) &= 1.2 + s \cdot f_{RS} \end{aligned}$$

Henceforth we will use the unary function g. Now, we have

$$C(R_{\pi(1)} \bowtie \ldots \bowtie R_{\pi(n)}) = \sum_{k=2}^{n} |R_{\pi(2)} \bowtie \ldots \bowtie R_{\pi(k)}| \, g_k(|R_{\pi(k)}|)$$

Please note that C covers almost all cost functions for joins as pointed out in [KBZ86] and it even covers the nontrivial cost function given in [IK84]. However, it does not account for expensive join predicates. These will be taken care of in the next section.

Next, we repeat some fundamental results concerning the optimization of *cost functions with ASI-property* and the *IK-algorithm* of Ibaraki and Kameda [IK84]. Ibaraki and Kameda were the first to recognize and successfully exploit a connection between a certain class of sequencing problems with ASI cost functions [MS79] and the traditional join ordering problem for left-deep trees without cross products.

As mentioned earlier, every left-deep tree corresponds to a permutation indicating the order in which the base relations are joined with the intermediate result relation. We will henceforth speak of permutations or sequences instead of left-deep processing trees.

We have just seen that all cost functions in the standard cost model for left-deep trees have the form

$$\begin{aligned} Cost(s) &= \sum_{i=2}^{n} |s_1 \dots s_{i-1}| * g_i(|s_i|) \\ &= \sum_{i=2}^{n} (\prod_{j=1}^{i-1} f_j * |s_j|) * g_i(|s_i|) \end{aligned} \tag{3.10}$$

where the function g_i accounts for the join algorithm used in the respective step. As can easily be verified, there is a recursive definition of these cost functions:

$$\begin{aligned} C(\epsilon) &= 0 \\ C(R_j) &= 0 \qquad \text{if } R_j \text{ is the starting relation} \\ C(R_j) &= g_j(|R_j|) \qquad \text{else} \\ C(s_1 s_2) &= C(s_1) + T(s_1) * C(s_2) \end{aligned}$$

with

$$\begin{aligned} T(\epsilon) &= 1 \\ T(s) &= \prod_{i=1}^{n} f_i s_i \end{aligned}$$

Here, s_1, s_2 and s denote sequences of relations.

We now define the *ASI property*[7] [MS79] of a cost function.

Definition 3.2.1 *(ASI property)*
A cost function C has the ASI property, if there exists a rank function $r(s)$ for sequences s, such that for all sequences a and b and all non-empty sequences u and v the following holds:

$$C(auvb) \leq C(avub) \Leftrightarrow r(u) \leq r(v)$$

For a cost function of the above form, we have the following lemma:

Lemma 3.2.1 *Let C be a cost function which can be written in the above form.*
Then C has the ASI property for the rank function

$$r(s) = \frac{T(s) - 1}{C(s)}$$

for non-empty sequences s.

Let us consider sequences with a fixed starting relation, say R_1. Since we do not allow any cross products in a processing tree, the second relation in a feasible join sequence is restricted to relations which are adjacent to R in the join graph. Similar restrictions hold for all following relations. These restrictions define a *precedence relation* on the set of all base relations. The graph of the precedence relation is a directed version of the join tree with R_1 being the root and all other relations directed away from the root. This shows that we actually have a *sequencing problem with tree-like precedence constraints* where the cost function satisfies the *ASI-property.*

[7] adjacent sequence interchange property

For the unconstrained sequencing problem—that is, if we had no precedence constraints—sorting the relations according to their rank leads to an optimal sequence![8] But if precedence constraints are present, they often make it impossible to sort the relations according to their ranks. The next result provides a means to resolve the conflict of ordering according to the precedence constraints and ordering according to the rank. A *composite relation* is defined as an ordered pair (r, s) where r and s are either single or composite relations and the condition $r(r) > r(s)$ holds. Rank, cost and size of the composite relation are defined to be the respective values of the sequence rs. The precedence relation generalizes to sequences of relations in a straightforward way. In [MS79] it is shown that if for two composite sequences r and s, where r precedes s in the precedence tree and $r(r) > r(s)$, there is an optimal sequence with r immediately preceding s. By iterating the process of tying pairs of composite relations together whose rank and precedence stay in conflict, we eventually arrive at a sequence of composite relations which is sorted by rank. This process of iterated tying is called *normalization*.

The reader can probably already anticipate the outlines of a recursive algorithm for solving the join ordering problem with a given starting relation: one starts at the bottom of the directed join tree and works upwards. To obtain the optimal sequence for a subtree of relations where all children are chains one simply normalizes each of the chains and merges them according to the ranks. The resulting sequence is again rank ordered and we replace the subtree by the corresponding chain of composite relations. By considering every base relation as starting relation, computing the optimal sequence for this starting relation and then choosing the cheapest of these sequences, we can determine an optimal sequence for the join ordering problem.
This is basically the *IK-algorithm* described in [IK84]. In [Law78], Lawler gives an efficient implementation of this algorithm that runs in time $O(n \log n)$, using a set representation instead of the straightforward sequence representation. The following description of the IK-algorithm is taken from [IK84].

[8] This is not true for the join ordering problem, where the analog problem would consider cross products.

Algorithm NORMALIZE(S):
input: a chain of nodes S
output: chain of nodes

1 while there is a pair of adjacent nodes, S_1 followed
by S_2, in S such that $r(S_1) > r(S_2)$ do
2 Find the first such pair (starting from the beginning of S) and
replace S_1 and S_2 by a new composite node (S_1, S_2).

Algorithm TREE-OPT(Q):
input: tree query Q with specified root R, the relations referenced in Q,
their sizes, and the predicates of Q together with their selectivity factors
output: optimal join ordering for Q with starting relation R

1 Construct the directed tree T_R with root R.
2 If T_R is a single chain, then stop. (The desired join ordering is given by the chain.)
3 Find a subtree whose left and right subtrees are both chains.
4 Apply NORMALIZE to each of the two chains.
5 Merge the two chains into one by ordering the nodes by increasing ranks,
and go to step 2.

Algorithm IK(Q):
input: tree query Q with specified root R, the relations referenced in Q,
their sizes, and the predicates of Q together with their selectivity factors
output: optimal join ordering for Q

1 Let S be some fixed initial ordering and C the costs of S
Let $R_1, \ldots, R_n$ be the relations involved in the query.
2 for $i \leftarrow 1$ to N do
3 Apply TREE-OPT to the directed join tree with root R_i
4 If the optimal join ordering starting with R_i has better costs than C, then
update C and S
5 return S

A slightly more efficient version of this algorithm is the *KBZ-algorithm* of Krishnamurthy, Boral and Zaniolo [KBZ86]. Their algorithm has a worst-case time complexity of $O(n^2)$. The idea is to reuse the computed optimal sequence for a starting relation R to compute an optimal sequence for a starting relation R' being adjacent to R in the join graph. This leads to a considerable reduction of work. For details we refer to [KBZ86].

3.2.2 Ordering Expensive Selections and Joins

Let us extend the notion of a precedence tree to capture select-join queries. Suppose we are to use a distinguished relation – say R_1 – as the starting relation in the processing tree (the leftmost leaf). Then, since no cross products are allowed, the join tree becomes a rooted tree with R_1 as the root. We can extend the directed tree to incorporate all the selection operators as follows. For every selection operator $\psi_i = \sigma_{p_i}$ relating to a single relation R_i, we add a new successor node to R and label it with ψ_i. The resulting tree defines a precedence relation among the operators ψ_i and we call it the *precedence tree* of the query with respect to the starting relation R_1. For an example see the end of this section.

For each operator ψ_i we define the cost factor d_i as

$$d_i = \begin{cases} c_i & \text{if } \psi_i \equiv \sigma_{p_i} \\ g(|R_i|) * c_{j,i} & \text{if } \psi_i \equiv \bowtie_{p_{j,i}} R_i \end{cases}$$

for $i > 1$ and $d_1 = 0$, where c_i and $c_{j,i}$ denote the cost of evaluating p_i and $p_{j,i}$ for one tuple, respectively. j is the index of the unique predecessor of R_i in the precedence tree. The size factors h_i are defined as

$$h_i = \begin{cases} f_i & \text{if } \psi_i \equiv \sigma_{p_i} \\ |R_i| * f_{j,i} & \text{if } \psi_i \equiv \bowtie_{p_{j,i}} R_i \end{cases}$$

d_i accounts for the costs incurred by applying operator ψ_i to an intermediate result R whose generation was in accordance with the precedence tree. Whenever ψ_i is applied to such an intermediate result R, we expect R to grow by a factor of h_i. We call a sequence *feasible*, if it satisfies all ordering constraints implied by the present attributes in the predicates of the operators, as expressed in the precedence tree. In the following we identify permutations and sequences.

For a sequence s we define the costs[9]

$$Cost(s) = |R_1| \sum_{i=2}^{n} F_{i-1}^{s} d_{s(i)} = |R_1| \sum_{i=2}^{n} \prod_{j=2}^{i-1} h_{s(j)} d_{s(i)},$$

where $s(i)$ is the i-th element of the operator sequence s and the intermediate result size F is given by

$$F_i^s = \prod_{j=2}^{i} h_{s(j)}$$

Then, we have to solve the following optimization problem

$$\text{minimize}_s \; [Cost(s)],$$

where the minimization is taken over all feasible sequences s.
Since R_1 just contributes a constant factor, it can easily be dropped from $Cost(s)$ without changing the optimization problem.

Our goal is to apply the IK and KBZ algorithms. Hence, we have to find a rank function for which the cost function satisfies the ASI property. We do so by first recasting the cost function into a more appropriate form. For

$$F(s) = \prod_{k \in s} h_k$$

we define the binary function C as

$$C(j, \epsilon) = C(\epsilon, j) = c_j \text{ for } j \in \{1, \ldots, n\}$$

and

$$C(s, t) = C(s', s'') + F(s)C(t', t'') \text{ for sequences } s, t$$

where $s = s's''$ and $t = t't''$ with $|s| > 1 \Rightarrow |s'| \geq 1 \wedge |s''| \geq 1$, and $|t| > 1 \Rightarrow |t'| \geq 1 \wedge |t''| \geq 1$.

[9] Empty sums equal 0, empty products 1.

A simple induction shows that C is consistent, that is, $s_1 s_2 = s_1' s_2'$ implies $C(s_1, s_2) = C(s_1', s_2')$. With the binary C being consistent, the unary C defined as

$$\begin{aligned} C(j) &= c_j \text{ for } j \in \{1, \ldots, n\} \\ C(st) &= C(s,t) \text{ for sequences } s,t \text{ with } |s| \geq 1 \ \wedge \ |t| \geq 1 \end{aligned}$$

is well-defined. Another simple proof by induction shows that the functions *Cost* and the unary C are equal for all feasible s. Summarizing, the following three lemmata hold.

Lemma 3.2.2 *The binary cost function C is consistent.*

Lemma 3.2.3 *The unary cost function C is well-defined.*

Lemma 3.2.4 *The unary cost function C and the cost function* Cost *are the same.*

Now, we come to the central lemma, which will allow us to apply the IK- and the KBZ-algorithms to our problem of optimally ordering expensive selections and joins with expensive predicates simultaneously.

Lemma 3.2.5 *C satisfies the ASI property [MS79] with*

$$r(s) = \frac{F(s) - 1}{C(s)}$$

being the rank *of a sequence s.*

Proof: We have to proof that

$$C(ustv) \leq C(utsv) \Leftrightarrow r(s) \leq r(t)$$

for all sequences u and v. Since

$$\begin{aligned} C(ustv) &= C(us) + F(us)C(tv) \\ &= C(u) + F(u)C(s) + F(us)[C(t) + F(t)C(v)] \\ &= C(u) + F(u)C(s) + F(us)C(t) + F(us)F(t)C(v) \end{aligned}$$

the following holds

$$\begin{aligned} C(ustv) - C(utsv) &= F(u)[C(t)(F(s) - 1) - C(s)(F(t) - 1)] \\ &= F(u)C(t)C(s)[r(s) - r(t)] \end{aligned}$$

With this equation the ASI property follows for C. □

Using the results summarized in section 3.2.1, we can apply the IK- or KBZ-algorithm. Both guarantee to find an optimal solution in time $O(n^2 \log(n))$ and $O(n^2)$, respectively. Since we do only consider strict left-deep trees, *non-expensive* selections will be placed after the corresponding join in any case! To avoid this drawback, we push cheap selections down the query tree prior to the invocation of the algorithm. Note that this preprocessing step changes the sizes of some relations which must be respected.

Next, we illustrate how the IK-algorithm in case of our new rank definition works.

Example: Consider the following select-join-query involving six relations:

$$\sigma_{p_2}(\sigma_{p_3}(\sigma_{p_5}(R_1 \bowtie_{p_{1,2}} R_2 \bowtie_{p_{1,3}} R_3 \bowtie_{p_{2,4}} R_4 \bowtie_{p_{3,5}} R_5 \bowtie_{p_{5,6}} R_6)))$$

The are eight operators. Three of them are (expensive) selections and five are joins. The associated selectivities, relation sizes and cost factors are specified in the three tables below.

n_1	n_2	n_3	n_4	n_5	n_6
50	60	30	10	40	20

$f_{1,2}$	$f_{1,3}$	$f_{2,4}$	$f_{3,5}$	$f_{5,6}$	f_2	f_3	f_6
0.6	0.7	0.05	0.3	0.2	0.5	0.6	0.4

$c_{1,2}$	$c_{1,3}$	$c_{2,4}$	$c_{3,5}$	$c_{5,6}$	c_2	c_3	c_6
6	5	2	7	4	10	4	3

The join graph is shown in Figure 1(a) and Figure 1(b) shows the directed join graph rooted at the starting relation R_1, i.e., the precedence graph.

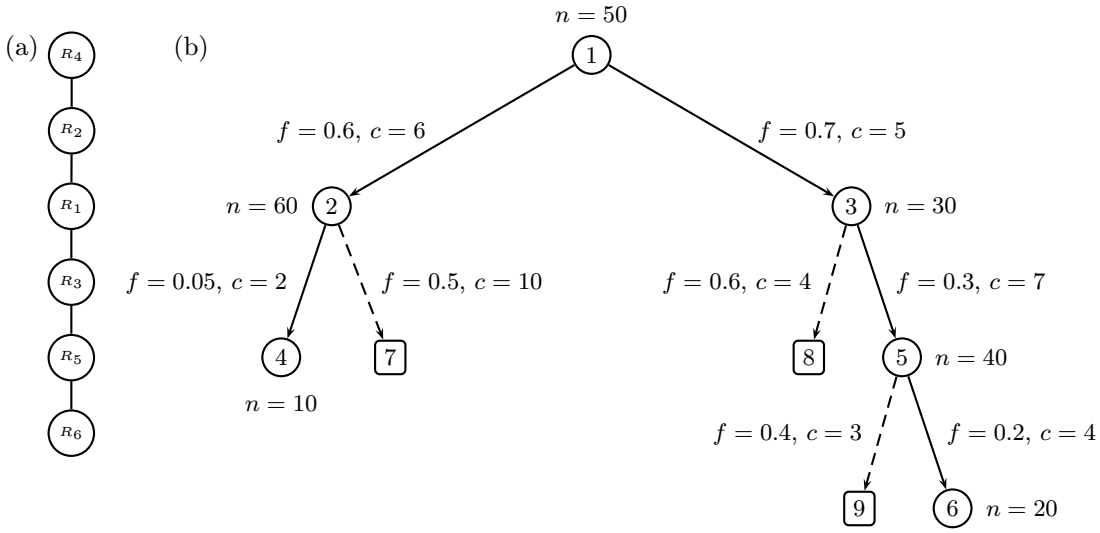


Figure 1: (a) join graph in the example query, (b) the associated precedence tree; selectivities, relation sizes and cost factors are shown. Dashed arrows pointing to square boxes indicate selections and solid arrows pointing to circular boxes correspond to joins.

Instead of considering every relation as a starting relation as the IK-algorithm does, we restrict ourselves to the single starting relation R_1. Since the nodes in the rooted tree uniquely correspond to the operators in the query, we henceforth use them interchangeably. For this we use the following coding scheme. Suppose the query has m selections and involves n base relations. Then, operator ψ_i $(1 < i \leq n)$ corresponds to the join operation $\bowtie_{p_{j,i}} R_i$ where j is the unique predecessor node in the precedence tree. For $n < i \leq n + m$, operator ψ_i corresponds to the selection operator σ_{p_j} where j is the unique predecessor node of i in the precedence tree. Operator ψ_1 represents an exception, it corresponds to the starting relation R_1. Nodes in the precedence tree are labeled with the number of the corresponding operator, i.e. node i $(1 \leq i \leq n + m)$ corresponds to operator

ψ_i. E.g., in our example, node 5 corresponds to operator ψ_5, which is the join $\bowtie_{p_{3,5}} R_5$ whereas node 7 corresponds to operator ψ_7, which is the selection σ_{p_2}.

In this example, we assume that all joins are hash-loop joins. The IK-algorithm works bottom-up. Let us first process the subtree with root 5. The sons of node 5 are the leaves 6 and 9 which are trivially ordered by rank. Node 6 is a join operator and its rank is

$$\begin{aligned} r(\psi_6) &= \frac{F(\psi_6)-1}{C(\psi_6)} \\ &= \frac{f_{5,6}n_6-1}{1.2\cdot c_{5,6}} \\ &= \frac{0.2*20-1}{1.2*4} = 0.625 \end{aligned}$$

Node 9 is a selection operator with rank

$$\begin{aligned} r(\psi_9) &= \frac{F(\psi_9)-1}{C(\psi_9)} \\ &= \frac{f_5-1}{c_5} \\ &= \frac{0.4-1}{3} = -0.20 \end{aligned}$$

Now we can merge the two nodes. Since $r(\psi_9) < r(\psi_6)$, node 9 has to precede node 6 and we can replace the subtree rooted at 5 with the chain 5-9-6. Next, we examine whether this chain is still sorted by rank. The rank of node 5 is

$$r(\psi_5) = \frac{0.3*40-1}{1.2*7} = 1.31$$

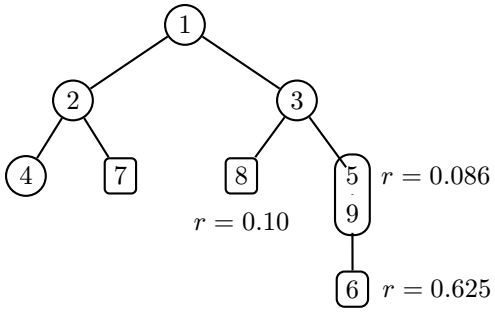
This shows that the ranks of the nodes 5 and 9 contradict their precedence and we have to tie these two nodes together as a composite node (5,9). The rank of the new node (5,9) is

$$\begin{aligned} r(\psi_5\,\psi_9) &= \frac{F(\psi_5\,\psi_9)-1}{C(\psi_5\,\psi_9)} \\ &= \frac{n_5 f_{3,5} f_5 - 1}{1.2c_{3,5} + n_5 f_{3,5} c_5} \\ &= \frac{40*0.3*0.4-1}{1.2*7+40*0.3*3} = 0.086 \end{aligned}$$

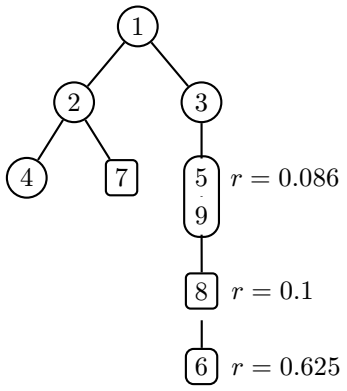
For the rank of the selection node 8 we have

$$r(\psi_8) = \frac{0.6-1}{4} = 0.1$$

and the new rooted join tree is



In the next step we merge node 8 and the chain consisting of the composite node (5,9) succeeded by node 6. The corresponding join tree is



The rank of the join node 3 is

$$r(\psi_3) = \frac{30 * 0.7 - 1}{1.2 * 5} = 3.333$$

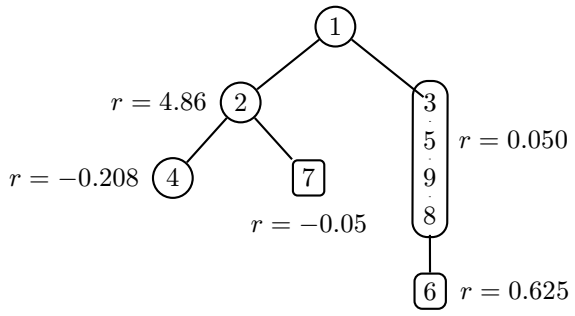
Since the nodes 3 and (5,9) have contradictory ranks, we build the new composite relation (3,5,9) with rank

$$\begin{aligned} r(\psi_3\,\psi_5\,\psi_9) &= \frac{n_3 f_{1,3} n_5 f_{3,5} f_5 - 1}{1.2 \cdot c_{1,3} + n_3 f_{1,3} \cdot 1.2 \cdot c_{3,5} + n_3 f_{1,3} n_5 f_{3,5} c_5} \\ &= \frac{30 * 0.7 * 40 * 0.3 * 0.4 - 1}{1.2 * 5 + 30 * 0.7 * 1.2 * 7 + 30 * 0.7 * 40 * 0.3 * 3} = 0.106 \end{aligned}$$

The nodes (3,5,9) and 8 still have contradictory ranks and must be tied together again. The new rank is

$$r(\psi_3\,\psi_5\,\psi_9\,\psi_8) = 0.050$$

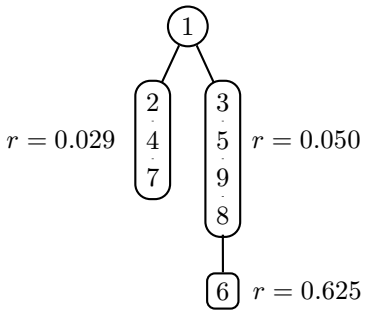
and the new join tree has the form



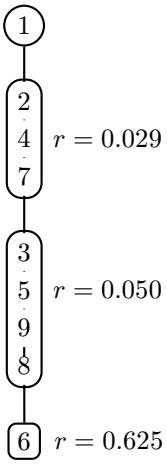
After having linearized the right subtree of R_1, we proceed with the left subtree. The ranks of nodes 2,4 and 7 are

$$r(\psi_2) = 4.86,\ r(\psi_4) = -0.208,\ r(\psi_7) = -0.05$$

We merge the subtree rooted at 2 and then normalize the resulting chain 2-4-7. The pair 2 and 4 has contradictory ranks, hence we build the composite node (2,4). Since the rank of (2,4) is 0.182, which is still greater then the rank of the succeeding node 7, we add 7 to the end of (2,4). The rank of the composite node (2,4,7) evaluates to 0.029 and the new precedence tree is



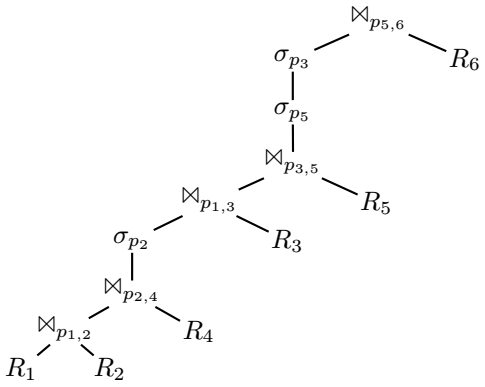
Finally, the left and right chains of R_1 are merged, yielding



As result, we have that the final sequence of operators

$$\psi_1\, \psi_2\, \psi_4\, \psi_7\, \psi_3\, \psi_5\, \psi_9\, \psi_8\, \psi_6$$

which correspond to the following optimal left-deep precessing tree for the starting relation R_1



Analogous computations are made for the precedence trees rooted at the relations R_2, R_3, R_4, R_5 and R_6. The cheapest of all the n operator sequences is the result of the IK-algorithm.

Chapter 4

Generation of Optimal Bushy Execution Plans

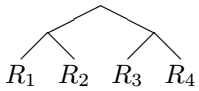
Since their introduction in [SAC+79], processing trees have traditionally been restricted to be left-deep. Lately, the much larger search space of bushy trees was considered. The vast majority of query optimization papers considers connected query graphs. For connected query graphs there is no need to introduce cross products into the processing trees. Since cross products are considered to be very expensive, for a given query only the processing trees not containing cross products were considered. This heuristic was also introduced in [SAC+79] in order to restrict the search space. Lately, it was shown that including bushy trees and cross products into the search space yields non-neglectable performance gains over the restricted search space [OL90].

An interesting question that arises with bushy trees is the following. Is there a (sub-) problem for which a polynomial algorithm generating optimal bushy trees exists? The result presented is quite discouraging. In section 4.1, we prove that constructing optimal bushy trees for a set of relations whose cross product has to be computed is NP-hard. This contrasts the left-deep case where the optimal left-deep tree can easily be computed by sorting the relations on their sizes. Moreover, since taking the cross product is a very special case in which all join selectivities are set to one, constructing optimal bushy trees for any join problem—independent of the join graph—is NP-hard. Thus, any hope of constructing optimal bushy trees in polynomial time has to be abandoned for whatever join problem at hand.

Consequently, section 4.2 focuses on the development of new, particularly efficient dynamic programming algorithms for the general problem of computing optimal bushy trees with cross products for queries containing joins, cross products, and expensive selections.

4.1 The Complexity of Computing Optimal Bushy Processing Trees

We assume that the cross product of n non-empty relations $R_1, \ldots, R_n$ has to be computed. This is done by applying a binary cross product operator $\times$ to the relations and intermediate results. An expression that contains all relations exactly once can be depicted as a regular binary tree, where the intermediate nodes correspond to applications of the cross product operator. For example, consider four relations $R_1, \ldots, R_4$. Then, the expression $(R_1 \times R_2) \times (R_3 \times R_4)$ corresponds to the tree



For each relation R_i, we denote its size by $|R_i| = n_i$. As the cost function we count the number of tuples within the intermediate results. Assuming the relations' sizes $n_1 = 10$, $n_2 = 20$, $n_3 = 5$ and $n_4 = 30$, the cost of the above bushy tree would be $10 * 20 + 5 * 30 + (10 * 20 * 5 * 30) = 30350$. Since the final result is always the same for all bushy trees—the product of all relation sizes—we often neglect it.

First, we need the following lemma.

Lemma 4.1.1 *Let $R_1, \ldots, R_n$ be relations with their according sizes. If $|R_n| > \prod_{i=1,n-1} |R_i|$, then the optimal bushy tree is of the form $X \times R_n$ or $R_n \times X$ where X is an optimal bushy tree containing relations $R_1, \ldots, R_{n-1}$.*

Proof The proof is by induction on n. The claim is trivially true for $n = 2$. Now, assume that the claim holds for any number of relations smaller than n, for some $n > 2$.
Consider the tree $T_1 = X \times R_n$, where X denotes an optimal tree for the set of relations $R_1, \ldots, R_{n-1}$. We assume that there exists an optimal tree T_2 for $R_1, \ldots, R_n$, which is not of the form $X \times R_n$ or $R_n \times X$, and derive a contradiction. Let $T_2 = Y \times Z$ with optimal subtrees Y and Z. Without loss of generality, assume that R_n occurs in Z. According to the induction hypothesis, $Z = W \times R_n$ or $Z = R_n \times W$, where W denotes an optimal subtree for the relations in Z without R_n. Assume that $Z = W \times R_n$ (the other case is analogous). Since X and $Y \times W$ contain the same relations and X is optimal, we have

$$\begin{aligned} cost(Y \times W) &= cost(Y) + cost(W) + |Y||W| \\ &\geq cost(X). \end{aligned} \tag{4.1}$$

Using $|T_1| = |T_2|$, $|R_n| > |Y||W| = |X|$, $|W| \geq 1$, $|R_n| \geq 1$, and (4.1), we have

$$\begin{aligned} cost(T_2) &= cost(Y \times (W \times R_n)) \\ &= cost(Y) + cost(W) + |R_n| + |W||R_n| + |T_2| \\ &\geq cost(X) - |Y||W| + |R_n| + |W||R_n| + |T_1| \\ &> cost(X) + |W||R_n| + |T_1| \\ &= cost(T_1) + (|W| - 1)|R_n| \\ &\geq cost(T_1). \end{aligned}$$

This contradicts the assumed optimality of T_2, and the claim follows. □

Definition 4.1.1 *(cross product optimization, XR)*
The problem of constructing minimal cost bushy trees for taking the cross product of n relations is denoted by XR*.*

In order to prove that *XR* is NP-hard, we need another problem known to be NP-complete for which we can give a polynomial time Turing reduction to *XR*. We have chosen to take the exact cover with 3-sets (*X3C*) as the problem of choice. The next definition recalls this problem which is known to be NP-complete [GJ79].

Definition 4.1.2 *(exact covering with 3-sets, X3C)*
Let S be a set with $|S| = 3q$ elements. Further let C be a collection of subsets of S containing three elements each. The following decision problem is called X3C*: Does there exist a subset C' of C such that every $s \in S$ occurs exactly once in C'?*

We are now prepared to prove our claim.

Theorem 4.1.1 *The problem* XR *is NP-hard.*

In order to understand the proof, it might be necessary to state the underlying idea explicitly. An optimal bushy tree for a set of relations is as balanced as possible. That is, a bushy tree $(T_1 \times T_2) \times (T_3 \times T_4)$ with subtrees T_i is optimal, if $\text{abs}(|T_1 \times T_2| - |T_3 \times T_4|)$ is minimal and cannot be reduced by exchanging relations from the left to the right subtree. This is not always true, since a left-deep tree can be cheaper even if this criterion is not fulfilled. In order to see this, consider the following counterexample. Let $R_1, \ldots, R_4$ be four relations with sizes $n_1 = 2$, $n_2 = 3$, $n_3 = 4$, and $n_4 = 10$. The optimal "real" bushy tree is $(R_1 \times R_4) \times (R_2 \times R_3)$ with cost $2 * 10 + 3 * 4 = 32$. Its top-level difference is $20 - 12 = 8$. But the left-deep tree $((R_1 \times R_2) \times R_3) \times R_4$ has lower cost $2 * 3 + 2 * 3 * 4 = 30$ although it has a higher top-level difference $24 - 10 = 14$. Considering our lemma, it becomes clear that it is a good idea to add some big relations at the top to fix the shape of an optimal tree. Further, these additional relations (named T and D in the following proof) are needed to guarantee the existence of a fully balanced and optimal tree.

Proof We prove the claim by reducing *X3C* to *XR*. Let (S, C) with $|S| = 3q$ be an instance of *X3C*. Without loss of generality, we assume that $|C| > q$. Obviously, if $|C| < q$ there exists no solution. If $|C| = q$, the problem can be decided in polynomial time.

We start by coding $X3C$. First, we map every element of S to an odd prime. Let $S = \{s_1, \ldots, s_{3q}\}$, then s_i is mapped to the i-th odd prime. Note that we can apply a sieve method to compute these primes in polynomial time. Subsequently, we identify s_i and p_i.

Every element $c = \{s_{i_1}, s_{i_2}, s_{i_3}\} \in C$ is mapped to the product $s_{i_1} * s_{i_2} * s_{i_3}$. Again, we identify c with its product.

Note that this coding allows to identify uniquely the s_i and c. Each c will now become a relation R of size c. In addition, we need two further relations T and D. We denote their sizes by T and D as well. This overloading can always be resolved by the context. The sizes T and D are defined via some more numbers:

$$\begin{aligned} S &:= \prod_{s \in S} s \\ C &:= \prod_{c \in C} \prod_{c' \in c} c' \\ H &:= \text{lcm}\,(S, (C/S)) \\ K &:= 2C^2 \\ T &:= (H/S)K \\ D &:= ((HS)/C)K \end{aligned}$$

where $lcm(x, y)$ denotes the least common multiple of the numbers x and y.

Without loss of generality, we assume that

$$C \equiv 0 \pmod{S}.$$

If this is not the case, obviously no solution for *X3C* exists.

We will now show that

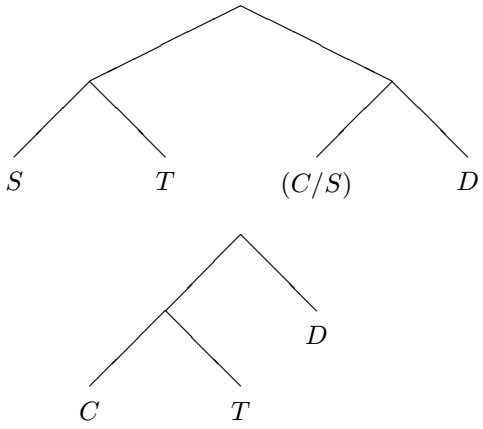


Figure 4.1: The first case

> *there exists a solution for X3C if and only if the optimal solution has the form* $(A \times T) \times (B \times D)$ *where* A *and* B *are subtrees and* T *and* D *are the special relations from above. Further,* $|A| = S$ *and* $|B| = (C/S)$ *must hold.*

Of course, the above must be seen with respect to possible interchanges of sibling subtrees which does not result in any cost changes.

Clearly, if there is no solution for the X3C problem, no bushy tree with these properties exists. Hence, it remains to prove that, if X3C has a solution, then the above bushy tree is optimal.

Within the following trees, we use the sizes of the intermediate nodes or relation sizes to denote the corresponding subtrees and relations. To proof the remaining claim, we distinguish three cases. Within the first case, we compare our (to be shown) optimal tree with two left-deep trees. Then, we consider the case where both T and D occur in either the left or the right part of a bushy tree. Last, we assume one part contains T and the other part contains D.

For the first case, the top tree of the figure 4.1 must be cheaper than the bottom left-deep tree.

As mentioned, the tree shows only the sizes of the missing subtrees. If some $C' \subseteq C$ is a solution for *X3C*, then it must have a total size $C' = S$.

Note that we need not to consider any other left-deep trees except where T and D are exchanged. This is due to the fact that the sizes of these relations exceed the C by far. (Compare with the above lemma.)

The following is a sequence of inequalities which hold also if T and D are exchanged in the bottom tree of figure 4.1. We have to proof that

$$\begin{aligned} ST + (C/S)D + cost(C') + cost(C \setminus C') \\ < C \min(D, T) + cost(C) \end{aligned}$$

Obviously,

$$cost(C) \quad \geq C$$

and

$$cost(C') \leq (C/2), \qquad cost(C \setminus C') \leq (C/2)$$

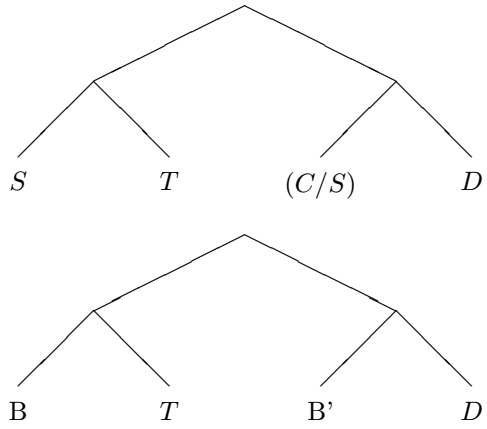


Figure 4.2: The last case

Hence,

$$cost(C') + cost(C \setminus C') \quad < \quad cost(C)$$

Further,

$$\begin{array}{rrr} & 2HK & < & C(H/S)K \\ \Longleftarrow & & & \\ & 2 & < & (C/S) \end{array}$$

and

$$\begin{array}{rrr} & 2HK & < & C((HS)/C)K \\ \Longleftarrow & & & \\ & 2 & < & S \end{array}$$

also hold. This completes the first case.

In order to follow the inequalities, note that the cost of computing a bushy tree never exceeds twice the size of its outcome if the relation sizes are greater than two, which is the case here.

If we assume T and D to be contained in either the right or the left subtree, we get the following cost estimations:

$$\begin{array}{rrrr} & ST + (C/S)D + cost(C') + cost(C \setminus C') & < & TD \\ \Longleftarrow & & & \\ & 2HK + C & < & ((HK)/C)HK \\ \Longleftarrow & & & \\ & 2 + (C/HK) & < & (1/C)HK \\ \Longleftarrow & & & \\ & 1 + (1/(HC)) & < & HC \end{array}$$

Again, the last inequality is obvious. This completes the second case.

Now consider the last case, where T and D occur in different subtrees. It is shown in Figure 4.2. It has to be proven that the upper processing tree is cheaper than the lower one.

Denote the size of the result of B by B and the size of the result of B' by B'. Further, note that B and B' arise from S and (C/S) by exchanging relations within the latter two. This gives

us

$$
\begin{array}{c}
2HK + cost(C') + cost(C \setminus C') \\
< BT + B'D + cost(B) + cost(B') \\
\Longleftarrow \\
2HK + cost(C') + cost(C \setminus C') \\
< BT + B'D \\
\Longleftarrow \\
2HK + C \\
< bST + (1/b)D(C/S) \\
\Longleftarrow \\
2HK + C \\
< (b + (1/b))HK
\end{array}
$$

where b is (B/S).

Since all relation sizes are odd primes, and we assume that the right tree is different from our optimal tree, S and B must differ by at least 2. Hence, either $b \geq (S+2/S)$ or $0 < b \leq (S/S+2)$. Since the function $f(x) = x + (1/x)$ has exactly one minimum at $x = 1$, and is monotonously decreasing to the left of $x = 1$ and monotonously increasing to the right of $x = 1$, we have:

$$
\begin{array}{rcl}
C & < & (((S+2)/2) + (S/(S+2)) - 2)\, HK \\
\Longleftarrow & & \\
4C & < & (((S+2)^2 + S^2 - 2S(S+2))/(S(S+2)))HK \\
\Longleftarrow & & \\
4C & < & ((S^2 + 4S + 4 + S^2 - 2S^2 - 4S)/(S(S+2)))HK \\
\Longleftarrow & & \\
4C & < & (4/(S(S+2)))HK \\
\Longleftarrow & & \\
C & < & (2/(S(S+2)))HC^2 \\
\Longleftarrow & & \\
1 & < & (2/(S(S+2)))HC
\end{array}
$$

The last inequality holds since $H \geq S$ and $C \geq S + 2$. This completes the proof.

□

The next corollary follows immediately from the theorem.

Corollary 4.1.1 *Constructing optimal bushy trees for a given join graph is —independent of its form—NP-hard.*

Whereas taking the cross product of vast amounts of relations is not the most serious practical problem, joining a high number of relations is a problem in many applications. This corollary unfortunately indicates that there is no hope of finding a polynomial algorithm to solve this problem. A similar discouraging result exists for the problem of generating optimal left-deep trees that possibly contain cross products for acyclic queries. Even if the height of the query graph (now a tree) is restricted to 1, the problem is NP-hard. The only remaining niche where we might find polynomial algorithms is when query graphs are restricted to chains. These have been considered in the section 3.1.

4.2 General Queries with Joins, Cross Products and Selections

In this section we focus on the development of new, particularly efficient dynamic programming algorithms for the general problem of computing optimal bushy trees with cross products for queries containing joins, cross products, and expensive selections. In section 4.2.1 we present a dynamic programming algorithm for the basic execution space. Section 4.2.2 discusses some problems to be solved in an efficient implementation of the algorithm. We discuss possibilities for fast enumeration of subproblems, fast computation of cost functions, and for saving space. In section 4.2.3 a dynamic programming algorithm for an enlarged execution space is presented. The algorithm accounts for the possibility to split conjunctive predicates. Section 4.2.4 presents a second dynamic programming algorithm for the enlarged execution space that makes use of structural information form the join graph in order to speed up the computation. In section 4.2.5 we analyze the time and space complexities of the algorithm. Section 4.2.6 describes how our algorithms can be enhanced to handle several join algorithms including sort merge join with a correct handling of interesting orders, affine join cost functions and query hypergraphs. Section 4.2.7 shows the results of timing measurements. Variants of the dynamic programming scheme are discussed in section 4.2.8.

4.2.1 A Dynamic Programming Algorithm for the Basic Execution Space

Let us denote the set of relations occurring in a bushy plan P by $Rel(P)$ and the set of relations to which selections in P refer by $Sel(P)$. Let R denote a set of relations. We denote by $Sel(R)$ the set of all selections referring to some relation in R. Each subset $V \subseteq R$ defines an *induced subquery* which contains all the joins and selections that refer to relations in V only. A *subplan* P' of a plan P corresponds to a subtree of the expression tree associated with P. A *(bi)partition* of a set S is a pair of non-empty disjoint subsets of S whose union is exactly S. For a partition S_1, S_2 of S we write $S = S_1 \uplus S_2$. S_1 and S_2 are the *blocks* of the partition. A partition is *non-trivial* if neither S_1 nor S_2 is empty. By a *k-set* we simply mean a set with exactly k elements.

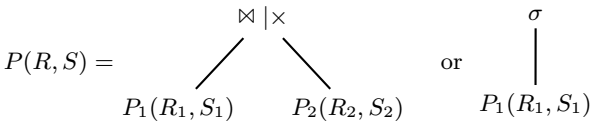


Figure 4.3: Structure of an optimal plan

Consider an optimal plan P for an induced subquery involving the non-empty set of relations $Rel(P)$ and the set of selections $Sel(P)$. Obviously, P has either the form $P \equiv (P_1 \bowtie P_2)$ for subplans P_1 and P_2 of P, or the form $P \equiv \sigma_i(P')$ for a subplan P' of P and a selection $\sigma_i \in Sel(P)$. The important fact is now that the subplans P_1, P_2 are necessarily *optimal plans* for the relations $Rel(P_1), Rel(P_2)$ and the selections $Sel(P_1), Sel(P_2)$, where $Rel(P_1) \uplus Rel(P_2) = Rel(P)$, $Sel(P_1) = Sel(P) \cap Sel(R_1)$, $Sel(P_2) = Sel(P) \cap Sel(R_2)$. Similarly, P' is an optimal bushy plan for the relations $Rel(P')$ and the selections $Sel(P)$, where $Rel(P') = Rel(P)$, $Sel(P') = Sel(P) - \{\sigma_i\}$. Otherwise we could obtain a cheaper plan by replacing the suboptimal part by an optimal one which would be a contradiction to the assumed optimality of P (note that our cost function is decomposable and monotone). The property that optimal solutions of a problem can be decomposed into a number of "smaller", likewise optimal solutions of the same problem, is known as *Bellman's optimality principle*. This leads immediately to the following recurrence for computing an optimal bushy plan[1] for a set of relations R and a set of selections S.

[1] min() is the operation which yields a plan with minimal costs among the addressed set of plans. Convention: $\min_{\emptyset}(\dots) := \lambda$ where λ denotes some artificial plan with cost ∞.

$$\mathrm{opt}(R,S) \quad = \quad \begin{cases} \min(\min_{\emptyset \subset R' \subset R}(\mathrm{opt}(R', S \cap \mathrm{Sel}(R')) \bowtie & \text{if } \emptyset \subseteq S \subseteq R, \\ \qquad \mathrm{opt}(R \setminus R', S \cap \mathrm{Sel}(R \setminus R'))) & \\ \qquad \min_{\sigma_i \in S}(\sigma_i(\mathrm{opt}(R, S \setminus \{\sigma_i\})))) & \\ R_i & \text{if } R = \{R_i\}, \\ & \quad S = \emptyset \end{cases} \tag{4.2}$$

The join symbol $\bowtie$ denotes a join *with the conjunction of all join predicates* that relate relations in R' to relations in $R \setminus R'$. Considering the join graph, the conjuncts of the join predicate correspond to the predicates associated with the edges in the cut $(R', R \setminus R')$. If the cut is empty the join is actually a *cross product*.
In our first algorithm we will treat such joins and selections with conjunctive predicates as single operations with according accumulated costs. The option to split such predicates will be discussed in section 4.2.3 where we present a second algorithm.

Now let us estimate the costs c_p of evaluating such a conjunctive join or selection predicate $p = p_1 \wedge p_2 \wedge \cdots \wedge p_k$, where $p_1, \ldots, p_k$ are *basic* predicates relating to the same relations. A naive approach would be to test each of the predicates in some fixed order. This would raise the costs $c_p = c_{p_1} + \cdots + c_{p_k}$—independent of the testing order. But we can do better if we make use of the predicate selectivities. If we test in the order given by a permutation π, the expected costs are

$$c_p(\pi) = \sum_{i=1}^{k} \prod_{j=1}^{i-1} s_{p_{\pi(j)}} c_{p_{\pi(i)}} \tag{4.3}$$

The problem of minimizing this cost function is equivalent to the *least cost fault detection problem* [MS79]. Since $c_p(\pi)$ exhibits the so-called *adjacent pairwise interchange (API) property* [Smi56, MS79], an optimal permutation is characterized by the condition

$$r(\pi(1)) \leq r(\pi(2)) \leq \cdots \leq r(\pi(k)) \tag{4.4}$$

with the *rank function* $r(i) := c_{p_i}/(1 - s_{p_i})$.
The costs c_p of evaluating the conjunctive predicate p are now defined by equation (4.3), where the permutation π is determined by condition (4.4).

Based on recurrence (4.2), there is an obvious recursive algorithm to solve our problem but this solution would be very inefficient since many subproblems are solved more than once. A much more efficient way to solve this recurrence is by means of a table and is known under the name of *dynamic programming*[2] [Min86, CLR90]. Instead of solving subproblems recursively, we solve them one after the other in some appropriate order and store their solutions in a table. The overall time complexity then becomes (typically) a function of the number of distinct subproblems rather than of the larger number of recursive calls. Obviously, the subproblems have to be solved in the right order so that whenever the solution to a subproblem is needed it is already available in the table. A straightforward solution is the following. We enumerate all subsets of relations by increasing size, and for each subset R we then enumerate all subsets S of the set of selections occurring in R by increasing size. For each such pair (R, S) we evaluate the recurrence (4.2) and store the solution associated with (R, S).

For the algorithm in Figure 4.4 we assume a given select-join-query involving n relations $\mathcal{R} = \{R_1, \ldots, R_n\}$ and $m \leq n$ selections $\mathcal{S} = \{\sigma_1, \ldots, \sigma_m\}$. In the following, we identify selections and relations to which they refer. Let P be the set of all join predicates $p_{i,j}$ relating two relations R_i and R_j. By R_S we denote the set $\{R_i \in \mathcal{R} \mid \exists \sigma_j \in \mathcal{S} : \sigma_j \text{ refers to } R_i\}$ which consists of all relations in $\mathcal{R}$ to which some selection in $\mathcal{S}$ relates. For all $U \subseteq \mathcal{R}$ and $V \subseteq U \cap R_S$, at the end of the algorithm $T[U, V]$ stores an optimal bushy plan for the subquery (U, V).

[2] the name derives from the fact that the method has its origins in the optimization of dynamic systems (systems that evolve over time) [Min86]

Algorithm Optimal-Bushy-Tree(R, P)

```
 1 for k = 1 to n do
 2     for all k-subsets M_k of R do
 3         for l = 0 to min(k, m) do
 4             for all l-subsets P_l of M_k ∩ R_S do
 5                 best_cost_so_far = ∞;
 6                 for all subsets L of M_k with 0 < |L| < k do
 7                     L' = M_k \ L, V = P_l ∩ L, V' = P_l ∩ L';
 8                     p = ⋀{p_{i,j} | p_{i,j} ∈ P, R_i ∈ V, R_j ∈ V'};    // p=true might hold
 9                     T = (T[L, V] ⋈_p T[L', V']);
10                     if Cost(T) < best_cost_so_far then
11                         best_cost_so_far = Cost(T);
12                         T[M_k, P_l] = T;
13                     fi;
14                 od;
15                 for all R ∈ P_l do
16                     T = σ_R(T[M_k, P_l \ {R}]);
17                     if Cost(T) < best_cost_so_far then
18                         best_cost_so_far = Cost(T);
19                         T[M_k, P_l] = T;
20                     fi;
21                 od;
22             od;
23         od;
24     od;
25 od;
26 return T[R, S];
```

Figure 4.4: Algorithm Optimal-Bushy-Tree(R,P)

Complexity of the algorithm: As the complexity yardstick we take the number of considered partial plans which equals the number of times the innermost loops are executed. To count the number of times the loops are executed we first have to determine the number of l-subsets P_l of $M_k \cap R_S$. Recall that M_k has exactly k elements and R_S has exactly m elements. Hence, from the $\binom{n}{k}$ subsets M_k of R, there are $\binom{m}{l}\binom{n-m}{k-l}$ subsets P_l with $|P_l \cap R_S| = l$ (we can choose l relations from R_S and the remaining $k-l$ relations from $R \setminus R_S$). Hence, the number of considered partial plans is

$$\underbrace{\sum_{k=1}^{n}}_{\text{line 1}} \underbrace{\sum_{l=0}^{\min(k,m)} \binom{m}{l}\binom{n-m}{k-l}}_{\text{line 2,3}} \underbrace{\left[\sum_{i=0}^{l}\binom{l}{i}\right]}_{\text{line 4}} (\underbrace{2^k-2}_{\text{line 6}} + \underbrace{l}_{\text{line 15}})$$

$$= 1 + \sum_{k=0}^{n}\sum_{l=0}^{\min(k,m)} \binom{m}{l}\binom{n-m}{k-l} 2^l[2^k - 2 + l] \qquad \text{due to (4.5)}$$

$$= 1 + \sum_{l=0}^{m}\sum_{k=l}^{n} \binom{m}{l}\binom{n-m}{k-l} 2^l[2^k - 2 + l] \qquad (4.8)$$

$$
\begin{aligned}
&= 1 + \sum_{l=0}^{m} \binom{m}{l} 2^l \sum_{k=0}^{n-l} \binom{n-m}{k} 2^{k+l} - 2 \sum_{l=0}^{m} \binom{m}{l} 2^l \sum_{k=0}^{n-l} \binom{n-m}{k} \\
&\quad + \sum_{l=0}^{m} \binom{m}{l} l 2^l \sum_{k=0}^{n-l} \binom{n-m}{k} && (4.6) \\
&= 1 + \sum_{l=0}^{m} \binom{m}{l} 4^l \sum_{k=0}^{n-m} \binom{n-m}{k} 2^k - 2 \sum_{l=0}^{m} \binom{m}{l} 2^l \sum_{k=0}^{n-m} \binom{n-m}{k} \\
&\quad + 2m \sum_{l=0}^{m-1} \binom{m-1}{l} 2^l \sum_{k=0}^{n-m} \binom{n-m}{k} && (4.7), (4.9) \\
&= 1 + \sum_{l=0}^{m} \binom{m}{l} 4^l 3^{n-m} - 2 \sum_{l=0}^{m} \binom{m}{l} 2^l 2^{n-m} + 2m \sum_{l=0}^{m-1} \binom{m-1}{l} 2^l 2^{n-m} && (4.5) \\
&= 5^m 3^{n-m} - 2\, 3^m 2^{n-m} + 2\, m\, 3^{m-1} 2^{n-m} + 1 \\
&= 3^n \left(\frac{5}{3}\right)^m + (2m/3 - 2) \cdot 2^n \left(\frac{3}{2}\right)^m + 1
\end{aligned}
$$

where we used the identities

$$
\begin{aligned}
\sum_k \binom{n}{k} x^k &= (x+1)^n \qquad \text{(binomial theorem—special case)} && (4.5) \\
\sum_{k=l}^{n} a_k &= \sum_{k=0}^{n-l} a_{k+l} && (4.6) \\
l\binom{m}{l} &= m\binom{m-1}{l-1} && (4.7) \\
\sum_{k=0}^{n} \sum_{l=0}^{\min(k,m)} a_{k,l} &= \sum_{l=0}^{m} \sum_{k=l}^{n} a_{k,l} && (4.8)
\end{aligned}
$$

and basic facts about finite sums (cf. [GKP89]). Furthermore, note that

$$
\binom{n}{k} = 0 \quad \text{for} \quad k > n \geq 0 \qquad (4.9)
$$

Since the complexity derivations in the following sections are quite similar we will present them with fewer intermediate steps and comments.

Let $c := m/n$ be the ratio of selections to the total number of involved relations. Then, we can express the number of partial plans as

$$
\left\lceil 3 \left(\frac{5}{3}\right)^c \right\rceil^n + (2cn/3 - 2) \left\lceil 2 \left(\frac{3}{2}\right)^c \right\rceil^n + 1
$$

Assuming an asymptotic optimal implementation of the enumeration part of the algorithm (see section 4.2.2), the amount of work per considered plan is constant and the asymptotic time complexity of our algorithm is

$$
O(\lceil 3\,(5/3)^c \rceil^n + n \lceil 2\,(3/2)^c \rceil^n)
$$

For the special cases $m = 0$ and $m = n$, this evaluates to $O(3^n + n2^n)$ and $O(5^n + n3^n)$, respectively.

Next we determine the space complexity of our algorithm. The number of table entries used by the algorithm to store the solutions of subproblems is exactly

$$\begin{aligned}
&\sum_{k=1}^{n}\sum_{l=0}^{\min(k,m)}\binom{m}{l}\binom{n-m}{k-l}\left[\sum_{i=0}^{l}\binom{l}{i}\right] \\
&= 1+\sum_{l=0}^{m}\sum_{k=l}^{n}\binom{m}{l}\binom{n-m}{k-l}2^l \\
&= 1+\sum_{l=0}^{m}\binom{m}{l}2^l\sum_{k=0}^{n-l}\binom{n-m}{k} \\
&= 1+\sum_{l=0}^{m}\binom{m}{l}2^l\sum_{k=0}^{n-m}\binom{n-m}{k} \\
&= 1+\sum_{l=0}^{m}\binom{m}{l}2^l 2^{n-m} \\
&= 1+3^m 2^{n-m} = 1+2^n(3/2)^m
\end{aligned}$$

For $c = m/n$ we have

$$1+[2(3/2)^c]^n$$

Note that this evaluates to $1+2^n$ and $1+3^n$ for the special cases $m=0$ and $m=n$, respectively.

Unfortunately, coding a pair of subsets $(R,P),\ P\subseteq R\subseteq\mathcal{M}$ by two separate bit vectors r and p both ranging from 0 to $2^{|\mathcal{M}|}-1$ is *not compact* since it obviously uses space $2^n\cdot 2^n$. Hence, the space complexity of the above algorithm would be $O(4^n)$ instead of $O(2^{n-m}3^m)$. An obvious improvement is to renumber the relations such that all relations to which a selection is applied *precede* all relations to which no selections are applied. Hence, we can represent subsets of selections by bit vectors $i,\ 0\le i<2^m$. This reduces the space complexity to $O(2^{n+m})$ which still wastes considerable space. In section 4.2.2 we describe how to store the tables in a more compact way.

4.2.2 An Efficient Implementation Using Bit Vectors

Fast Enumeration of Subproblems

The frame of our dynamic programming algorithm is the systematic enumeration of subproblems consisting of three nested loops iterating over subsets of relations and predicates, respectively.

The first loop enumerates all non-empty subsets of the set of all relations in the query. It turns out that enumerating all subsets strictly by increasing size seems not to be the most efficient way. The whole point is that the order of enumeration only has to guarantee that for every enumerated set S, all subsets of S have already been enumerated. One of such orderings, which is probably the most suitable, is the following. We use the standard representation of n-sets, namely bit vectors of length n. A subset of a set S is then characterized by a bit vector which is component-wise smaller than the bit vector of S. This leads to the obvious ordering in which the bit vectors are arranged according to their value as binary numbers. This simple and very effective enumeration scheme (*binary counting method*) is successfully used in [VM96]. The major advantage is that we can pass over to the next subset by merely incrementing an integer, which is an extremely fast hardwired operation.

Example
Enumeration of the subsets of $\{R_0, R_1, R_2, R_3, R_4\}$ in subset order (binary counting method).

11111	$\{R_0, R_1, R_2, R_3, R_4\}$
00000	$\{\}$
00001	$\{R_0\}$
00010	$\{R_1\}$
00011	$\{R_0, R_1\}$
00100	$\{R_2\}$
00101	$\{R_0, R_2\}$
⋮	
11111	$\{R_0, R_1, R_2, R_3, R_4\}$

□

The next problem is to enumerate subsets S of a fixed subset M of a set Q. If Q has n elements then we can represent M by a bit vector m of length n. Since M is a subset of Q some bit positions of m may be zero. Vance and Maier propose in [VM96] a very efficient and elegant way to solve this problem. In fact, they show that the following loop enumerates all bit vectors S being a subset of M, where $M \subseteq Q$.

$S \leftarrow 0$; // S, M, Q bit vectors
repeat
  ...
  $S \leftarrow M \,\&\, (S - M)$;
until $S = 0$

We assume two's-complement arithmetic. Bit operations are denoted as in the language C. As an important special case, we mention that

$$M \,\&\, -M$$

yields the bit with the *smallest* index in M. Similarly, one can count downward using the operation $S \leftarrow M \,\&\, (S - 1)$ instead of $S \leftarrow M \,\&\, (S - M)$.

Example
Enumeration of the subsets of $\{R_1, R_3\} \subseteq \{R_0, R_1, R_2, R_3, R_4\}$.

01010	$\{R_1, R_3\}$
00000	$\{\}$
00010	$\{R_1\}$
01000	$\{R_3\}$
01010	$\{R_1, R_3\}$

□

Example
Iteration through the elements of the set $\{R_1, R_3, R_4\}$.

11010	$\{R_1, R_3, R_4\}$
00010	$\{R_1\}$
01000	$\{R_3\}$
10000	$\{R_4\}$

□

Combining the operations above, one can show that $S = M \,\&\, ((S \,|\, (M \,\&\, (S-1))) - M)$ (initial value $S = M \,\&\, -M$) iterates through each bit in the bit vector M in order of increasing indices. Nevertheless, the following method is both simpler and faster. R holds the bits that are still to be enumerated. While R is not zero L is set to the lowest order non-zero bit in R. After L has been used the lowest order bit is deleted from R and the loop starts again.

$R \leftarrow S$;
while $R \neq 0$ **do**
  $L \leftarrow R \,\&\, -R$;
  ...
  $R \leftarrow R \wedge L$;
od

Efficient Computation of the Cost Function

Now, we discuss the efficient evaluation of the cost function within the nested loops. Obviously, for a given plan we can compute the costs and the size in (typically) linear time but there is a even more efficient way using the recurrences for these functions. If R', R'' and S', S'' are the partitions of R and S, respectively, for which the recurrence (4.2) assumes a minimum, we have

$$Size(R,S) \quad = \quad Size(R',S') * Size(R'',S'') * Sel(R',R''),$$

where

$$Sel(R',R'') := \prod_{R_i \in R', R_j \in R''} f_{i,j}$$

is the product of all selectivities between relations in R' and R''. Note that the last equation holds for *every* partition R', R'' of R independent of the root operator in an optimal plan. Hence we may choose a certain partition in order to simplify the computations. Now if $R' = U_1 \uplus U_2$ and $R'' = V_1 \uplus V_2$, we have the following recurrence for $Sel(R', R'')$

$$Sel(U_1 \uplus U_2, V_1 \uplus V_2) = Sel(U_1,V_1) * Sel(U_1,V_2) * Sel(U_2,V_1) * Sel(U_2,V_2)$$

Choosing $U_1 = \alpha(R)$, $U_2 := \emptyset$, $V_1 = \alpha(R \setminus U_1)$, and $V_2 := R \setminus U_1 \setminus V_2$, where the function α is given by $\alpha(A) := \{R_k\},\ k = \min\{i \mid R_i \in A\}$
leads to

$$\begin{aligned} &Sel(\alpha(R), R \setminus \alpha(R)) = \\ &\quad Sel(\alpha(R), \alpha(R \setminus \alpha(R))) * Sel(\alpha(R), (R \setminus \alpha(R)) \setminus \alpha(R \setminus \alpha(R))) = \\ &\quad Sel(\alpha(R), \alpha(R \setminus \alpha(R))) * Sel(\alpha(R), (R \setminus \alpha(R \setminus \alpha(R))) \setminus \alpha(R)) \end{aligned}$$

Defining the *fan-selectivity* $Fan\_Sel(R)$ as $Sel(\alpha(R), R \setminus \alpha(R))$, gives the simpler recurrence

$$\begin{aligned} Fan\_Sel(R) \quad &= \quad Sel(\alpha(R), \alpha(R \setminus \alpha(R))) * Fan\_Sel(R \setminus \alpha(R)) \\ &= \quad f_{i,j} * Fan\_Sel(R \setminus \alpha(R \setminus \alpha(R))) \end{aligned} \tag{4.10}$$

where we assumed $\alpha(R) = \{R_i\}$ and $\alpha(R \setminus \alpha(R)) = \{R_j\}$.

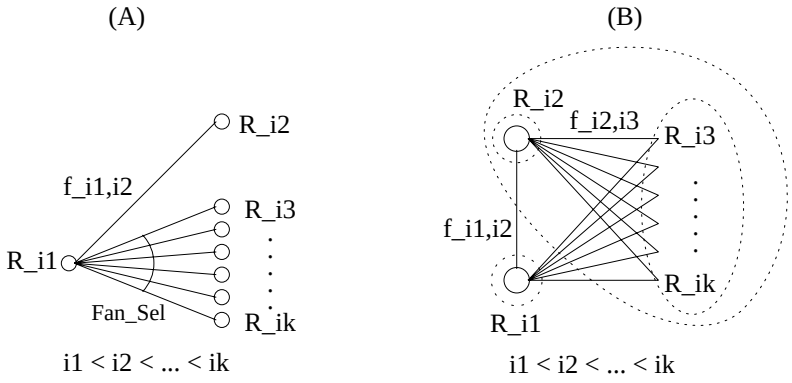


Figure 4.5: Recurrences for computing fan-selectivity (A) and size (B)

As a consequence, we can compute $Size(R,S)$ with the following recurrence

$$\begin{aligned} Size(R,S) \quad &= \quad Size(\alpha(R), S \cap \alpha(R)) * Size(R \setminus \alpha(R), (R \setminus \alpha(R)) \cap S) \\ &\quad * Fan\_Sel(R) \end{aligned} \tag{4.11}$$

We remind that the single relation in $\alpha(R)$ can be computed very efficiently via the operation $a \,\&\, -a$ on the bit vector of R. Recurrences (4.10) and (4.11) are illustrated in Figure 4.5. The encircled sets of relations denote the nested partitions along which the sizes and selectivities are computed. Next we state a pseudocode version of our first algorithm.

Program DP-OPT-A

```
 1 Dynamically allocate array table with dimensions 2^n by 2^m
   table[i][j] is a record with the following attributes:
   cost: cost of an optimal plan with relations i and selections j
   size: size of a plan with relations i and selections j
   type: type of the root operator in an optimal plan (⋈, × or σ)
   best: bit vector of relations in the left subplan (if type ∈ {'⋈','×'})
       or bit vector of a selection (if type ='σ')

 2 sels ← (1 ≪ m) − 1;
 3 for i ← 0 to n − 1 do                          // initialize tables
 4     a ← 1 ≪ i;
 5     table[a, 0].cost ← 0.0;
 6     table[a, 0].size ← n_i;
 7     table[a, 0].fan_sels ← 1.0;
 8     if (a & sels = a) then
 9         table[a, a].fan_sels ← f_i;
10         table[a, a].size ← f_i * n_i;
11     fi;
12     for j ← i + 1 to n − 1 do
13         b ← 1 ≪ j; c ← a | b;
14         table[c, 0].fan_sels ← f_{i,j};
15         if (a & sels = a) then
16             table[c, a].fan_sels ← f_{i,j} * f_i;
17         fi;
18         if (b & sels = b) then
19             table[c, b].fan_sels ← f_{i,j} * f_j;
20         fi;
21         if (c & sels = c) then
22             table[c, c].fan_sels ← f_{i,j} * f_i * f_j;
23         fi;
24     od;
25 od;
26 for m ← 1 to (1 ≪ n) − 1 do                    // enumerate subsets of relations
27     m_1 ← m & sels;
28     s ← 0;
29     repeat                                    // enumerate subsets of selections
30         a_1 ← m & − m;                        // relation with smallest index in m
31         b_1 ← m ∧ a_1;                        // all relations in m except relation with smallest index
32         c ← b_1 & − b_1;                      // relation with second smallest index in m
33         c_1 ← a_1 | c;                        // relations with two smallest indices in m
34         d ← b_1 ∧ c;                          // all relations in m except rel. with two smallest indices
35         d_1 ← a_1 | d;                        // all relations in m except rel. with second smallest index
36         d_2 ← d_1 & s;                        // corresponding selections
37         // compute sizes
38         if b_1 ≠ 0 then                       // is there more than one relation in m?
39             if d ≠ 0 then                     // are there more than two relations in m?
40                 table[m, s].fan_sels ← table[c_1, c_2].fan_sels * table[d_1, d_2].fan_sels;
41             fi;
42         else
```

```
43            table[m,s].size ← table[a1,0].size * table[b1,0].size * table[m,s].fan_sels;
44        fi;
45        // consider possible joins
46        l ← m & −m;                    // initial left block of partition
47        best_cost_so_far ← ∞;
48        while l < m do                 // enumerate partitions of relations
49            l' ← m ∧ l;                // the right block of partition
50            u ← l & s;                 // selections in the left block
51            v ← l' & s;                // selections in the right block
52            cost' ← table[l,u].cost + table[l',v].cost;                    // cost of subtree
53            cost ← cost' + g_jn(table[l,u].size, table[l',v].size, table[m,s].size);   // join costs
54            if cost < best_cost_so_far then                               // new plan cheaper?
55                best_cost_so_far ← cost;                                  // update best plan
56                table[m,s].best ← l;
57                table[m,s].type ← '⋈';
58            fi;
59            l ← m & (l − m);           // next subset
60        od;
61        u ← s;
62        // consider possible selections
63        while u ≠ 0 do                 // iterate through applicable selections
64            r ← u & −u;
65            v ← s ∧ r;                 // the remaining selections in s
66            cost ← table[m,v].cost + g_s(table[m,v].size, table[m,s].size);   // selection costs
67            if cost < best_cost_so_far then          // new plan cheaper?
68                best_cost_so_far ← cost;             // update best plan
69                table[m,s].best ← r;
70                table[m,s].type ← 'σ';
71            fi;
72            u ← u ∧ r;                 // next selection
73        od;
74        table[m,s].cost ← best_cost_so_far;          // store best plan
75        s ← m1 & (s − m1);             // next subset of selections
76    until s = 0;
77 od                                   // next subset of relations
```

Space Saving Measures

A problem is how to store the tables without wasting space. For example, suppose $n = 10$, $m = 10$ and let r and s denote the bit vectors corresponding to the sets R and S, respectively. If we use r and s directly as indices of a two-dimensional array $cost[\,][\,]$, about 90% of the entries in the table will not be accessed by the algorithm. To avoid this immense waste of space we have to use a *contracted version* of the second bit vector s.

Definition 4.2.1 *(bit vector contraction)*
Let r and s be two bit vectors consisting of the bits $r_0 r_1 \ldots r_n$ and $s_0 s_1 \ldots s_n$, respectively. We define the contraction of s with respect to r *as follows:*

$$contr_r(s) \quad = \quad \begin{cases} \epsilon & \text{if } s = \epsilon \\ contr_{r_1 \ldots r_n}(s_1 \ldots s_n) & \text{if } s \neq \epsilon \text{ and } r_0 = 0 \\ s_0\, contr_{r_1 \ldots r_n}(s_1 \ldots s_n) & \text{if } s \neq \epsilon \text{ and } r_0 = 1 \end{cases}$$

Example
Let us contract the bit vector $s = 0100010100$ with respect to the bit vector $r = 1110110100$. For each bit s_i of s, we examine the corresponding bit r_i in r. If $r_i = 0$ we "delete" s_i in s, otherwise we retain it. The result is the contracted bit vector 010011.

```
s:  0100010100
r:  1110110100
    ----------
    010 01 1     ⟶   010011
```

□

The following figure shows the structure of the two-dimensional ragged arrays. The first dimension is of fixed size 2^n, whereas the second dimension is of size 2^k where k is the number of common non-zero bits in the value of the first index i and the bit vector of all selections $sels$. The number of entries in such a ragged array is $\sum_{k=0}^{m} \binom{m}{k} 2^k 2^{n-m} = [2(\frac{3}{2})^c]^n$. Note that this number equals 2^n for $m = 0$ and 3^n for $m = n$. A simple array would require 4^n entries. Figure 4.6 shows the worst case where the number of selections equals the number of relations.

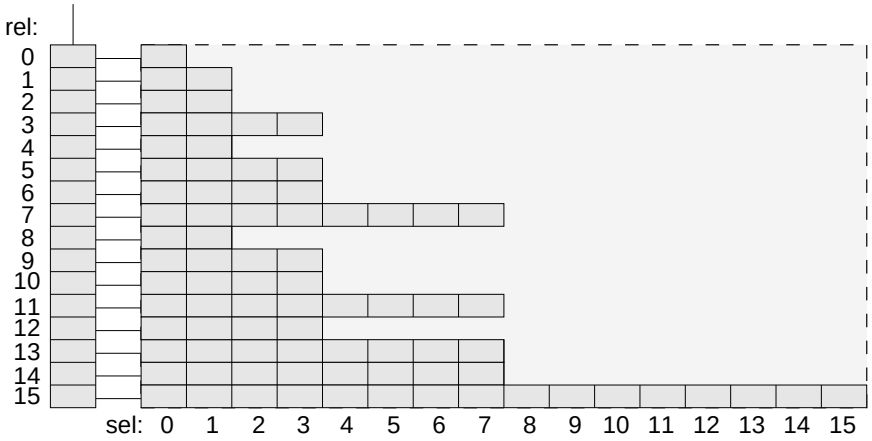


Figure 4.6: Using ragged arrays to reduce storage space

The simplest way would be to contract the second index on the fly for each access of the array. This would slow down the algorithm by a factor of m. It would be much better if we could contract all relevant values in the course of the other computations—therefore not changing the asymptotic time complexity. Note that within our algorithm the first and second indices are enumerated by increasing values which makes a contraction easy. We simply use pairs of indices i, ic, one uncontracted index i and its contracted version ic and count up their values independently.

As mentioned before, in the two outermost loops bit vector contraction is easy due to the order of enumeration of the bit vectors. Unfortunately, this does not work in the innermost loop where we have to contract the result of a conjunction of two contracted bit vectors again. Such a "double contraction" cannot be achieved by simple binary counting or a constant number of arithmetic and bit operations. Hence, there are two possibilities: either we compute the contractions on the fly or we tabulate the contraction function. The following code fragment computes a table for the contraction function.

```
sels = (1 ≪ m) − 1;
for i ← 0 to sels do
    ci ← sels ∧ i;              // complement
    j ← 0;
    jc ← 0;                     // contracted version of j
```

```
    repeat                          // j iterates through subsets of i
        k ← 0;
        repeat                      // k iterates through subsets of the complement of i
            contr[s, j | k] ← jc;   // store contraction of j | k with respect to s
            k ← ci & (k − ci);      // next subset k
        until k = 0;
        j ← i & (j − i);            // next subset j
        jc ← jc + 1;                // increment contracted bit vector
    until j = 0;
od
```

Since the nested loops enumerate all subsets of selections i together with all the subsets of i, tabulation requires $O(\sum_{i=0}^{m} \binom{m}{i} 2^i) = O(3^m)$ additional time and $O(2^m * 2^m) = O(4^m)$ additional space. Now consider the option of contracting bit vectors on the fly. The following code computes the contraction function.

```
Function contract(x, y)  // contract bit vector x with respect to bit vector y
    z ← 1;
    res ← 0;
    while y ≠ 0 do
        if (y & 1) ≠ 0 then
            if (x & 1) ≠ 0 then
                res ← res | z
            fi;
            z ← z ≪ 1;
        fi;
        x ← x ≫ 1;
        y ← y ≫ 1;
    od;
    return res;
end
```

Contraction on the fly increases the time complexity by a factor of m and does not increase the asymptotic space complexity. In the following we assume that bit vector contractions are computed on the fly by a call to the function `contract` above.

The pseudocode of our first algorithm using bit vector contraction is shown below.

Program DP-OPT-B

1 Dynamically allocate ragged array *table* with dimensions
$0 \le i < 2^n$ by $0 \le j < 2^{ones(i \,\&\, sels)}$ where ones(k) denotes the number
of non-zero bits in the binary representation of k.
table$[i][j]$ is a record with the following attributes:
cost: cost of optimal plan with relations i and selections j
size: size of a plan with relations i and selections j
type: type of the root operator in an optimal plan ($\bowtie$, $\times$ or σ)
best: bit vector of relations in the left subplan (if *type* $\in$ {'$\bowtie$','$\times$'})
or bit vector of a selection (if *type* ='σ')

```
2  nn ← (1 ≪ n) − 1;
3  sels ← (1 ≪ m) − 1;                    // assumption: selections refer to the first m relations
4  // initialize tables:
5  for i ← 1 to n − 1 do
6  // initialize tables for single relations
7      a ← 1 ≪ i;
8      table[a, 0].cost ← 0.0;
9      table[a, 0].size ← n_i;
10     table[a, 0].fan_sels ← 1.0;
11     if (a & sels = a) then
12         table[a, 1].fan_sels ← f_i;
13         table[a, 1].size ← f_i * n_i;
14     fi;
15         for j ← i + 1 to n − 1 do
16         // initialize tables for pairs of relations
17             b ← 1 ≪ j;
18             c ← a | b;
19             table[c, 0].fan_sels ← f_{i,j};
20             if (a & sels = a) then
21                 table[c, 1].fan_sels ← f_{i,j} * f_i;
22             fi;
23             if (b & sels = b) then
24                 table[c, 2].fan_sels ← f_{i,j} * f_j;
25             fi;
26             if (c & sels = c) then
27                 table[c, 3].fan_sels ← f_{i,j} * f_i * f_j;
28             fi;
29         od;
30     od;
31 // enumerate subproblems:
32 for m ← 1 to nn do                     // enumerate subsets of relations
33     m_1 ← m & sels;                    // applicable selections
34     s ← 0;
35     sc ← 0;                            // contracted version of s
36     repeat                             // enumerate subsets of applicable selections
37         a_1 ← m & − m;                 // relation with smallest index in m
38         b_1 ← m ∧ a_1;                 // all relations in m except relation with smallest index
39         c ← b_1 & − b_1;               // relation with second smallest index in m
40         c_1 ← a_1 | c;                 // relations with two smallest indices in m
41         cc_2 ← sc & 3;                 // contraction of bit vector c_1 & s
42         d ← b_1 ∧ c;                   // all rel. in m except rel. with two smallest indices
43         d_1 ← a_1 | d;                 // all rel. in m except rel. with second smallest index
44         dc_2 ← ((sc ≫ 2) ≪ 1) | (sc & 1);          // contraction of bit vector d_1 & s
45         if b_1 ≠ 0 then                // is there more than one relation in m?
46             if d ≠ 0 then              // are there more than two relations in m?
47                 table[m, sc].fan_sels ← table[c_1, cc_2].fan_sels * table[d_1, dc_2].fan_sels;
48             fi;
49             table[m, sc].size ← table[a_1, 0].size * table[b_1, 0].size * table[m, sc].fan_sels;
50         fi;
51         // consider possible joins:
52         l ← a_1;                       // initial left block of the partition
53         ll ← contr(s, s);              // note: contr(s, s) = 2^{ones(s)} − 1
54         best_cost_so_far ← ∞;
```

```
55        while l < m do              // enumerate partitions of relations
56            l' ← m ∧ l;             // right block of the partition
57            uc ← contract(l & sels, l & s);          // contract bit vectors u and v
58            vc ← ll ∧ uc;
59            cost' ← table[l, uc].cost + table[l', vc].cost;              // cost of subtree
60            cost ← cost' + g_jn(table[l, uc].size, table[l', vc].size, table[m, sc].size); // join costs
61            if cost < best_cost_so_far then                              // new plan cheaper?
62                best_cost_so_far ← cost;                                 // update best plan
63                table[m, sc].best ← l;
64                table[m, sc].type ← '⋈';
65            fi;
66            l ← m & (l − m);        // proceed to next subset
67        od;
68        // consider possible selections:
69        u ← s;
70        uc ← sc;
71        while u ≠ 0 do              // iterate through applicable selections
72            r ← u & − u;            // current selection
73            rc ← uc & − uc;         //contracted bit vector
74            v ← s ∧ r;              // the remaining selections in s
75            vc ← sc ∧ rc;           //contracted bit vector
76            cost ← table[m, vc].cost + g_s(table[m, vc].size, table[m, sc].size);     // sel. costs
77            if cost < best_cost_so_far then          // new plan cheaper?
78                best_cost_so_far ← cost;             // update best plan
79                table[m, sc].best ← r;
80                table[m, sc].type ← 'σ';
81            fi;
82            u ← u ∧ r;              // next selection
83            uc ← uc ∧ rc;           // next selection in the contracted bit vector
84        od;
85        table[m, sc].cost ← best_cost_so_far;        // store best plan
86        s ← m_1 & (s − m_1);        // next subset of relations
87        sc ← sc + 1;                // contracted bit vector is sc + 1
88    until s = 0;
89 od
```

The functions g_{sl}, g_{cp}, g_{jn} describe the operator costs for selections, cross products and joins, respectively (cf. section 3.2.1). If there exist different implementations of an operator that do not influence the costs of subsequent operators we can just enumerate all these operators in a loop, compute the respective costs and update the best plan.

Computation of Cost functions

Suppose the cost function of a join is *symmetric*, i.e. $Cost(A \bowtie B) = Cost(B \bowtie A)$. This induces a symmetry over plans. Let us call two bushy trees that can be transfered into each other by interchanging the order of the subtrees of internal nodes *order-isomorphic*. Since order-isomorphic plans obviously have equal costs we can restrict ourselves to the search space of non-order-isomorphic plans if the join cost functions are symmetric. To achieve this[3], we change line

[3]although the enumeration orders are slightly different, both modifications have the effect of skipping order-isomorphics trees

6 of the algorithm in Figure 4.4 to:

$$6 \quad \textbf{for} \text{ all subsets } L \text{ of } M_k \text{ with } 0 < |l| \leq \lfloor k/2 \rfloor \textbf{ do}$$

In Algorithm A we insert the new line

$$50' \quad l \leftarrow m \,\&\, (l - m);$$

after line 50. This modification cuts down the number of considered joins by a factor of 2.

Although realistic join cost functions are typically *asymmetric* it is nevertheless sufficient to consider the space of non-isomorphic plans since any asymmetric operator *op* and operator cost function c_{op} can be replaced by a *pair of operators* op_l, op_r ("twin operators") and operator cost functions c_l, c_r. For a join operator there usually exist several *physical join operators* corresponding to different join implementations. We replace each of the logical or physical operators with asymmetric cost function by a corresponding pair of twin operators. This approach is particularly useful if a direct comparison of twin cost functions can be done more efficient than two evaluations of the original cost functions. For example, suppose the cost function for a hash join $\bowtie_{hash}$ is

$$c_h(r, s) = c_1 * r + c_2 * r * s * f$$

with constants c_i and join selectivity f. This corresponds to the twin operators $\bowtie_{hash-left}$ and $\bowtie_{hash-right}$ with the respective cost functions

$$\begin{aligned} cost_{hl}(r, s) &= c_1 * r + c_2 * r * s * f \\ cost_{hr}(r, s) &= c_3 * s + c_2 * r * s * f. \end{aligned}$$

Note that $c_{hl}(r, s) < c_{hr}(r, s)$ if and only if $r < s$. Hence, if $r < s$ the plan corresponding to c_{hr} is suboptimal and hence we need only to compute the costs $c_{hl}(r, s)$.

Usually cost functions can be decomposed into a *system of recurrences* involving *auxiliary functions* like (e.g.) the *size* of intermediate results. Whereas the cost function always depends on the concrete subplans this is often not the case for the auxiliary functions. We call an auxiliary function *plan-invariant* if the function does not depend on the concrete plan but only on the set of predicates and relations involved. For example, the size function that computes the number of tuples (number of blocks on disk) resulting from the execution of plan has this property. We can compute the value of plan-invariant cost functions after the loop enumerating the subproblems, i.e. in line 37 of Algorithm A.

Tuning Options

In this section we briefly sketch some possible modifications to "tune" our algorithms (see also [VM96]). General guidelines to improve the performance of programs are described in [Ben82, Ben00]. Note that the following tuning measures are heuristic rules which should improve the performance in most (but not all) cases.

- It is common practice to group related variables in form of records. Nevertheless it may be better to *split tables of records* and use different tables for each record element. The author of [Van95] mentions that this appears to improve cache performance.
- Often there is a choice between on-the-fly computation and tabulation. Which method is faster is usually machine dependent and should be tested. Tabulation is favored by large available physical memory whereas on-the-fly computation is favored by fast processors. Hence, if main memory falls short the choice is shifted from tabulation to on-the-fly computation, since array accesses may now cause page faults leading to expensive I/O operations. It might be beneficial to implement both strategies and decide on which strategy to apply at run-time.

- *Nested if-statements* with simple conditions can often be more efficient than a single if-statement with an intricate condition[4]. In particular it is beneficial to let an if-statement with a complex condition C be preceded by an if-statement with a weaker and *simpler* condition C' (i.e. C' is implied by C). As for Algorithm A, consider the if-statement in line 45 of the second algorithm. One can make the execution of the lines 43-49 dependent of the condition that neither $table[l,u].cost$ nor $table[l',v].cost$ surpasses $best\_cost\_so\_far$.[5] A similar modification can be applied to the if-statement in line 57. Alternatively we could make the execution of lines 44-49 dependent of the weaker condition that $cost'$ does not surpass $best\_cost\_so\_far$.

- Due to the overhead in the implementation of loops it is sometimes slightly more efficient to *unwind a loop partially*. In doing so, the author of [Van95] observes that successive applications of the next-subset-operator $M \,\&\, (S-M)$ causes cyclic patterns. By precomputing the pattern, one cuts the number of next-subset computations by a factor of 4.

- Most of the time in the algorithm is spent for performing arithmetic computations. Hence, it is certainly of advantage if we can drop these costs. One way to achieve this, is to *sacrifice accuracy in favor of performance* by using fixed precision floating point arithmetic instead of arbitrary precision rational arithmetic. The less the precision the better the performance and the lower the constant in the space complexity. Note that if we use fixed precision instead of arbitrary precision, our otherwise exact DP-algorithm is no longer guaranteed to yield an optimal plan.
  Although sufficient precision is important for reliable results it hardly makes sense to consider extremely high costs. In [VM96], Vance and Maier introduced a fixed cost limit to avoid the computation of plans with senseless high costs.

4.2.3 A First DP Algorithm for the Enlarged Execution Space

So far our algorithm does consider joins and selections over *conjunctive predicates* which may occur in the course of the algorithm as indivisible operators of the plan. For example, consider a query on three relations with three join predicates relating all the relations. Then, every join operator in the root of a processing tree has a join predicate which is the conjunction of two basic join predicates. In the presence of expensive predicates this may lead to suboptimal[6] plans, since there may be a cheaper plan which splits a conjunctive join predicate into a join with high selectivity and low costs and a number of secondary selections with lower selectivities and higher costs. Consequently, we do henceforth consider the larger search space of all queries formed by joins and selections with *basic predicates* which are equivalent to our original query. The approach is similar to our first approach but this time we have to take into account the basic predicates involved in a partial solution. First, we replace all conjunctive selection and join predicates in the original query by all its conjuncts. Note that this makes our query graph a multigraph. Let $p_1, \ldots, p_m$ be the resulting set of basic predicates. We shall henceforth use bit vectors of length m to represent sets of basic predicates. Let us now consider an optimal plan P which involves the relations in R and the predicates in P. We denote the costs of such an optimal plan by $C(R,P)$. Obviously, the root operator in P is either a cross product, a join with a basic predicate h_1 or a selection with a basic predicate h_2. Hence, exactly one of the following four cases holds:

cross product: $P \equiv P_1 \times P_2$, $Rel(P) = Rel(P_1) \uplus Rel(P_2)$, $Pred(P) = Pred(P_1) \uplus Pred(P_2)$. The join graphs induced by P_1 and P_2 are not connected with respect to the join graph induced by P. Besides, the subplans P_1 and P_2 are optimal with respect to $Rel(P_1)$, $Pred(P_1)$ and $Rel(P_2)$, $Pred(P_2)$, respectively.

[4] cf. section 4.2.1, evaluation order of conjunctive predicates

[5] we could also compare $table[l,u].cost$ and $table[l',v].cost$ with $\min(best\_cost\_so\_far, cost\_bound)$ where *cost_bound* is an upper bound on the cost of an optimal plan determined with a fast greedy heuristic

[6] with respect to the larger search space defined next

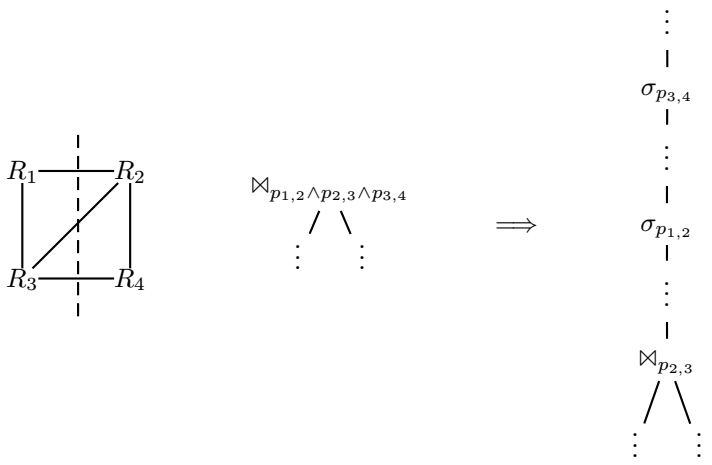


Figure 4.7: Splitting conjunctive predicates

join: $P \equiv P_1 \bowtie_{h_1} P_2$, $Rel(P) = Rel(P_1) \uplus Rel(P_2)$, $Pred(P) = Pred(P_1) \uplus Pred(P_2) \uplus \{h_1\}$. The join graphs induced by P_1 and P_2 are connected by a bridge h_1 in the join graph induced by P. Furthermore, P_1, P_2 are optimal with respect to $Rel(P_i)$ and $Pred(P_i)$.

primary selection: $P \equiv \sigma_{h_2}(P_1)$, $Rel(P) = Rel(P_1)$, $Pred(P) = Pred(P_1) \uplus \{h_2\}$ and P_1 is optimal with respect to the $Rel(P_1)$ and $Pred(P_1)$.

secondary selection: $P \equiv \sigma_{h_1}(P_1)$, $Rel(P) = Rel(P_1)$, $Pred(P) = Pred(P_1) \uplus \{h_1\}$ and P_1 is optimal with respect to $Rel(P_1)$ and $Pred(P_1)$.

Again, it is not difficult to see that the optimality principle holds and one can give a recurrence which determines an optimal plan $Opt(R, P)$ for the problem (R, P) by iterating over all partitions of R applying the corresponding cost function according to one of the above cases.

This leads to the following recurrence

$$Opt(R,P) \quad = \quad \text{min-cost-plan}_{\mathrm{R_1,R_2,P_1,P_2,h}} \left(\begin{array}{l} \{Opt(R_1,P_1) \times Opt(R_2,P_2) \mid \phi_1\}, \\ \{Opt(R_1,P_1) \bowtie_h Opt(R_2,P_2)) \mid \phi_2\}, \\ \{\sigma_h(Opt(R,P \setminus \{h\})) \mid \phi_2\}, \\ \{\sigma_h(Opt(R,P \setminus \{h\})) \mid \phi_3\}, \\ \{\sigma_h(Opt(R,P \setminus \{h\})) \mid \phi_4\} \end{array} \right) \tag{4.12}$$

with the conditions

ϕ_1: $R = R_1 \uplus R_2 \wedge P_1 = R_1 \cap P \wedge \ P_2 = R_2 \cap P$ and the join graphs induced by $P_1$and P_2 are not interconnected in the join graph induced by P.

ϕ_2: $R = R_1 \uplus R_2 \wedge \ P_1 = R_1 \cap P \wedge \ P_2 = R_2 \cap P$ and the induced join graphs of P_1 and P_2 are interconnected by a bridge h in the induced join graph of P.

ϕ_3: there exist subsets R_1, R_2, P_1, P_2 such that $R = R_1 \uplus R_2 \wedge P_1 = R_1 \cap P \wedge \ P_2 = R_2 \cap P$ and in the induced join graph of P the join graphs induced by P_1 and P_2 are interconnected by at least two join predicates, one of them being h.

ϕ_4: $h \in P$ is a primary selection.

Since only atomic join and selection predicates are involved in the above recurrence the corresponding recurrence for computing the costs of an optimal bushy tree is straightforward and we omit it here. More interesting is the problem of how to enumerate the subsets R and P. In principle, there are two "dual" approaches. We first enumerate all subsets of relations and for each subset all possible subsets of predicates. Or we first enumerate all subsets of predicates and for each subset all possible subsets of relations. We choose the following order. First we enumerate all subsets of relations in increasing order with respect to their values as bit vectors. For each subset of relations R, we enumerate all subsets of predicates P' of the maximal set of predicates of the subgraph induced by R in increasing order. In the innermost loop we then enumerate all partitions of R and update the current best optimal plan. One problem which still occurs is to efficiently determine (i.e. by means of a few bit operations) the bit vector of all predicates in the subgraph induced by a certain set of relations[7]. Now, we can inductively compute the bit vectors $e_1(r), e_2(r)$ of all predicates *incident to at least one relation* in r and the bit vector of all predicates *incident and odd number of times* to relations in r by using the 'or' and 'xor' operations on bit vectors, respectively. This is illustrated in Figure 4.8 and 4.9. First consider Figure 4.8. The two graphs on the left side are subgraphs of the graph on the right side, i.e. the nodes of the right graph are the union of the nodes in the two subgraphs on the left. "Thick" edges are edges incident to at least one node of the (sub)graph. Let us denote the bit vector of the nodes of the subgraphs on the left side with r_1 and r_2, respectively, consequently $r = r_1 \,|\, r_2$ is the bit vector of nodes in the right graph. Denoting the bit vectors corresponding to sets of thick edges by $e_1(r), e_1(r_1), e_1(r_2)$, respectively, we have the recurrence $e_1(r_1 \,|\, r_2) = e_1(r_1) \,|\, e_1(r_2)$.

Figure 4.9 is similar to 4.8 except that thick edges are now edges that are incident exactly an odd number of times with a node from the graph. Now, $e_2(r), e_2(r_1), e_2(r_2)$ denote the bit vectors corresponding to the set of thick edges and we have the recurrence $e_2(r_1 \,|\, r_2) = e_2(r_1) \wedge e_2(r_2)$.

From this we can directly compute the bit vectors of all predicates in the induced subgraph which is the set of predicates incident an even and positive number of times to the relations in r, i.e. $e_1(r) \wedge e_2(r)$. Figure 4.10 shows the result of the above example graph. A detailed pseudocode formulation of the algorithm is given below.

```
1  Program DP-OPT-C
2
3     ; n:      number of relations
4     ; m:      number of basic predicates
5     ; c:      global array for optimal costs
6     ;         c[r, p] contains opt. costs of the query involving
7     ;               relations r and predicates p
8     ; s:      global array for output sizes
9     ;         s[r, p] contains size of query with relations r and predicates p
10    ; d:      global array for predicates in the root of an optimal processing tree
11    ;         d[r, p] finally contains opt. root predicate for the query with
12    ;               relations r and predicates p
13    ; e:      global array for optimal splittings
14    ;         e[r, p] contains an optimal partition of r resulting from a join
15    ;               with predicate d[r, p]
16    ; p:      local array for storing information about induced join graphs
```

[7]In the dual approach we have to compute the set of relations $rel[p]$ involved in a set of predicates p. which can be computed in a similar way. The outermost loop then enumerates all subsets of predicates p and for each p the second loop enumerates all supersets of $rel[p]$.

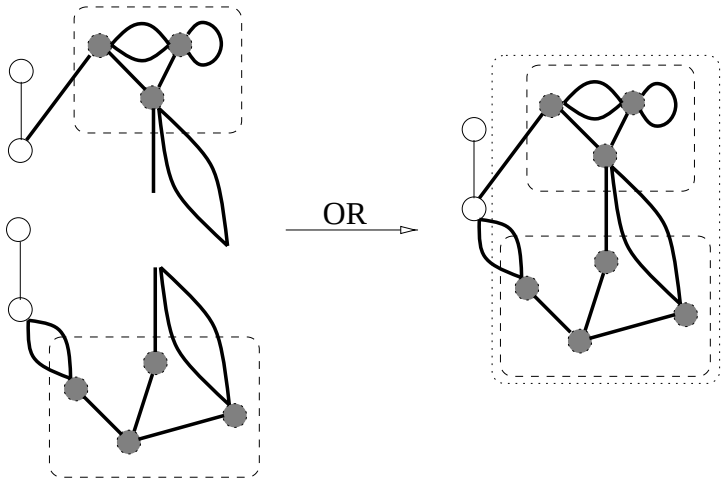


Figure 4.8: Recursive computation of all edges incident to at least one node from a set of nodes

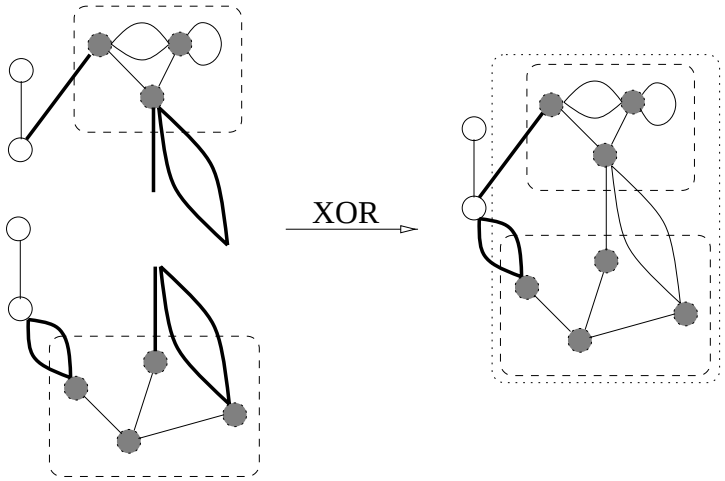


Figure 4.9: Recursive computation of all edges incident an odd number of times to nodes from a set of nodes

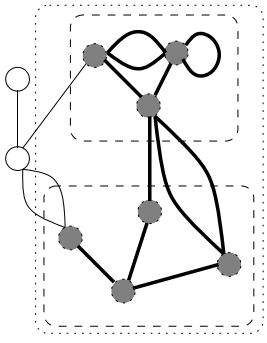


Figure 4.10: Edges in a subgraph induced by a set of nodes

```
17  ;            p[r] contains the bit vector of all predicates in the join graph
18  ;                 induced by the relations r
19  ; p_0, p_1:   local arrays; used to compute elements of p
20  ; sel(h):    selectivity of basic predicate h
21  ; c_h:        cost-factor of basic predicate h
22  ; sels:       bit vector for the set of all basic selection predicates
23
24  // initialization
25  for r ← 0 to n − 1 do
26      s ← 1 ≪ r;
27      p_0[s] ← 0;
28      p_1[s] ← 0;
29      s[s, 0] ← n_r;
30      c[s, 0] ← 0.0;
31      e[s, 0] ← nil;
32  od;
33  for k ← 0 to m − 1 do
34      let r_1, r_2 be the relations to which
35      the basic predicate k refers to;
36      l ← 1 ≪ k;
37      r_3 ← 1 ≪ r_1;
38      r_4 ← 1 ≪ r_2;
39      p_0[r_3] ← p_0[r_3] ∧ l;
40      p_0[r_4] ← p_0[r_4] ∧ l;
41      p_1[r_3] ← p_1[r_3] | l;
42      p_1[r_4] ← p_1[r_4] | l;
43  od;
44  p_0[0] ← 0;  p_1[0] ← 0;
45  // enumerate subproblems
46  for r ← 1 to 2^n − 1 do                  // enumerate subsets of relations
47      r_1 ← r & − r;                       // relation with smallest index
48      r_2 ← r ∧ r_1;                       // all rel. except for rel. with smallest index
49      p_0[r] ← p_0[r_1] ∧ p_0[r_2];        // recurrences to compute p_0[r] and p_1[r]
50      p_1[r] ← p_1[r_1] | p_1[r_2];
51      p[r] ← p_1[r] ∧ p_0[r];              // all predicates in the subgraph induced by r
52      best_cost_so_far ← ∞;                // initialize current best plan
53      best_pred_so_far ← nil;
54      best_split_so_far ← nil;
55      k ← r_1;
56      if k ≠ r then
57          s[r, 0] ← s[k, 0] ∗ s[r ∧ k, 0];
58      fi;
59      while k ≠ 0 and k ≠ r do             // enumerate partitions of r
60          cost ← c[k, 0] + c[r ∧ k, 0] + g_cp(s[k, 0], s[r ∧ k, 0]);   // cross product
61          if cost < best_cost_so_far then          // new plan is cheaper?
62              best_cost_so_far ← cost;             // yes, update!
63              best_pred_so_far ←′ ×′;
64              best_split_so_far ← (k, 0, 0);
65          fi;
66          k ← r & (k − r);                 // next partition
67      od;
68      if (r & − r) ≠ r then                // r contains more than one relation?
69          c[r, 0] ← best_cost_so_far;
70          d[r, 0] ← best_pred_so_far;
```

```
71          e[r,0] ← best_split_so_far;
72      fi;
73      l ← p[r] & −p[r];
74      while l ≠ 0 do                          // enumerate subsets of predicates
75          k ← l & −l;                                  // predicate with smallest index in l
76          best_cost_so_far ← ∞;                        // initialize current best plan
77          best_pred_so_far ← nil;
78          best_split_so_far ← nil;
79          s[r,l] ← s[r, l ∧ k] ∗ sel[k];
80          k ← r & −r;                                  // relation with smallest index in r
81          pd1 ← p[k] & l;                              // corresponding selection
82          pd2 ← p[r ∧ k] & l;                          // selections in r ∧ l
83          while k ≠ 0 and k ≠ r do                     // enumerate partitions of r
84              cut ← l & (p[r] ∧ (p[k] | p[r ∧ k]));    // cut w.r.t. partition (r, r ∧ k)
85              h ← cut & −cut;                          //predicate with smallest index in cut
86              if h = 0 then                // cut empty?
87                  cost ← c[k,pd1] + c[r ∧ k,pd2] + g_cp(s[k,pd1], s[r ∧ k,pd2]);   // cross prod.
88                  if cost < best_cost_so_far then
89                      best_cost_so_far ← cost;
90                      best_pred_so_far ← '×';
91                      best_split_so_far ← (k,pd1,pd2)
92                  fi;
93              else if h = cut then                     // a bridge?
94                  cost ← c[k,pd1] + c[r ∧ k,pd2] + g_jn(s[k,pd1], s[r ∧ k,pd2], c_cut);
95                  if cost < best_cost_so_far then
96                      best_cost_so_far ← cost;
97                      best_pred_so_far ← cut;
98                      best_split_so_far ← (k,pd1,pd2)
99                  fi;
100                 cost ← c[r, l ∧ h] + g_sl(s[r, l ∧ h], c_h);
101                 if cost < best_cost_so_far then
102                     best_cost_so_far ← cost;
103                     best_pred_so_far ← h;
104                     best_split_so_far ← nil
105                 fi;
106             else        // cut contains at least two edges
107                 while h ≠ 0 do        // consider each edge in turn
108                     cost ← c[r, l ∧ h] + g_sl(s[r, l ∧ h], c_h);        // costs of second. join
109                     if cost < best_cost_so_far then                    // new plan cheaper?
110                         best_cost_so_far ← cost;                       // update!
111                         best_pred_so_far ← h;
112                         best_split_so_far ← (k,pd1,pd2)
113                     fi;
114                     cut ← cut ∧ h;      // next predicate in cut
115                     h ← cut & −cut
116                 od;
117             fi;
118             k ← r & (k − r);            // next partition
119             pd1 ← p[k] & l;             // update pd1, pd2
120             pd2 ← p[r ∧ k] & l;
121         od;
122         u ← l & sels;
123         h ← u & −u;
124         while h ≠ 0 do                  // enumerate selections predicates
```

```
125                 cost ← c[r, l ∧ h] + g_sl(s[r, l ∧ h], c_l);      // selection cost
126                 if cost < best_cost_so_far then                   // new plan cheaper?
127                     best_cost_so_far ← cost;                      // update!
128                     best_pred_so_far ← h;
129                     best_split_so_far ← nil;
130                 fi;
131                 u ← u ∧ h;                  // next selection predicate
132                 h ← u & − u
133             od;
134         if (r & − r) ≠ r and l ≠ 0 then
135             c[r, l] ← best_cost_so_far;     // store optimal plan in table
136             d[r, l] ← best_pred_so_far;
137             e[r, l] ← best_split_so_far;
138         fi;
139         l ← p[r] & (l − p[r])               // next subset of predicates
140     od              // next subset of relations
```

The functions g_{sl}, g_{cp}, g_{jn} describe the operator costs for selections, cross products and joins, respectively (cf. section 3.2.1). The algorithm can easily be modified to account for different implementations of operators (cf. end of section 4.2.2).

Complexity Issues

Let m be the number of different basic predicates (joins and selections) and n the number of base relations. In order to analyze the asymptotic time and space complexities we reduce the algorithm to its nested loops:

```
loop 1      for r ← 1 to 2^n − 1 do
loop 2          for all l which are bitwise less or equal to p[r] do
loop 3              for all k which are bitwise less or equal to r do
loop 4                  for all edges h in the cut induced by the partition (k, r − k) do
                            ...
                        od
                    od
loop 5              for all non-zero bits h in l ∧ sels do
                        ...
                    od
                od
            od
```

Recall that *sels* denotes the bit vector of all selection predicates.

Time Complexity

First we derive a crude upper bound on the number of considered partial plans. Surely, every subgraph of the join (multi)graph G induced by k relations has no more than m edges and every cut of G has no more than m edges. Hence we have the following upper bound on the number of

partial plans:

$$\begin{aligned}
&\underbrace{\sum_{k=1}^{n}\binom{n}{k}}_{\text{loop 1}}\underbrace{\sum_{j=0}^{m}\binom{m}{j}}_{\text{loop 2}}\Big(\underbrace{\sum_{i=1}^{k-1}\binom{k}{i}}_{\text{loop 3}}\underbrace{m}_{\text{loop 4}}+\underbrace{j}_{\text{loop 5}}\Big)\\
&= \sum_{k=1}^{n}\binom{n}{k}\sum_{j=0}^{m}\binom{m}{j}((2^k-2)m+j)\\
&= m\sum_{k=1}^{n}\binom{n}{k}(2^k-2)\sum_{j=0}^{m}\binom{m}{j}+\sum_{k=1}^{n}\binom{n}{k}\sum_{j=0}^{m}\binom{m}{j}j\\
&= m2^m\sum_{k=1}^{n}\binom{n}{k}(2^k-2)+m\sum_{k=1}^{n}\binom{n}{k}\sum_{j=0}^{m-1}\binom{m-1}{j}\\
&= m2^m(3^n-2^{n+1}+1)+m2^{m-1}(2^n-1)\\
&\leq m2^m3^n-3m2^{m+n-1}+m2^{m-1}]
\end{aligned}$$

which is $O(m2^m3^n)$ for $m>0$.

Note that this is a rather pessimistic upper bound which does not account for the structure of the query graph G. Unfortunately, even for simple chain queries the complexity bounds are so complex that we were not able to derive closed forms.

Space Complexity
We have the following coarse upper bound on the number of table entries used by the algorithm

$$\sum_{k=1}^{n}\binom{n}{k}\sum_{j=0}^{m}\binom{m}{j}=\sum_{k=1}^{n}\binom{n}{k}2^m=2^m(2^n-1)$$

Space utilization It turns out that the new algorithm wastes space too. E.g. for the case of 4 relations and 8 predicates of which 2 are selection predicates, only about 10% of all table entries are used to compute the optimal solution. Since induced subgraphs have fewer edges, one should also use fewer bits to code their edge sets. More exactly, for every bit vector r representing a subset of the set of nodes one should compress the bit vector l representing a subset of edges in the corresponding induced subgraph with respect to $p[r]$, the bit vector representing all edges induced by the nodes in r. Furthermore, rows of the tables $c[][], d[][]$ and $e[][]$ are dynamically allocated as soon as new values of $p[r]$ are known. The dimension of the respective rows is $2^{\beta(r)}$, where $\beta(r)$ denotes the number of non-zero bits in the bit vector r. Now, the same techniques can be applied than for the non-splitting algorithm, although we shall not discuss them here, since the amount of wasted space is far less than for the non-splitting algorithm and may be tolerated in practice.
The tuning measures and short-cuts discussed in section 4.2.2 apply to the second algorithm too. Hence, we will not discuss these issues again.

4.2.4 A Second DP Algorithm for the Enlarged Execution Space

So far we concentrated on *fast enumeration* without analyzing structural information about the problem instance at hand. To give an example, suppose that the query graph is 2-connected. Then every cut contains at least two edges; therefore, for the subproblem consisting of all relations and

all predicates, the topmost operator in a processing tree can neither be a join nor a cross product. Hence, the second algorithm iterates in vain over all 2^n partitions of n relations.

We will now describe the approach in our third algorithm which uses information about the *structure of the join graph* to avoid the enumeration of unnecessary subproblems. Suppose that a query refers to the set of relations R and the set of predicates P. Let $n = |R|$, $m = |P|$, and assume that s out of m predicates are selections. Now, consider an optimal processing tree T for a subquery induced by the relations R' and the predicates P' where $R' \subseteq R$ and $P' \subseteq P$. $G(R', P')$ denotes the join graph induced by R and P. Obviously, the root operator in T is either

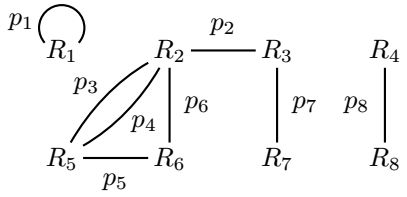
- a cross product or
- a join or
- a primary selection or
- a secondary selection.

Depending on the operation in the root of the processing tree, we can classify the join graph $G(R', P')$ as follows. If the root operation is a cross product, then $G(R', P')$ decomposes into *two or more connected components*. Otherwise, if it is a join with (basic) predicate h then $G(R', P')$ contains a *bridge* h. If it is a secondary selection with (basic) predicate h then $G(R', P')$ contains an *edge* h connecting two different relations. And, if it is a primary selection with predicate h then $G(R', P')$ contains a *loop* h. Table 4.1 summarizes the different cases.

root operator:	query graph:
a cross product	a partition of the connected components
a join	a bridge + a partition of the connected components
a primary selection	a loop
a secondary selection	an edge (no loops)

Table 4.1: Correspondence between operator types and graph structure

Example Consider the query graph on the right. The connected components $\{R_1\}, \{R_2, R_3, R_5, R_6, R_7\}$ and $\{R_4, R_8\}$ give raise to $2^3 - 2 = 6$ cross products. p_1 is a loop that corresponds to a selection and the bridges p_2, p_7 and p_8 correspond to joins. Note that each bridge leads to 6 joins—one for each combination of the connected components. The other edges p_3, p_4, p_5, p_6 correspond to secondary selections. □



To enumerate all possible cross products we build all non-trivial partitions of the set of connected components of G'. All possible joins can be enumerated as follows. We iterate through all bridges of G. For each bridge (join predicate) h we determine the connected component C' containing h together with the two subcomponents of C' connected by h. Then we enumerate all partitions of $C - C'$ and add the first subcomponent to the first block of the partition and the second subcomponent to the second block of the partition (interchanging the role of the two subcomponents gives raise to another join). All secondary selections can be enumerated by iterating through all edges (skipping loops). By stepping through all loops in $G(R', P')$, we enumerate all primary selections. In a sense we tackle subproblems in the reverse direction than we did before. Instead of enumerating all partitions and analyzing the type of operation it admits we consider possible types of operations and enumerate the respective subproblems.

The pseudocode of our algorithm is:

```
1  Program DP-OPT-D
2      // n: number of relations
3      // p: number of predicate s
4      initialize tables cost[][] and plan[][];
5      for r ← 1 to 2^n − 1 do
6          use a recurrence to compute p[r], the bit vector of all relations
7              in the subgraph of G(R, P) induced by the relations in r.
8          for all l bitwise smaller or equal to p[r] do
9              best_cost_so_far ← ∞;
10             best_plan_so_far ← nil;
11             perform a depth-first search to determine the following parameters:
13             k:      number of components in G(r, l);
14             u:      number of bridges in G(r, l);
15             b:      bit vector of all bridges in G(r, l);
16             cp[i]:  bit vector of all relations in i-th connected component of G(r, l);
17             lc[i]:  bit vector of all relations in one of the components resulting from
18                     the removal of the i-th bridge in G(r, l);
19             cn[i2]: number of the conn. component two which relation i belongs (i2 = 2^i)
20             for each partition (r1, r2) of r with each of the k connected
21                   components being completely in r1 or r2 do
22                 let T1 be an optimal plan for G(r1, p[r1]);
23                 let T2 be an optimal plan for G(r2, p[r2]);
24                 consider the plan T ← T1 × T2;
25                 update best_cost_so_far, best_plan_so_far;
26             od;
27             for i ← 1 to u do
28                 let h be the i-th bridge w.r.t the ordering in b;
29                 let j be the index of the connected comp in G(r, l) to which h belongs
30                 let T1 be an optimal plan for G(w[i], p[w[i]]);
31                 let T2 be an optimal plan for G(c[h] − w[h], p[c[h] − w[h]]);
32                 consider the plan T ← T1 ⋈_h T2;
33                 update best_cost_so_far, best_plan_so_far;
34             od;
35             for each predicate h in l do
36                 let T1 be an optimal plan for G(r, l − h);
37                 consider the plan T ← σ_h(T1);
38                 (including primary as well as secondary selections)
39                 update best_cost_so_far, best_plan_so_far;
40             od;
41             cost[r, l] ← best_cost_so_far;
42             plan[r, l] ← best_plan_so_far;
43             od;
44         od
45     od;
46     return plan
47 end
```

There is still the problem of identifying bridges along with their subcomponents. We shall describe three possible solutions to this problem. In the first solution we perform a depth-first search to determine each of the connected components of a subgraph. It turns out that it is not necessary to compute all the components of a subgraph if we tabulate one specially chosen component—the component that contains the relation with the smallest index. In the following we call the relation with the smallest index *the distinguished relation* and the component that contains the distinguished relation the *distinguished (connected) component*. It is possible to compute the distinguished component without depth-first search by only making use of the previously computed tables. Let G be the query graph induced by the relations R and predicates P and let us denote the relations with the smallest and second smallest indices with R_1 and R_2, respectively. Furthermore, let us denote the query graph induced by $R' = R - \{R_1\}$ and $R'' = R - \{R_2\}$ with G' and G'', and the distinguished connected components of R' and R'' with C_1 and C_2, respectively. Now, if there is an edge between R_1 and R_2 in G or if C_1 and C_2 overlap then C_1 is connected to C_2 in G and $C = C_1 \cup C_2$. Otherwise, C_1 and C_2 are disjoint and $C = C_1$. The remaining components can be looked up using the table of the previously computed components.

After we have computed the connected components of the subgraph we enumerate all partitions of the connected components to build all possible cross products. Next we consider possible joins and selections. We iterate through all predicates. If the predicate p is a primary selection we compute the corresponding plan and update the current best plan. Otherwise, we start a depth-first search on one of the relations incident with p to find out whether the predicate is a bridge. If it is a bridge, we use the two subcomponents connected by the bridge to construct two plans. The first two plans are joins $\bowtie_p$—with the two subplans interchanged in the second plan. Finally we consider the case where p is a secondary selection σ_p. Note that bit vectors can be used to speed up depth-first search.

In the second version we identify all the bridges in the induced subgraph directly. The problem of identifying bridges is very similar to the problem of identifying *articulation points*—nodes whose removal increases the number of connected components.[8] Articulation points and bridges can be computed by the following augmented version of depth-first search. We assume that the reader is familiar with the depth-first search algorithm (as discussed in e.g. [CLR90]) and the related notions of *dfs-numbers*, *forward* and *back edges*. In the course of depth-first search we maintain two numbers for each visited node v: the dfs-number and the minimum dfs-number in a node reachable from v in a sequence of forward edges followed by one back edge. Bridges (u, v) are now identified by the condition $dfs[u] < low[v]$. Along with the bridge we also need to store one of the two subcomponents connected by the bridge. Note that such a subcomponent is just the set of visited nodes when we depth-first search encounters a bridge. Instead of computing all the components and bridges with depth-first search we can alternatively compute the distinguished connected component and look up the remaining components in the tables of previously computed subproblems.

The third solution manages without depth-first search. The idea is to iterate through the edges while modifying the graph such that the edge under consideration is always in the distinguished connected component. This can be done by iterating through the edges component by component in the order of increasing indices of distinguished relations while successively removing the connected components that have already been examined.

As the first step we compute the distinguished connected component by using a recurrence and look up the remaining connected components. Then we iterate through the edges of the subgraph. We do this by first iterating through the connected components in the order that has been determined in the previous step and for each component C we iterate through all the edges in C. Then, for each such edge e, we enumerate all partitions of relations such that the relations of no component are split across the partition. This can be done efficiently by enumerating the subsets of the set of connected components in gray code order. The transition from one partition to the next is just an xor-operation with the bit vector of the connected component addressed by the change-bit in

[8] Articulation points occur in connection with *biconnected components*. A biconnected component is a maximal set of nodes such that each pair of nodes lies on a simple circle [CLR90].

the gray code. Within the innermost loop we first consider the case of a cross product between the relations in the partition (only if neither of the blocks of the partition is empty) and update the cost of the best plan so far. Then we consider the case of a join with predicate e. In order to determine whether e is a bridge we look up the distinguished component C' in the subgraph with the edge e removed. If C is a true subset of the distinguished component in G, then we know that e is a bridge and hence corresponds to a join predicate. In this case, C is one of the subcomponents connected by the bridge e. We compute the cost of the join and update the cost of the best plan so far. Otherwise, if C' equals C, e lies on a circle and therefore corresponds to a secondary selection predicate. We compute the cost of the new plan and update the best plan so far.
An implementation in C is given below[9].

```
Algorithm DP-OPT-D:

// initialization
p0[0] = p1[0] = 0;
for (r = 0, r2 = 1; r < n, r++, r2 <<= 1) {
  p0[r2] = p1[r2] = bit vector of edges incident to node r;
  optPlan[r2][0].size = relation[r].size;  // cardinality of relation r
  optPlan[r2][0].cost = scanCost(size[r]);
  optPlan[r2][0].pred = 0;
  optPlan[r2][0].leftRel   = 0;
  optPlan[r2][0].leftPred  = 0;
  optPlan[r2][0].rightPred = 0;
}
for (r = 1; r < 1<<n; r++){ // enumerate all subsets of relations
  r1 = r&-r;                // relation with smallest index
  r2 = r^r1;
  p0[r] = p0[r1]^p0[r2];       // edges incident with an even number of relations in r
  p1[r] = p1[r1]|p1[r2];       // edges incident with some relations in r
  p2[r] = p0[r]^p1[r];         // edges induced by the relations r
  l  = p2[r]&-p2[r];
  while (l) {               // enumerate all subsets of edges
    // use recurrence to compute optPlan[r][l].size
    if (l)   // any predicates in l?
      optPlan[r][l].size = optPlan[r][l^(l-1)].size * predicate[l&-l].selectivity;
    else if (r2)   // two or more relations and no predicates?
      optPlan[r][l].size = optPlan[r1][0].size * optPlan[r2][0].size;
    // use recurrence to compute the distinguished connected component
    if (r2) {                 // is there more than one relation in r?
      r3 = r2&-r2;            // relation with 2nd smallest index
      r4 = r^r3;
      comp1 = component[r4][l&p2[r4]]; // conn. components in subgraph without r3
      comp2 = component[r2][l&p2[r2]]; // conn. components in subgraph without r1
      if ((l&p2[r1|r3]&~sels)            // is r1 connected to r2 or
         || (comp1&comp2)) {             // do comp1 and comp2 overlap?
        component[r][l] = comp1|comp2;   // result is union of comp1 and comp2
      } else                             // r1 is not connected to r2
        component[r][l] = comp1;         // result is comp1
    } else
      component[r][l] = r;               // special case of one relation
    newRel  = r;
    newPred = l;
```

[9] we use C++-style comments

```
nc = 1;
// look up remaining connected components
// (components are ordered by increasing smallest indices)
while (newRel) {                          // still more connected components?
  block[nc] = component[newRel][newPred];
  newRel  ^= block[nc];                   // remove distinguished component
  newPred &= p2[newRel];                  // update new predicates
  nc <<= 1;
}
// enumerate all partitions of connected components (using a gray code)
for(index = 0,
      last  = 0,
      gray  = 0,           // gray code
      delta = 0,           // bit that changed
      thePartition = 0;    // relations in the left block of the partition
    index < nc;
    index++,
      last  = gray,
      gray  = index^(index>>1),  // next gray code
      delta = last^gray) {
  thePartition   |= block[delta];
  leftRelations   = thePartition;
  rightRelations  = r^leftRelations;
  leftPredicates  = l&p2[leftRelations];
  rightPredicates = l&p2[rightPredicates];
  // consider cross products
  if (leftRelations && rightRelations) {            // skip trivial partitions
    dcost = cost[leftRelations][leftPredicates] +
            cost[rightRelations][rightPredicates];  // cost of subtrees
    cost  = dcost1 + cpCost(size[leftRelations][leftPredicates],
                            size[rightRelations][rightPredicates]);
    if (cost<bestPlan.cost) {
      bestPlan.cost = cost;    // update best plan
      bestPlan.pred = 0;
      bestPlan.leftRel   = leftRelations;
      bestPlan.leftPred  = leftPredicates;
      bestPlan.rightPred = rightPredicates;
    }
  }
  // iterate through the connected components
  for (i = 1,
         newRel = r;
       i < nc;
       newRel = r^block[i],      // delete relations of previous components
         i <<= 1) {
    blockPred = l&p2[block[i]];  // predicates in connected component i
    currPred = blockPred&-blockPred;
    while (currPred) {    // iterate through predicates in blockPred
      newPred = (l&p2[newRel])^currPred;
      // compute distinguished component in subgraph (newRel,newPred)
      if (i&gray)   // is currPred in the left block of the partition?
        subComp = component[newRel][newPred];
      else
        subComp = block[i]^component[newRel][newPred];
      if (subComp&&(block[i]^subComp)) {  // is currPred a join predicate?
        leftRelations   = thePartition|subComp;
        leftPredicates  = l&p2[leftRelations];
        rightRelations  = r^leftRelations;
```

```
            rightPredicates = l&p2[rightRelations];
            dcost = cost[leftRelations][leftPredicates]       // cost of subtrees
                  + cost[rightRelations][rightPredicates];
            cost  = dcost + joinCost(size[leftRelations][leftPredicates],
                                     size[rightRelations][rightPredicates],
                                     size[r][l],pred[currPred].costfactor);
            if (cost<bestPlan.cost) {
              bestPlan.cost = cost;      // update best plan
              bestPlan.pred = currPred;  // join predicate
              bestPlan.leftRel   = leftRelations;
              bestPlan.leftPred  = leftPredicates;
              bestPlan.rightPred = rightPredicates;
            }
          }
          else {   // currPred is a selection
            dcost = cost[r][l^currPred];   // cost of subtree
            cost  = dcost + selCost(size[r][l^currPred],
                                    size[r][l],pred[currPred].costfactor);
            if (cost<bestPlan.cost) {
              bestPlan.cost = cost;      // update best plan
              bestPlan.pred = currPred;  // selection predicate
              bestPlan.leftRel   = 0;
              bestPlan.leftPred  = 0;
              bestPlan.rightPred = 0;
            }
          }
        }
        blockPred ^= currPred;
        currPred   = blockPred&-blockPred;   // next predicate
      }
    }
    optPlan[r][l].cost  = bestPlan.cost;   // store optimal plan
    optPlan[r][l].pred  = bestPlan.pred;
    optPlan[r][l].leftRel   = bestPlan.leftRel;
    optPlan[r][l].leftPred  = bestPlan.leftPred;
    optPlan[r][l].rightPred = bestPlan.rightPred;
    l = p2[r]&(l-p2[r]);      // next subset of predicates
  }
 }
return optPlan;
```

An implementation of the second version (with ordinary depth-first search instead of augmented dfs) can be found in the appendix A.

4.2.5 Analysis of Complexity

Let us now analyze the asymptotic time and space complexities of the new dynamic programming algorithm. Our yardstick will be the number of considered partial plans and the number of table entries to store optimal plans, respectively. Obviously, the number of connected components of a subgraph induced by k relations and j edges is bounded by k and the number of bridges is

bounded by $k-1$. Hence, the number of considered partial plans is at most

$$\begin{aligned}
&\sum_{k=1}^{n}\binom{n}{k}\sum_{j=0}^{m}\binom{m}{j}[2^{k-1}+j2^{k-j+1}+j] \\
&= \sum_{k=1}^{n}\binom{n}{k}2^{k-1}\sum_{j=0}^{m}\binom{m}{j}+\sum_{k=1}^{n}\binom{n}{k}2^{k+1}\sum_{j=0}^{m}\binom{m}{j}j2^{-j}+\sum_{k=1}^{n}\binom{n}{k}\sum_{j=0}^{m}\binom{m}{j}j \\
&= 2^{m-1}\sum_{k=1}^{n}\binom{n}{k}+2m\sum_{k=1}^{n}\binom{n}{k}2^{k}\sum_{j=0}^{m-1}\binom{m-1}{j}2^{-j}+m\sum_{k=1}^{n}\binom{n}{k}\sum_{j=0}^{m-1}\binom{m-1}{j} \\
&= 2^{m-1}(3^n-1)+2m\left(\frac{3}{2}\right)^{m-1}\sum_{k=1}^{n}\binom{n}{k}2^{k}+m2^{m-1}\sum_{k=1}^{n}\binom{n}{k}
\end{aligned}$$

$$\begin{aligned}
&= 2^{m-1}(3^n-1)+2m\left(\frac{3}{2}\right)^{m-1}(3^n-1)+m2^{m-1}(2^n-1) \\
&= O(2^m3^n+m\left(\frac{3}{2}\right)^{m}3^n+m2^{m+n})
\end{aligned}$$

As to the asymptotic worst case time complexity, we additionally have to account for the depth-first search in the innermost loop which can be done in time $O(k+j)$. This leads to the complexity

$$\begin{aligned}
&O\Big(\sum_{k=1}^{n}\binom{n}{k}\sum_{j=0}^{m}\binom{m}{j}[O(k+j)+2^{k-1}+j2^{k-j+1}+j]\Big) \\
&= O\Big(\sum_{k=1}^{n}\binom{n}{k}k\sum_{j=0}^{m}\binom{m}{j}+\sum_{k=1}^{n}\binom{n}{k}\sum_{j=0}^{m}\binom{m}{j}j+ \\
&\quad 2^{m-1}(3^n-1)+2m\left(\frac{3}{2}\right)^{m-1}(3^n-1)+m2^{m-1}(2^n-1)\Big) \\
&= O(n2^{n+m}+m2^{m-1}(2^n-1)+2^{m-1}(3^n-1)+2m\left(\frac{3}{2}\right)^{m-1}(3^n-1)+m2^{m-1}(2^n-1)) \\
&= O(2^{m-1}(3^n-1)+2m\left(\frac{3}{2}\right)^{m-1}(3^n-1)+(m+n)2^{m+n}-m2^m) \\
&= O(2^m3^n+m\left(\frac{3}{2}\right)^{m}3^n+(m+n)2^{m+n}-m2^m)
\end{aligned}$$

The number of table entries used by the algorithm is

$$\sum_{k=0}^{n}\binom{n}{k}\sum_{j=0}^{m}\binom{m}{j} = 2^{n+m}$$

The above time and space complexities are generous upper bounds which do not account for the structure of a query graph. We will next study the complexity of our algorithm for the important special case of acyclic queries.

Acyclic join graphs without multiple edges

Consider an acyclic join graph G with n nodes, $m=n-1$ edges (no loops and multiple edges) and s loops (possibly some multiple loops) and let G' be an induced join graph with k nodes and

l edges and t loops. Since G' obviously has $m-l+1 = n-l$ connected components, the number of induced subgraphs with k nodes, l edges and t loops is

$$\binom{n-1}{l}\binom{n-(l+1)}{k-(l+1)}\binom{s}{t} \quad = \quad \binom{n-1}{n-l-1}\binom{n-l-1}{n-k}\binom{s}{t}$$

Each of these subgraphs has

- $n-l$ components and hence 2^{n-l-1} possibilities for cross products
- l bridges and hence $l2^{n-l+1}$ possibilities for joins
- $l+s$ edges (including loops) and hence $l+s$ possibilities for selections

This leads to the following upper bound on the number of considered partial plans

$$\begin{aligned}
&\sum_{k=1}^{n}\sum_{l=0}^{m}\sum_{t=0}^{s}\binom{n-1}{n-l-1}\binom{n-l-1}{n-k}\binom{s}{t}[2^{n-l-1}+l2^{n-l+1}+(l+s)] \\
&= 2^s\sum_{l=0}^{n-1}\binom{n-1}{l}[2^l+(n-l+1)2^{l+2}+n-l+1]\sum_{k=0}^{n-1}\binom{n-l-1}{k} \\
&\quad +s2^s\sum_{l=0}^{n-1}\binom{n-1}{l}\sum_{k=0}^{n-1}\binom{n-l-1}{k} \\
&= 2^s\sum_{l=0}^{n-1}\binom{n-1}{l}[(4(n-l)-3)2^l+n-l-1]\sum_{k=0}^{n-l-1}\binom{n-l-1}{k} \\
&\quad +s2^s\sum_{l=0}^{n-1}\binom{n-1}{l}\sum_{k=0}^{n-l-1}\binom{n-l-1}{k} \\
&= 2^s\sum_{l=0}^{n-1}\binom{n-1}{l}[(4(n-l)-3)2^l+n-l-1]2^{n-l-1}+\sum_{l=0}^{n-1}\binom{n-1}{l}2^{n-l-1} \\
&= 2^s\sum_{l=0}^{n-1}\binom{n-1}{l}[(4(n-l)-3)2^{n-1}+(n-l-1)2^{n-l-1}]+s2^{s-1}2^{n-1}\sum_{l=0}^{n-1}\binom{n-1}{l}2^{-l} \\
&= 2^s2^{n-1}\sum_{l=0}^{n-1}\binom{n-1}{l}(4(n-l)-3)+2^s2^{n-1}\sum_{l=0}^{n-1}\binom{n-1}{l}(n-l-1)2^{-l}+s2^{s-1}2^{n-1}\left(\frac{3}{2}\right)^{n-1} \\
&= 4n2^{n-1}2^s2^{n-1}-42^{n-1}2^s(n-1)2^{n-2}-32^{n-1}2^s2^{n-1}+n2^{n-1}2^s\left(\frac{3}{2}\right)^{n-1} \\
&\quad -(n-1)2^{n-1}2^s\left(\frac{3}{2}\right)^{n-2}-2^{n-1}2^s\left(\frac{3}{2}\right)^{n-1}+s2^{s-1}2^{n-1}\left(\frac{3}{2}\right)^{n-1} \\
&= 2^s\left[4^n\left(\frac{n}{2}-\frac{1}{4}\right)+3^n\left(\frac{2n}{9}-\frac{2}{9}+\frac{s}{6}\right)\right] \\
&= O(n4^n2^s+(s+n)3^n2^s)
\end{aligned}$$

And for the asymptotic time complexity we have

$$\begin{aligned}
&O(\sum_{k=1}^{n}\sum_{l=0}^{m}\sum_{t=0}^{s}\binom{n-1}{n-l-1}\binom{n-l-1}{n-k}\binom{s}{t}[O(k+l)+2^{n-l-1}+l2^{n-l+1}+(l+s)]) \\
&\quad = O(2^s4^n\left[\frac{3n^2}{8}-\frac{3n}{16}+\frac{1}{16}\right]+2^s3^n\left[\frac{8n^2}{27}-\frac{2n}{9}-\frac{2}{27}+\frac{s}{6}\right]) \\
&\quad = O(2^s(n^24^n+(n^2+s)3^n))
\end{aligned}$$

4.2.6 Generalizations

Different Join Algorithms

In addition to the problem of optimal ordering expensive selections and joins the optimizer eventually has to select a join algorithm for each of the join operators in the processing tree. Let us call *annotated processing trees* those where a join method from a certain pool of join implementations is assigned to each join operator. We now describe how our dynamic programming algorithm can be generalized to determine optimal annotated bushy processing trees in one integrated step.

The central point is that the application of a certain join method can change the physical representation of an intermediate relation such that the join costs in subsequent steps may change. For example, the application of a sort-merge join leaves a result which is sorted with respect to the join attribute. Nested loop joins are order preserving, that is if the outer relation is sorted with respect to an attribute A then the join result is also sorted with respect to A. A join or selection may take advantage of the sorted attribute. In the following discussion we restrict ourselves to the case where only nested-loop joins and sort-merge joins are available. Furthermore, we assume that all join predicates in the query are equi-joins so that both join methods can be applied for every join that occurs. This is not a restriction of the algorithm but makes the following discussion less complex.

Consider an optimal bushy plan P for the relations in $Rel(P)$ and the selections in $Sel(P)$. Again we can distinguish between a join operator representing the root of the plan tree and a selection operator representing the root. In case of a join operator we further distinguish between the join algorithms nested-loop (nl) and sort-merge (sm). Let $C_a(R,S)$ be the costs of an optimal subplan for the relations in R and selections in S, where the result is sorted with respect to the attribute a (of some relation $r \in R$). $C(R,S)$ is similarly defined, but the result is not necessarily sorted with respect to any attribute. Now, the optimality principle holds and the cost function satisfies the following recurrence:

$$C(R,S) = \min \begin{cases} \min_{\emptyset \subset R_1 \subset R} & (C(R_1,S) + C(R \setminus R_1,S) + g_{nl}(S(R_1,S), \\ & \quad S(R \setminus R_1,S), S(R,S), c_p)) \\ \min_{\sigma(R_i) \in S} & (C(R, S \setminus R_i) + g_{sl}(S(R, S \setminus R_i), S(R,S), c_p)) \end{cases} \tag{4.13}$$

$$C_{a|b}(R,S) = \min \begin{cases} \min_{\emptyset \subset R_1 \subset R}(C(R_1,S) + C(R \setminus R_1,S) + g_{sm}(S(R_1,S), S(R \setminus R_1,S), S(R,S), c_p)) \\ \min_{\emptyset \subset R_1 \subset R}(C_a(R_1,S) + C(R \setminus R_1,S) + g^1_{sm}(S(R_1,S), S(R \setminus R_1,S), S(R,S), c_p)) \\ \min_{\emptyset \subset R_1 \subset R}(C(R_1,S) + C_b(R \setminus R_1,S) + g^2_{sm}(S(R_1,S), S(R \setminus R_1,S), S(R,S), c_p)) \\ \min_{\emptyset \subset R_1 \subset R}(C_a(R_1,S) + C_b(R \setminus R_1,S) + g^{1,2}_{sm}(S(R_1,S), S(R \setminus R_1,S), S(R,S), c_p)) \\ \min_{\sigma_{R_i} \in S}(C(R, S \setminus R_i) + g^1_{sl}(S(R, S \setminus R_i), S(R,S), c_p)) \end{cases} \tag{4.14}$$

where the join cost functions are defined as

$$\begin{aligned} g_{nl}(n_1,n_2,n_3,c_p) &= n_1 * n_2 * c_p + n_3 * c_m \\ g_{sm}(n_1,n_2,n_3,c_p) &= (n_1 \log n_1 + n_2 \log n_2) * c_p + n_3 * c_m \\ g^1_{sm}(n_1,n_2,n_3,c_p) &= n_2 \log n_2 * c_p + n_3 * c_m \\ g^2_{sm}(n_1,n_2,n_3,c_p) &= n_1 \log n_1 * c_p + n_3 * c_m \\ g^{1,2}_{sm}(n_1,n_2,n_3,c_p) &= n_3 * c_m \end{aligned}$$

c_p denotes the per-tuple cost factor of evaluating the predicate in a selection or join and c_m is a constant accounting for writing the results back to disk. An upper index i indicates that the ith input relation is already sorted with respect to the join attribute. a and b range over all attributes of the relations R_1 and $R \setminus R_1$,respectively, such that attribute a is a join attribute with respect to R_1 and attribute b is a join attribute with respect to $R \setminus R_1$. c denotes a join attribute in the result.

Time Complexity Every basic join predicate contributes at most two relevant attributes, hence for every specific pair of values R, S, the number of attributes a for which we have to compute the table entry $C_a(R, S)$ is bounded from above by twice the number of all basic join predicates r. Under the assumptions of our first algorithm we have $r = O(n^2)$, which leads to the following asymptotic upper bound on the number of considered partial plans:

$$O(n^2[3(5/3)^c]^n + (2cn^3/3)[2(3/2)^c]^n)$$

Space Complexity By an analogous argumentation, the space complexity also grows by at most a factor of $2r$. Hence, the asymptotic number of table entries in our new algorithm is

$$O(n^2[2(3/2)^c]^n)$$

Affine join cost functions

Linear join cost functions usually greatly simplify the problem of ordering joins and selections [HS93, Hel94, Hel98, CS97, IK84, KBZ86]. For example, the authors of [IK84] first used a complex cost function counting disc accesses in a special block-wise nested loop join. With this cost function they proved that the problem of finding optimal left-deep trees is NP-hard. The authors of [IK84] also showed that the problem is solvable in polynomial time using a simpler join cost function that is linear in the first argument (the size of the subtree). A generalization to linear cost functions are affine[10] cost functions of the form $C(R, S) = a + b * R + c * S + d * R * S$ with constants a, b, c, d. Suppose the join cost function is affine. As a consequence of the results in [SM96, YKY+91], in an optimal bushy processing tree all selection and join predicates that are *applicable to a certain inner node v of the tree* occur in *rank-sorted order* on the path from v to the root of the tree! Although this was first observed in [CS96], the authors claimed that it holds for *all* cost functions but failed to give a proof. In fact, it is not difficult to find counter examples which show their claim to be wrong for non-ASI cost functions.[11] Later they fixed this by constraining their results to affine join cost functions [CS97]. Now, how can our algorithms benefit from affine join cost functions? There are two places where we need to iterate over sets of primary and secondary selection predicates, respectively. Let us first consider the case of selection predicates. Since two selection predicates referring to different relations induce different paths to the root of tree these predicates need not be rank-sorted. The case is different for relations referring to the same relation. All the induced paths from the relation to the root coincide and the predicates have to be rank-sorted. Therefore modify the algorithm as follows. First, at the start of the optimization algorithm we re-number the selections by increasing ranks. Note that we have to undo this re-ordering in the result of the algorithm. Second we replace line 191 in the algorithm by the following two lines

```
l_compl         = (p2[r] ^ (l & sels)) & sels;
root_predicates = l & sels & ((l_compl & -l_compl) - 1);
```

In the first line we compute in `l_compl` all selections in the join graph induced by r which do not occur in l. The second line computes all selections in l whose local rank is smaller than the rank of all selections in `l_compl`.
How can the enumeration of joins benefit from affine cost functions? It turns out that join predicates can be used in a similar way but cause considerable more work. The problem is that sorting the join predicates at the beginning of the algorithm does not help much since the ranks of join predicates depend on the relations in the subtree. Hence we suggest the following compromise. With a statement similar to the one above we compute the bit vector `root_predicates` of all predicates applicable to the relations r and predicates l:

[10] In [CS97] these functions are called *regular cost functions.*
[11] For example, take the standard cost function for sort-merge joins from [KBZ86].

```
l_compl    = (p2[r] ^ (l & ~sels)) & ~sels;
root_predicates = l & ~sels & ((l_compl & -l_compl) - 1);
```

If the number of non-zero bits in `root_predicates` is small (does not exceed, say 4) we spare the effort of finding a predicate with minimum rank and proceed as in the unmodified algorithm. Otherwise, we iterate through the bits of `root_predicates`, compute the local rank of the corresponding join predicate, and proceed with the predicate with the smallest rank.

Query Hypergraphs

Sometimes predicates refer to more than two relations, e.g. $R.x + S.y < T.z$. In principle, a predicate may refer to an arbitrary number of relations although predicates with more than tree relations are rare. Queries of this type can be represented by *hypergraphs*—a generalization of (ordinary) graphs. A hypergraph $G = (V, E)$ consists of a set of nodes V and a set of (hyper)edges $E \subseteq 2^V$. Note that ordinary graphs are special hypergraphs where each edge has cardinality two.

Hypergraphs are an ideal framework to represent queries. For a given query the *query hypergraph* is defined as follows. The nodes of the query hypergraph are the base relations involved in the query and for each predicate in the query there is an edge containing the relations the predicate refers to. Note that edges of cardinality one correspond to primary selections whereas edges of cardinality greater than one correspond to joins. In the following we briefly sketch how our algorithms can be modified to work with query hypergraphs.

The generalization is straightforward. We use a similar representation for graphs and hypergraphs. Nodes are numbered 0 to $n-1$ and edges are numbered 0 to $p-1$. Sets of nodes as well as sets of edges are represented by bit vectors. The edges in E are stored as a table `nodes` such that `nodes[i]` is the bit vector of all nodes in the i-th edge. The auxiliary arrays `p0`, `p1`, `p2`, `incidence_list`, and `parallel_edges` can be used as in `DP-OPT-C`, except for some minor changes in the initialization of the arrays. Consequently, there is no change in the auxiliary arrays `component` and `component_id` and the procedure `dfs`. What changes is the following. Since an edge can refer to more than two relations the removal of a bridge may increase the number of connected components by more than one. Hence, for each considered join predicate (bridge in the hypergraph) we additionally have to enumerate the partitions of the connected components in the hypergraph with the join predicate removed. Suppose $C_1, \ldots, C_k$ are the bit vectors of the nodes in each of the k connected components and C_i is the connected component to which the bridge h belongs. We can use depth-first-search to determine the connected components $C'_1, \ldots, C'_l$ of C_2 resulting from the removal of the bridge h. Then we enumerate all non-trivial partitions of $\{C'_1, \ldots, C'_l\}$ in graycode order. The blocks of the partition are combined with the blocks of the partitions of $\{C_1, \ldots, C_{i-1}, C_i, \ldots, C_k\}$. For example, consider the bridge h in the query hypergraph in Figure 4.11 [12]. We first enumerate all sets of nodes that correspond to the partitions of the connected components C_1, C_3. For each such set we then enumerate the sets of nodes that correspond to the non-trivial partitions of C'_1, C'_2, C'_3.

Complexity The time complexity increases at most by a factor of 2^t where t denotes the maximal number of relations to which a predicate in the query refers to. The space complexity does not change.

4.2.7 Performance Measurements

In this section we investigate how our dynamic programming algorithms perform in practice. Note that we give no results of the quality of the generated plans, since they are optimal. The only

[12] encircled sets of relations denote edges

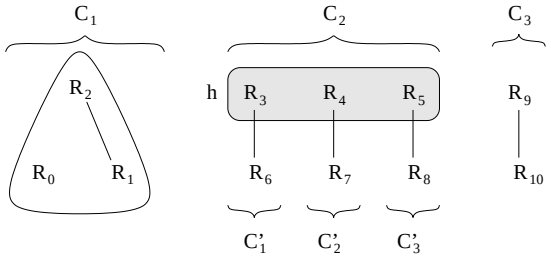


Figure 4.11: Upon removal of edge h the connected component C_2 decomposes into the connected components C_1', C_2' and C_3'

remaining question is whether the algorithm can be applied in practice since the time bounds seem quite high. We begin with algorithm Optimal-Bushy-Tree(R,P) without enhancements. The table below shows the timings for random queries with $n = 10$ relations and query graphs with 50% of all possible edges having selectivity factors less than 1. Join predicates are cheap ($c_p = 1$) whereas selection predicates are expensive ($c_p > 1$). All timings in this section were measured on a lightly loaded workstation with Alpha 21164 Processor (533MHz) and 256 MB of main memory. Run times are averages over a few hundred runs with randomly generated queries.

$n = 10$ *sel* :	0	1	2	3	4	5	6	7	8	9	10
time [s]:	0.01	0.01	0.02	0.03	0.06	0.10	0.18	0.30	0.57	1.21	2.31

The next table shows the timings for queries with join predicates only. Note that the number of join predicates does not influence the runtimes since joins and cross products are treated alike.

n :	$1-4$	5	6	7	8	9	10	11	12	13
time [s]:	< 0.0001	0.0001	0.0003	0.0007	0.0015	0.0037	0.0110	0.0329	0.1014	0.3466
n :	14	15	16	17	18					
	1.329	4.203	13.96	52.45	162.1					

Let us now consider algorithm Optimal-Bushy-Tree(R,P) enhanced by splitting join predicates.

The next table shows running times for random queries on $n = 5$ relations. We varied the number of predicates from 0 to 16, the fraction of selections was held constant at 50%. For this example we used a modified "near-uniform" random generation of queries in order to produce connected join graphs whenever possible, i.e. we did only produce queries where the number of components equals $\max(n - p + s, 1)$.

$n = 5$ p:	$0-6$	7	8	9	10	11	12	13	14	15	16
time [s]:	< 0.001	0.001	0.003	0.006	0.011	0.022	0.041	0.080	0.160	0.310	0.692

Finally, we considered queries with 10 predicates, out of which 5 are selections. By varying the number of relations from 1 to 10, we obtained the following timings.

$p = 10$, $s = 5$ n :	1	2	3	4	5	6	7	8	9	10
time [s]:	-	0.001	0.002	0.005	0.011	0.029	0.075	0.262	0.828	2.764

The following timings concern the third algorithm which uses structural information from the query graph. The next table shows running times for random "near-uniform" queries on $n = 5$ relations. The number of predicates varies from 0 to 13, the fraction of selections was held constant at 50%. Note the break-even point with respect to the second algorithm. For smaller number of predicates (here $p \leq 9$) the second algorithm is faster than the third and for larger values (here $p > 9$) the third algorithm beats the second algorithm. This is not surprising since the few predicates lead to many disconnected subgraphs and hence to many cross products. In this case

analyzing the connected components is too much overhead. On the other hand, many predicates lead to more connected subgraphs from which the third algorithm benefits.

$n = 5$ p :	$0-4$	5	6	7	8	9	10
time [s]:	< 0.001	0.001	0.002	0.002	0.005	0.008	0.014
p :	11	12	13	14	15	16	
time [s]:	0.025	0.054	0.087	0.174	0.295	0.636	

4.2.8 Dynamic Programming: Variants and Applicability

Dynamic Programming is a general mathematical optimization principle applicable to many discrete optimization problems. All these optimization problems have one thing in common—their cost functions are *decomposable* [Min86]. Basically, the notion of decomposability comprises the following two properties. First, the cost function (say f) can be formulated as a recurrence[13] involving one or more calls to f with simpler arguments ("separability"). Second, in such a recurrence the function that combines the different recursive function applications is monotonically non-decreasing with respect to the arguments where f is called recursively ("monotonicity"). For example, the function

$$f(x_1, \ldots, x_n) = h(f(x_1, \ldots, x_{n-1}), x_n)$$

is decomposable, provided that h is monotonically non-decreasing with respect to the first argument. The basis of Dynamic Programming is the Principle of Optimality which stipulates[14] that every optimal solution can only be formed by partially optimal solutions. The validity of the Principle of Optimality ensures that we can state a recurrence that computes the cost of an optimal solution (together with the optimal solution itself).

The relation "is a subproblem of" defines a partial order P among all subproblems. The valid enumeration orders of the subproblems are exactly the linear extensions of P. Since we cannot a priori decide which of these enumeration orders is the best, it seems reasonable to chose one that can be generated efficiently—as we did in our approach. On the other hand, there are well-known methods that use cost information to direct the enumeration of subproblems. Two examples are the A* and IDA* [15] algorithms from the area of heuristic search [Pea84]. These algorithms can be used to compute an optimal path in a directed graph. Unlike bushy trees, computing optimal left-deep trees can be stated as a problem of computing an optimal path in a graph. As far as we know, there is no generalization of A* search to non-path problems.

Although A* is optimal (and IDA* is asymptotically optimal) among all informed search methods [Pea84], these algorithms have some drawbacks. First, compared to dynamic programming, A* has the additional overhead of keeping track of (potentially very large) priority queues and hash tables and of computing non-trivial lower bounds to future costs. Second, the efficiency of A* crucially depends on the quality of the used lower bound. Although IDA* does without priority queues and hash tables, it considers slightly more nodes than A*. Nevertheless, for larger problems IDA* usually outperforms A*. We made a few experiments that indicated that even if we use the excellent lower bound $c \in (0, 1) \times$ "the true future cost", A* essentially considers as many subproblems as with the trivial lower bound 0 (corresponding to best-first search).[16] The number of considered subproblems decreases only if c is close to 1. At least for join ordering problems such lower bounds seem unattainable. However this experiment strongly indicates that cost-based pruning (e.g. as proposed by Graefe in his top-down dynamic programming algorithm

[13] or a system of recurrences

[14] Actually, the following weaker formulation would be sufficient unless we are to enumerate *all* optimal solutions: "There exists an optimal solution that is only formed by partially optimal solutions".

[15] Iterative Deepening A*

[16] These results are not new. In [Poh70] it is shown that for constant relative errors the running times of A* and IDA* are exponential in the depth of the graph.

or as sugested by Vance and Maier [VM96]) does not seem to reduce the number of considered subproblems substantially.

Dynamic programming algorithms are usually classified as bottom-up or top-down approaches [17]. However, the names "bottom-up" and "top-down" are somewhat misleading. They refer to the order in which the subproblems are generated and not in which they are solved. In both approaches the subproblems are solved bottom-up. One advantage of the top-down approach is that new subproblems can be generated and optimized dynamically during the optimization process and multiple, overlapping problems can be optimized together. For example, similar as in the volcano optimizer generator [GM93], we could mix top-down dynamic programming with a transformation-based approach. In each step of the recursive optimization algorithm, first all applicable transformations are applied in turn (with provisions to avoid reverse transformations and duplicate subproblems) and then all possible decompositions into subproblems are enumerated and the subproblems optimized recursively. Another advantage of the top-down approach is that useless subproblems are automatically avoided in the computation. For example, suppose a query gives raise to N different subproblems and for each subproblem we have to account for k different physical properties $p_1, \ldots, p_k$, where each property p_i can assume k_i different values, we have to enumerate $N * \prod_{i=i}^{k} k_i$ combinations of subproblems and physical properties. Often, not all these combinations really make sense. While the top-down approach seems to be more flexible in dealing with subproblems it is also less efficient in the enumeration of subproblems [Van98]. Although cost bounds can be used in both the top-down as well as the bottom-up version of dynamic programming they seem to have only little effect. Note that in the top-down approach it is possible to use upper bounds on the costs to "prune" whole subproblems (which is not directly possible in the bottom-up approach), this rarely is effective too because all the subproblems are highly interdependent. It is much more beneficial to use upper bounds for saving some cost computations.

Allmost all references in the literatur refer to this "traditional" version of the dynamic programming [SAC+79, GD87, OL90, GM93, STY93, VM96, CYW96]. However, it is sometimes necessary to use a slightly more general version of dynamic programming named *partial order dynamic programming* in [GHK92].
Let us first describe a straightforward generalization of the traditional dynamic programming scheme. In the traditional scheme, costs are numeric values that define a total ordering among all plans for a subproblem. Now suppose that we cannot always decide whether one plan is better than another plan, i.e. all we have is a plan comparison relation which defines a partial order among plans. Obviously, when the Priciple of Optimality holds it is save to discard all suboptimal plans for subproblems (since an optimal plan can cannot contain suboptimal subplans). In other words, instead of computing one optimal plan for each subproblem we compute all plans that do not prove to be suboptimal. As to the implementation, all that changes is that we now have to deal with lists of plans instead of single plans.
The idea behind partial order dynamic programming is the following. Suppose our comparison relation $\prec_1$ does not fulfill the Principle of Optimality. Now, if we can find a weaker comparison relation $\prec_2$[18] that does fulfill the Priciple of Optimality, we can use the above described generalization with $\prec_2$ to compute a set of potentially optimal plans from which we determine the true optimal plan using $\prec$.

With traditional dynamic programming the cost function computes a scalar, numeric cost value and keeps track of the best plan. So does partial order dynamic programming, but now the cost function may be non-scalar. A typical representation of costs are *resource vectors* (or *resource descriptors*). A resource vector is a tuple where the components quantify the usage of a certain resource. For example the components of a resource vector used in parallel query processing might be the time to complete a query (elapsed time), the time when the first tuple is produced, the

[17] though different orders are conceivable
[18] i.e. suboptimality with respect to $\prec_2$ implies suboptimality with respect to $\prec_1$

sum of processing times for each processor (total work), the total amount of buffer space used, disk access times, network communication costs, etc. Sometimes additional parameters, which do not reflect physical resources, are incorporated into resource vectors such that the resource vector of a plan can now be computed in terms of the resource vectors of its suplans (depending on the properties of the subproblem). Obviously, if one plan uses less resources than another plan the plan using less resources is superior. This defines a natural (partial) order amoung resource vectors that we can use to eliminate all sub-optimal plans for a subproblem: plan p_1 is superior to plan p_2 if the resource vector of p_1 is "component-wise" smaller than the resource vector of p_1. More on partial order dynamic programming can be found in [GHK92].

Chapter 5

Summary and Outlook

5.1 Summary

This thesis investigated several subclasses of the problem of computing optimal processing trees for conjunctive queries. The subproblems arise from restricting certain problem features, like the shape of the processing trees, the operators allowed in processing trees and the shape of the join graph. This gives rise to a number of interesting problem classes with different complexities. We were able to settle the complexity status and/or develop new efficient optimization algorithms that outperform the previously known algorithms.

For the problem of computing optimal left-deep trees with cross products for chain queries two algorithms were devised. The first algorithm produces the optimal plan, but we could not prove that it has polynomial time. The second algorithm runs in polynomial time, but we could not prove that it produces the optimal result. A conjecture is stated, which implies that both algorithms run in polynomial time and produce an optimal plan.

Another important type of queries are acyclic queries with expensive predicates. By modeling selections as joins, we showed that the algorithm of Ibaraki and Kameda [IK84] can be applied to the problem. The resulting algorithm runs in polynomial time but has the following limitations. Expensive selections can only be placed on the path from the left-most leaf to the root of the tree, and the cost function has to fulfill the ASI property.

Although the space of bushy trees is larger than the space of left-deep trees it may contain considerably cheaper plans [OL90]. The question that immediately arises, is whether we can expect polynomial algorithms for this more general problem. We proved that the problem is NP-hard independent of the query graph. Hence, unless P=NP, there is no hope of computing optimal bushy trees in polynomial time. Consequently, we focused on the general problem of computing optimal bushy trees for general queries with expensive joins and selections. Although several researchers have proposed algorithms for this problem, apart from [CS97] all approaches turned out to be wrong. We presented three formally derived correct algorithms for the problem. Our algorithms can handle different join algorithms, split conjunctive predicates, and exploit structural information from the join graph to speed up the computation. The time and space complexities are analyzed carefully, and efficient implementations based on bitvector arithmetic are presented.

5.2 Future Research

In object-oriented and object-relational database systems, the *map operator* (χ-operator) is used for dereferenciation. The highest optimization potential is with path expressions. Each dot in a path expression is translated to a map operator. Converting map operators into joins may lower or rise the cost. Hence, this decision should be cost-based within a dynamic programming algorithm. Map operators can also be used to factorize calls of expensive user-defined functions.

Another starting point of future work is to incorporate the factorization of common subexpressions into our dynamic programming algorithms. So far, factorization can only be performed *after* the plan has been generated. Since factorization changes the costs, the new, factorized plans may no longer be optimal.

A major weakness of traditional optimization techniques is that they assume cost functions to be accurate. Unfortunately, this is not the case. Cost functions only approximate the true costs and they rely on statistical parameters which are often inaccurate. Parameters are either estimated through database statistics or they are just "guessed" if not sufficient information is available. For example, parameters that refer to run-time resources cannot be estimated and have to be given "typical" values. Errors in the estimations are due to approximations or outdated statistics. In [IC91] it has been shown that errors grow exponentially with the number of joins in a query. It is not clear how this amplification of errors influences the results of query optimization. Hence, extending our dynamic programming approaches to deal with these types of errors would be a possible direction of future research. Current approaches can be classified into three classes.
Dynamic optimization [CG94, GW89] tries to react to inaccurate parameter estimations by evaluating these parameters along the execution of the query and comparing them to the estimated parameters. If estimated and computed parameters differ considerably, appropriate action can be taken. Either the query is (partially) re-optimized or—if alternative plans are available—the execution switches to a different execution plan.
Another approach to dynamic optimization is the competition model of Antoshenkov [Ant93, Ant96]. Here, different plans are executed concurrently. If a plan shows better that the others, the execution of the sub-optimal plans is stopped.
Parametric optimization [Gan98, INSS92, CG94, GK94] attempts to identify several execution plans; each one is optimal for a certain region of the parameter space. Parametric optimization is usually combined with dynamic optimization. Unfortunately, parametric optimization seems to work only for one or two parameters and affine (or special non-linear) cost functions [Gan98].
Least expected cost (LEC) optimization [CHS99] is a technique to optimize a cost function involving stochastic parameters (i.e. random variables). Some parameters—like the buffer space available in a concurrent environment—cannot be estimated at compile time. However,the parameter may follow a certain distribution if averaged over many executions. Provided that such a distribution exists and can be determined by the statistics module, dynamic programming can be modified to compute an execution plan with minimum expected[1] cost among all execution plans. A LEC plan will be cheaper than any "specific cost" plan if costs are averaged over many executions. Furthermore, if the cost function is concave, it will always be cheaper. Although least expected cost optimization is an interesting approach, it suffers from the yet unsatisfactory solved problem of efficiently handling and transforming probability distributions with sufficient accuracy.

[1] expectation taken with respect to distribution of the statistical parameter

Chapter 6

Zusammenfassung

In dieser Dissertation werden bestimmte Probleme der algebraischen Anfrageoptimierung untersucht. Für die betrachteten Problemklassen werden neue, effiziente Algorithmen entwickelt und deren Komplexitätsstatus bestimmt (polynomial/NP-hart). Die Teilprobleme werden dabei im Wesentlichen durch die Operatoren der Anfrage, die Form des Anfragegraphen, die Form der Auswertungsbäume, die in den Auswertungsbäumen erlaubten Operatoren und die Kostenfunktion bestimmt.

In der Anfrageoptimierung werden oft linkstiefe Auswertungsbäume verwendet, da diese verschiedene Vorteile bieten. Zum einen ist der Suchraum bei linkstiefen Bäumen erheblich kleiner als bei verzweigten Bäumen und zum anderen führen linkstiefe Bäume zu gewissen Vereinfachungen im Anfrageoptimierer und im Laufzeitsystem. Eine weitere Reduktion des Suchraum erzielt man durch die Vermeidung von Kreuzprodukten, was allerdings zu höheren Auswertungskosten führen kann. Ferner spielt die Struktur des Joingraphen eine wichtige Rolle. Das Standardverfahren zur Berechnung optimaler Auswertungsbäume basiert auf Dynamischem Programmieren. Leider berücksichtigt dieses Verfahren weder Eigenschaften der Kostenfunktion noch die Struktur des Joingraphen. Algorithmen, die spezielle Eigenschaften des Problems nutzen, sind weitgehend unbekannt. Lediglich für den Fall azyklischer Anfragegraphen, linkstiefer Auswertungsbäume ohne Kreuzprodukte und ASI-Kostenfunktionen ist ein dedizierter polynomialer Algorithmus bekannt, der auf einem Job-Sequencing Algorithmus basiert.

Kapitel 3 widmet sich zunächst der Berechnung optimaler linkstiefer Auswertungsbäume. Eine der einfachsten Problemklassen stellen die kettenförmigen Anfragen dar, wie sie z.B. bei der Umsetzung von Pfadausdrücken in objekt-orientierten und objekt-relationalen Datenbanksystemen auftreten. Kapitel 3.1 widmet sich dem Problem der Berechnung optimaler linkstiefer Auswertungsbäume mit Kreuzprodukten für kettenförmige Anfragegraphen und ASI-Kostenfunktionen. Es werden zwei neue Algorithmen entwickelt, wobei der erste Algorithmus polynomiale Zeitkomplexität hat, während der zweite Algorithmus korrekt ist (d.h. niemals suboptimale Auswertungsbäume liefert). Obwohl beide Algorithmen in der Praxis polynomiale Laufzeit aufweisen und identische Ergebnisse liefern, konnte die Äquivalenz beider Algorithmen bisher nicht gezeigt werden.

In modernen Datenbanksystemen kann der Benutzer eigene Funktionen und Prädikate definieren, welche bei Anfragen verwendet werden können. Da benutzerdefinierte Prädikate hohe Kosten verursachen können, spricht man auch von teuren Prädikaten. Die vielfach verwendete Heuristik, Selektionen so früh wie möglich zu berechnen, hat im Fall teurer Selektionsprädikate keine Gültigkeit mehr. Joins und teure Selektionen müssen gemeinsam optimiert werden.

In Kapitel 3.2 wird für das Problem azyklischer Anfragegraphen, linkstiefer Auswertungsbäume, teurer Selektions- und Joinprädikate und ASI-Kostenfunktionen ein effizienter Algorithmus ent-

wickelt, der auf dem schon erwähnten Job-Sequencing Algorithmus basiert. Unter der Voraussetzung, dass man die Menge der direkt auf die Basisrelationen anzuwendenden, teuren Selektionen erraten kann, besitzt der Algorithmus polynomiale Zeitkomplexität.

Obwohl die Menge der verzweigten Auswertungsbäume erheblich größer ist als die Menge aller linkstiefen Bäume, ist es dennoch vorteilhaft, diesen größeren Suchraum zu betrachten, da verzweigte Bäume aufgrund ihrer besseren Balancierungseigenschaften die Kosten deutlich reduzieren können. Ähnlich verhält es sich bei den Kreuzprodukten. Zusätzliche Kreuzprodukte vergrößern zwar den Suchraum, können aber ebenfalls zu billigeren Auswertungsbäumen führen. Aus diesem Grund widmet sich Kapitel 4 der Berechnung optimaler verzweigter Auswertungsbäume mit möglichen Kreuzprodukten. Es stellt sich die Frage, ob es für den Fall verzweigter Bäume und Kreuzprodukte spezielle Teilprobleme gibt, die in polynomialer Zeit lösbar sind. In Kapitel 4.1 wird gezeigt, dass dies nicht der Fall ist. Das Problem ist NP-hart, unabhängig von der Form des Anfragegraphen.

Kapitel 4.2 widmet sich dem allgemeinen Problem der Berechnung verzweigter Auswertungsbäume für beliebige Anfragegraphen und teurer Selektions- und Joinprädikate. Für dieses Problem werden mehrere auf dynamischem Programmieren basierende Algorithmen entwickelt. Die Komplexität der Algorithmen wird analysiert und effiziente Implementierungen werden beschrieben. Die Algorithmen sind in der Lage, verschiedene Joinalgorithmen zu berücksichtigen, konjunktive Prädikate zu trennen und die Struktur des Joingraphen auszunutzen.

Appendix A

An Implementation

In this section we present an implementation of the algorithm from section 4.2.4. The implementation reflects the first version of the algorithm, i.e. we perform a depth-first search for each edge of the induced subgraph to determine whether the edge is a bridge or is part of a cycle. In case the edge is a bridge we also determine the components the bridge connects. The C code for the algorithm is listed below[1].

```
 1  struct dfsFrame {
 2    bitvector startNode; // starting node
 3    bitvector lastNode;  // node last visited
 4    int mark;            // label<mark <=> node not marked
 5    nodeType& adj;       // adjacency list
 6  };
 7
 8
 9  ....
10
11
12  int optimize(int n,int p,int sl, double* f_i, float* c_i, int* rel1, int* rel2){
13
14    struct nodeType {
15      int lb;        // label
16      bitvector nb;  // neighbor nodes
17    };
18
19    struct splitting {
20      bitvector rel_left;   // relations in the left subgraph
21      bitvector pred_left;  // predicates in the left subgraph
22      bitvector pred_right; // predicates in the right subgraph
23    };
24
25
26    bool       flag;
27    int        i,j,noc,label;
28    bitvector  i2,j2,k,cut,h,k,l,ll,i2,nn,pp,pd1,pd2,r,r1,r2,u,iv,sels,rest,best_pred_so_far,
29               delta,left_block,right_block,left_relations,right_relations,left_predicates,
30               right_predicates,the_component,new_predicates,current_pred,root_predicates,
31               relations,predicates;
32    float      best_cost_so_far,cost1,dcost1;
33
34    splitting  best_split_so_far;  // partition in left and right subproblems
35    nodeType*  adjList[nn];        // adjacency list
36    dfsFrame   (*st)[n];           // stack for dfs
37
38    // allocate tables
39    predicate* pred = new predicate[pp];
40    float** size = new (float*)[nn];
41    float** cost = new (float*)[nn];
```

[1] with C++-style comments

```
42      splitting** split = new (splitting*)[nn];
43      bitvector** root = new (bitvector*)[nn];
44      bitvector** component = new (bitvector*)[nn];
45      char ** no_components = new (char*)[nn];
46
47      for(i=0; i<nn; i++){
48        size[i] = new costType[pp];
49        cost[i] = new costType[pp];
50        split[i] = new splitting[pp];
51        root[i] = new bitvector[pp];
52        component[i] = new bitvector[pp];
53        no_components[i] = new char[pp];
54      }
55      float* sz = new float[n];
56      bitvector* p0 = new bitvector[nn];
57      bitvector* p1 = new bitvector[nn];
58      bitvector* p2 = new bitvector[nn];
59
60      bitvector* parallel_edges = new bitvector[pp];
61      nodetype* adjList = new nodetype[nn];
62      char* component_id = new char[nn];
63      bitvector* block = new bitvector[nn<<1];
64      bitvector* graycode = new bitvector[nn];
65
66      // initialize everything
67      sp = 0;      // stack pointer for st
68      label = 1;   // initial label for dfs
69      nn = 1<<n;
70      pp = 1<<p;
71      sels = (1<<sl)-1;
72      for (i=0; i<p; i++) {    // initialize pred[]
73        j2 = 1<<i;
74        pred[j2].sel = f_i[i];
75        pred[j2].cost = c_i[i];
76        pred[j2].rel1 = rel1[i];
77        pred[j2].rel2 = rel2[i];
78      }
79      for (i=0, i2=1; i<p; i++, i2<<=1)
80        parallel_edges[i2] = 0;        // initialize parallel_edges[]
81      for (i=0, i2=1; i<p; i++, i2<<=1)
82        for (j=i+1, j2=1<<(i+1); j<p; j++, j2<<=1)
83          if (pred[i2].rel1==pred[j2].rel1 &&
84              pred[i2].rel2==pred[j2].rel2) {
85            parallel_edges[i2] |= j2;
86            parallel_edges[j2] |= i2;
87          }
88
89      component[0][0]     = 0;
90
91      for (i=0, i2=1; i<n; i++, i2 <<= 1 ){
92        p0[i2] = p1[i2] = 0;
93        size[i2][0] = n_i[i];
94        cost[i2][0] = scanCosts(n_i[i]);
95        root[i2][0] = 0;
96        split[i2][0].rel_left = 0;
97        split[i2][0].pred_left = 0;
98        split[i2][0].pred_right = 0;
99      }
100     for (i=0, i2=1; i<p; i++, i2 <<= 1) {
101       r1 = 1<<pred[i2].rel1;
102       r2 = 1<<pred[i2].rel2;
103       p0[r1] = p0[r1]^i2;
104       p0[r2] = p0[r2]^i2;
105       p1[r1] = p1[r1]|i2;
106       p1[r2] = p1[r2]|i2;
107     }
108     p0[0] = p1[0] = p2[0] = 0;
109
110     // enumeration of subproblems
111
112     for (r=1; r<nn; r++){  // enumerate subsets of relations
113       r1 = r&-r;
114       r2 = r^r1;
115       p0[r] = p0[r1]^p0[r2];  // initialize p0[],p1[],p2[]
116       p1[r] = p1[r1]|p1[r2];
117       p2[r] = p1[r]^p0[r];
118       flag = true;
119       l = 0;
120       while (flag || l){  // enumerate subsets of predicates
121         flag = false;
```

```
122         k = l&-l;         // predicate with smallest index
123         best_cost_so_far = INFINITY;  // initialize best plan so far
124         best_pred_so_far = 3;
125         best_split_so_far.rel_left   = 0;
126         best_split_so_far.pred_left  = 0;
127         best_split_so_far.pred_right = 0;
128         r1 = r&-r;      // relation with smallest index
129         r2 = r^r1;      // relation with second smallest index
130         if (l)          // are there any predicates?
131           size[r][l] = size[r][l^k] * pred[k].sel;   // yes, use recurrence over predicates
132         else            // no predicates!
133           if (r2)       // are there at least two relations in r?
134             size[r][l] = size[r1][0]*size[r2][0];    // yes, use recurrence over relations
135         for(i=0, i2=1; i<n; i++, i2<<=1){
136           adjList[i2].nb = 0;   // initialize incidence list for dfs
137           adjList[i2].lb = 0;
138         }
139         ll = l;
140         k  = ll&-ll;
141         while (ll){   // iterate through predicates in l
142           r1 = 1<<pred[k].rel1;
143           r2 = 1<<pred[k].rel2;
144           if (r1 != r2) {
145             adjList[r1].nb |= r2;   // update incidence list
146             adjList[r2].nb |= r1;
147           }
148           ll ^= k;
149           k = ll&-ll;
150         }
151         // perform dfs to compute the connected component containing relation r&-r
152         component[r][l] = dfs(r&-r, 3, label, adjList);
153         label++;   // update label
154         relations  = r;
155         predicates = l;
156         i = 0;
157         i2 = 1;
158         // determine remaining connected components by table look-up...
159         while (block[i2] = component[relations][predicates]){  // more components?
160           rest = block[i2];
161           iv = rest & -rest;
162           while (rest){ // iterate through rest
163             component_id[iv] = i;  // iv is in connected component number i
164             rest ^= iv;
165             iv = rest & -rest;
166           }
167           relations ^= block[i2];           // update relations
168           predicates = l & p2[relations];   // update predicates
169           i++;
170           i2 <<= 1;
171         }
172         noc = i;     // number of connected components
173         if (noc>1){  // is the graph disconnected?
174           // yes, use graycode to enumerate all subsets of connected components
175           gray = 0;
176           bitvector left_block = 0;  // left block of partition
177           bitvector right_block;     // right block of partition
178           for(i=1, i< 1<<noc; i++){  // enumerate graycode
179             last = gray;           // last graycode
180             gray = i ^ (i>>1);     // new graycode
181             delta = last ^ gray;   // bit that has changed
182             left_block ^= block[delta];    // new left block of partition
183             right_block = r ^ left_block;  // new right block
184             left_predicates  = l & p2[left_block];  // update left predicates
185             right_predicates = l & p2[right_block]; // update right predicates
186             if (left_block != r)   // neither left block nor right block should be empty
187               dcost1 = cost[left_block][left_predicates] +
188                        cost[right_block][right_predicates]; // cost of subtrees
189             // cost of the cross product between left and right block
190             cost1 = dcost1 + cpCosts(size[left_block][left_predicates],
191                                      size[right_block][right_predicates]);
192             if (cost1<best_cost_so_far) {  // cheaper than best_cost_so_far?
193               best_cost_so_far = cost1;    // yes, update!
194               best_pred_so_far = 0;
195               best_split_so_far.rel_left = left_block;
196               best_split_so_far.pred_left = left_predicates;
197               best_split_so_far.pred_right = right_predicates;
198             }
199           }
200         }
201       }
```

```
202       root_predicates = l & sels;  // possible selection predicates
203       current_pred = root_predicates & -root_predicates;
204       while (root_predicates){   // iterate through all possible selection predicates
205         new_predicates = l ^ current_pred;
206         dcost1 = cost[r][new_predicates];    // cost of subtree below the selection
207         // cost of the selection operator
208         cost1 = dcost1 + selCosts(size[r][new_predicates],
209                                   size[r][l],pred[current_pred].cost);
210         if (cost1<best_cost_so_far) {  // is plan cheaper?
211           best_cost_so_far = cost1;    // yes, update!
212           best_pred_so_far = current_pred;
213           best_split_so_far.rel_left = 0;
214           best_split_so_far.pred_left = 0;
215           best_split_so_far.pred_right = 0;
216         }
217         root_predicates ^= current_pred;
218         current_pred = root_predicates & -root_predicates;
219       }
220       root_predicates = l & ~sels;   // possible join predicates
221       current_pred = root_predicates & -root_predicates;
222       while (root_predicates){   // iterate through possible join predicates
223         if (!(parallel_edges[current_pred] & l)) { // is it a primary join predicate?
224           r1 = 1<<pred[current_pred].rel1;
225           r2 = 1<<pred[current_pred].rel2;
226           // temporarily remove current_pred from adjList
227           adjList[r1].nb ^= r2;
228           adjList[r2].nb ^= r1;
229           // determine connected component that contains current_pred
230           relations = dfs(1<<pred[current_pred].rel1, 3, label, adjList);
231           label++;
232           // undo removal of current_pred
233           adjList[r1].nb |= r2;
234           adjList[r2].nb |= r1;
235         } else {   // no, a secondary join predicate
236           // removal of current_edge does not change adjList!
237           // determine connected component that contains current_pred
238           relations = dfs(1<<pred[current_pred].rel1, 3, label, adjList);
239           label++;
240         }
241
242         if (relations == block[1<<component_id[1<<pred[current_pred].rel1]]){
243           // it is a secondary join!
244           new_predicates = l ^ current_pred;
245           dcost1 = cost[r][new_predicates];   // cost of subtree below selection operator
246           cost1 = dcost1+selCosts(size[r][new_predicates],   // selection costs
247                                   size[r][l],pred[current_pred].cost);
248           if (cost1<best_cost_so_far) {  // new plan cheaper?
249             best_cost_so_far = cost1;    // update!
250             best_pred_so_far = current_pred;
251             best_split_so_far.rel_left = 0;
252             best_split_so_far.pred_left = 0;
253             best_split_so_far.pred_right = 0;
254           }
255         }
256         else
257           {    // can be a primary or a seconary join...
258             if (noc==1){    // special case: graph connected
259               left_relations =  relations;            // relations on the left side
260               right_relations = r ^ left_relations;   // relations on the right side
261               left_predicates = p2[left_relations] & l;   // left predicates
262               right_predicates = p2[right_relations] & l; // right predicates
263               dcost1 = cost[left_relations][left_predicates]       // subtree costs
264                      + cost[right_relations][right_predicates];
265               cost1 = dcost1 + jCosts(size[left_relations][left_predicates],  // join costs
266                                       size[right_relations][right_predicates],
267                                       size[r][l],pred[current_pred].cost);
268               if (cost1<best_cost_so_far) {    // new plan cheaper?
269                 best_cost_so_far = cost1;      // update!
270                 best_pred_so_far = current_pred;
271                 best_split_so_far.rel_left = left_relations;
272                 best_split_so_far.pred_left = left_predicates;
273                 best_split_so_far.pred_right = right_predicates;
274               }
275               // now interchange role of left and right side
276               cost1 = dcost1 + jCosts(size[right_relations][right_predicates],  // join costs
277                                       size[left_relations][left_predicates],
278                                       size[r][l],pred[current_pred].cost);
279               if (cost1<best_cost_so_far) {  // cheaper?
280                 best_cost_so_far = cost1;    // update!
281                 best_pred_so_far = current_pred;
```

```
282                best_split_so_far.rel_left = right_relations;
283                best_split_so_far.pred_left = right_predicates;
284                best_split_so_far.pred_right = left_predicates;
285              }
286            }
287            else  // more than one connected component
288              {   // compute connected component containing current_pred
289                the_component = 1<<component_id[1<<pred[current_pred].rel1];
290                // swap with last connected component
291                swap(block[the_component],block[1<<(noc-1)]);
292                noc--;   // pretend as if last component does not exist
293                gray = 0;        // initialize graycode
294                left_block = 0;  // initialize left block of partition
295                for(i=0; i< 1<<noc; i++){  // enumerate all subsets of connected components
296                  last = gray;           // last graycode
297                  gray = i ^ i>>1;       // new graycode
298                  delta = last ^ gray;   // bit that has changed
299                  left_block ^= block[delta];   // new left block of partition
300                  right_block = r ^ block[1<<noc] ^ left_block;  // new right block
301                  left_relations = left_block | relations;       // update left
302                  right_relations = r ^ left_relations;
303                  left_predicates = p2[left_relations] & l;
304                  right_predicates = p2[right_relations] & l;
305                  dcost1 = cost[left_relations][left_predicates]   // subtree cost
306                    + cost[right_relations][right_predicates];
307                  cost1 = dcost1 + jCosts(size[left_relations][left_predicates], // join cost
308                                          size[right_relations][right_predicates],
309                                          size[r][l],pred[current_pred].cost);
310                  if (cost1<best_cost_so_far) {  // new plan cheaper?
311                    best_cost_so_far = cost1;    // update!
312                    best_pred_so_far = current_pred;
313                    best_split_so_far.rel_left = left_relations;
314                    best_split_so_far.pred_left = left_predicates;
315                    best_split_so_far.pred_right = right_predicates;
316                  }
317                  // interchange role of left and right side
318                  cost1 = dcost1 + jCosts(size[right_relations][right_predicates],  // join cost
319                                          size[left_relations][left_predicates],
320                                          size[r][l],pred[current_pred].cost);
321                  if (cost1<best_cost_so_far) {  // new plan cheaper?
322                    best_cost_so_far = cost1;    // update!
323                    best_pred_so_far = current_pred;
324                    best_split_so_far.rel_left = right_relations;
325                    best_split_so_far.pred_left = right_predicates;
326                    best_split_so_far.pred_right = left_predicates;
327                  }
328                }
329                swap(block[the_component],block[1<<noc]);  // undo swap of connected components
330                noc++;                                     // restore previous number of components
331              }
332            new_predicates = l ^ current_pred;  // consider possibility of a secondary join
333            dcost1 = cost[r][new_predicates];   // subtree cost
334            cost1 = dcost1 + selCosts(size[r][new_predicates],      // selection cost
335                                      size[r][l],pred[current_pred].cost);
336            if (cost1<best_cost_so_far) {  // new plan cheaper?
337              best_cost_so_far = cost1;    // update!
338              best_pred_so_far = current_pred;
339              best_split_so_far.rel_left = 0;
340              best_split_so_far.pred_left = 0;
341              best_split_so_far.pred_right = 0;
342            }
343          }
344        root_predicates ^= current_pred;        // proceed to next predicate
345        current_pred = root_predicates & -root_predicates;
346      }
347      if (l || r&(r-1)) {
348        cnt++;
349        cost[r][l] = best_cost_so_far;   // store best solution in table
350        root[r][l] = best_pred_so_far;
351        split[r][l] = best_split_so_far;
352      }
353      l = p2[r]&(l-p2[r]);       // switch to the next subset of predicates
354    }
355  }
356
357  printPlan(split,root,nn-1,pp-1,theJoinProblem){   // print optimal plan
358
359    // free arrays
360    delete block;
361    delete component_id;
```

```
362     delete adjList;
363     delete graycode;
364
365     for (k=1; k<nn; k++){
366       delete[] size[k];
367       delete[] cost[k];
368       delete[] split[k];
369       delete[] root[k];
370       delete[] component[k];
371       delete[] no_components[k];
372     }
373     delete[] size;
374     delete[] cost;
375     delete[] split;
376     delete[] root;
377     delete[] component;
378     delete[] no_components;
379     delete[] p0;
380     delete[] p1;
381     delete[] p2;
382     delete[] sz;
383     delete[] pred;
384     delete[] parallel_edges;
385
386   }
387
388   ....
389
390
391    // display optimal plan
392   void print_plan(splitting** split, bitvector** root, bitvector r, bitvector s){
393     if !(s || r&(r-1))    // a base relation?
394       printf("r%d",log2(r));
395     else
396       if (split[r][s].rel_left==0){  // a selection?
397         printf("sigma[p%d])",log2(root[r][s]));
398         print_solution(split,root,r,s^root[r][s],pi);
399         printf(")");
400       } else
401         {
402           if (root[r][s] == 0)       // a cross product?
403             {
404               printf("cross-product(");
405               print_solution(split,root,split[r][s].rel_left,
406                              split[r][s].pred_left,pi);
407               printf(",");
408               print_solution(split,root,r^split[r][s].rel_left,
409                              split[r][s].pred_right,pi);
410               printf(")");
411             }
412           else                    // a join!
413             {
414               printf("join[p%d](",log2(root[r][s]));
415               print_solution(split,root,split[r][s].rel_left,
416                              split[r][s].pred_left,pi);
417               printf(",");
418               print_solution(split,root,r^split[r][s].rel_left,
419                              split[r][s].pred_right,pi);
420               printf(")");
421             }
422         }
423   }
424
425
426   bitvector dfs(bitvector current_node, bitvector last_node,      // depth-first search
427                 int& currentMark, nodetype *adjList, dfsFrame* st){
428
429     dfsFrame frame,newframe;
430
431     bitvector neighbors,nextNode;
432     bitvector block = 0;             // the connected component
433
434     frame.currNode = start;          // current node
435     frame.lastNode = 3;              // node last visited
436     frame.mark = currentMark;        // a label smaller than frame.mark means
437                                      // that the node has not been visited
438     sp=0;
439     st[sp++] = frame;     // push to stack
440
441     while (sp) {
```

```
442       frame = st[--sp];                                // pop from stack
443       if (adjList[frame.currNode].lb < frame.mark) {   // node not visited?
444         adjList[frame.currNode].lb = frame.mark;       // ok, mark as visited
445         block |= frame.currNode;                       // update connected component
446         neighbors = adjList[frame.currNode].nb;
447         nextNode = neighbors & -neighbors;             // first neighbor
448         neighbors ^= nextNode;
449         while (nextNode){                              // consider all neighbors
450           if (nextNode != frame.lastNode ||
451               frame.currNode == frame.lastNode){       // update current node
452             block |= next;
453             newFrame.currNode = next;
454             newFrame.lastNode = frame.currNode;
455             newFrame.mark = frame.mark;
456             st[sp++] = newFrame;                       // push to stack
457           }                                            // switch to next neighbor
458           nextNode    = neighbors & -neighbors;        // next neighbor
459           neighbors ^= nextNode;
460         }
461       }
462     }
463     return block;
464   }
```

We assume that predicates are ordered such that the indices 0 to $sl-1$ are corresponding to selections and predicates with indices sl to $p-1$ are corresponding to joins. The table below lists the variables and functions used in the algorithm. Throughout the table *rel* denotes a bitvector of relations and `pred` a bitvector of predicates.

variables and functions	
`int n`	number of relations
`int p`	number of predicates (joins and selections)
`int sl`	number of selections among the `p` predicates
`float n_i[j]`	cardinality of the j-th base relation
`float f_i[j]`	selectivity of predicate j
`float c_i[j]`	cost factor of predicate j
`int a_i[j],b_i[j]`	indices of the two relations to which the j-th predicate refers (`a_i[j]`$\leq$`b_i[j]`)
`bitvector sels`	bitvector of all selections ($sels = 2^{sl} - 1$)
`float pred[1<<j].sel`	equal to `f_i[j]`
`float pred[1<<j].cost`	equal to `c_i[j]`
`int pred[1<<j].rel1`	equal to `rel1[j]`
`int pred[1<<j].rel2`	equal to `rel2[j]`
`float size[rel][pred]`	cardinality of plans with relations (`rel` and predicates `pred`)
`float cost[rel][pred]`	cost of an optimal plan for (`rel`, `pred`)
`bitvector root[rel][pred]`	root operator of an optimal plan for (`rel`, `pred`)
`bitvector split[rel][pred].rel_left`	relations in the left sub-plan if the root operator is a join—otherwise 0
`bitvector split[rel][pred].pred_left`	predicates in the left sub-plan if the root operator is a join—otherwise 0
`bitvector split[rel][pred].pred_right`	predicates in the right sub-plan if the root operator is a join—otherwise 0
`bitvector component[rel][pred]`	connected component in the join graph induced by (`rel`, `pred`) that contains the relation with smallest index
`bitvector p0[rel]`	predicates incident with an even number of relations in `rel`
`bitvector p1[rel]`	predicates incident with at least one relation in `rel`
`bitvector p2[rel]`	predicates induced by the relations in `rel`
`bitvector parallel_edes[1≪i]`	edges parallel to the edge `i`
`bitvector adjList[r].nb`	neighbors of node `r`
`int adjList[r].lb`	label of node `r` (dfs)
`bitvector block[1≪i]`	relations in the i-th connected component of the subgraph induced by `r`,`l`
`int component_id[r]`	connected component that contains relation `r`
`int log2(k)`	discrete logarithm to base 2
`float scanCosts(size_in)`	scan operator costs
`float cpCosts(size_in1,size_in2)`	cross product operator costs
`float jCosts(size_in1,size_in2,size_out)`	join operator costs
`float selCosts(size_in,size_out)`	selection operators costs

Comments The call to `dfs` in line 152 can be avoided since we can compute directly the relations in the connected component that contains the relation with the smallest index if we make use of the previously computed connected components and induced subgraphs. Suppose we consider a subproblem with relations R and predicates P. Let G be the query graph induced by R and P. Let us denote the relations with smallest and second smallest indices with R_1 and R_2, respectively. The query graph induced by $R' = R - \{R_1\}$ and the query graph induced by $R'' = R - \{R_2\}$ are denoted by G' and G'', respectively. Let us denote the connected components of R' and R'' with C_1 and C_2, respectively. Now, if there is an edge between R_1 and R_2 in G or if C_1 and C_2 overlap then C_1 is connected to C_2 in G and $C = C_1 \cup C_2$. Otherwise, C_1 and C_2 are disjoint and $C = C_1$. The code fragment below replaces lines 152-153 of the above algorithm.

```
r1 = r&-r;                              // relation with smallest index
rel1 = r^bit1;                          // remaining relations in r
if (rel1) {                             // more than one relations in r?
  r2 = rel1&-rel1;                      // relation with second smallest index
  rel2 = r^r2;                          // remaining relations in r2
  pred2 = l&p2[rel2];                   // predicates induced by rel2
  comp1 = component[rel2][pred2];       // conn. component in subgraph (rel2,pred2)
  comp2 = component[rel1][pred1];       // conn. component in subgraph (rel1,pred1)
  if ((l&p2[r1|r2]&~sels)||             // is r1 connected to r2 or
      (comp1&comp2)) {                  // do comp1 and comp2 overlap?
    pred1 = l&p2[rel1];                 // predicates induced by rel1
    component[r][l] = comp1|comp2;      // take the union of comp1 and comp2
  } else                                // r1 not connected to r2!
    component[r][l] = comp1;            // result is comp1
} else
  component[r][l] = r;                  // special case: r contains one relation
```

Bibliography

[Ant93] G. Antoshenkov. Dynamic query optimization in Rdb/VMS. In *Proc. of the Int. Conference of Data Engineering (ICDE)*, pages 538–547, Vienna, Austria, April 1993.

[Ant96] G. Antoshenkov. Dynamic optimization of index scans restricted by booleans. In *Proc. of the Int. Conference of Data Engineering (ICDE)*, pages 430–440, New Orleans, Lousiana, USA, February 1996.

[Ben82] J. Bentley. *Writing Efficient Programs.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1982.

[Ben00] J. Bentley. *Programming Pearls, 2nd edition.* Addison-Wesley, Reading, MA, USA, 2000.

[CBB+97] R.G.G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade, editors. *The Object Database Standard : ODMG 2.0.* Morgan Kaufman Publishers, San Mateo, CA, USA, 1997.

[CD92] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, pages 383–392, San Diego, CA, USA, June 1992.

[CG94] R.L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, pages 150–160, Minneapolis, Minnesota, USA, June 1994.

[CGK89] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for $\mathcal{LDL}$. In *Proc. of the Int. Conference on Very Large Data Bases (VLDB)*, pages 195–203, Amsterdam, The Netherlands, August 1989.

[Cha98] S. Chaudhuri. An overview of query optimization in relational systems. In *Proc. of the ACM SIGMOD/SIGACT Conference on Principles of Database Systems (PODS)*, pages 34–43, Seattle, Washington, USA, June 1998.

[CHS99] F. Chu, J.Y. Halpern, and P. Seshadri. Least expected cost query optimization. In *Proc. of the ACM SIGMOD/SIGACT Conference on Principles of Database Systems (PODS)*, Philadelphia, Pennsylvania, USA, May 1999.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, USA, 1990.

[CM95] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *Proc. of the Int. Conference on Database Theory (ICDT)*, pages 54–67, Prague, Czech Republic, 1995.

[CS93] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proc. of the Int. Conference on Very Large Data Bases (VLDB)*, pages 529–542, Dublin, Ireland, 1993.

[CS96] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *Proc. of the Int. Conference on Very Large Data Bases (VLDB)*, pages 87–98, Bombay, India, 1996.

[CS97] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. Technical Report MSR-TR-97-03, Microsoft Research, Advanced Technology Division, One Microsoft Way, Redmond, WA 98052, USA, 1997.

[CYW96] M.-S. Chen, P. S. Yu, and K.-L. Wu. Optimization of parallel execution for multi-join queries. *IEEE Transactions on Data and Knowledge Eng.*, 8(3):416–428, June 1996.

[Feg97] L. Fegaras. Optimizing large OODB queries. In *Proc. of the Int. Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 421–422, Montreux, Switzerland, December 1997.

[Gan98] S. Ganguly. Design and analysis of parametric query optimization algorithms. In *Proc. of the Int. Conference on Very Large Data Bases (VLDB)*, pages 228–238, New York City, New York, USA, August 1998.

[GD87] G. Graefe and D.J. DeWitt. The EXODUS optimizer generator. In *Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, pages 160–172, San Francisco, CA, USA, May 1987.

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

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability—A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA, USA, 1979.

[GK94] S. Ganguly and R. Krishnamurthy. Parametric query optimization for distributed databases based on load conditions. In *Proc. of the 1994 Int. Conf. on Management of Data (COMAD)*, 1994.

[GKP89] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics. A Foundation For Computer Science*. Addison-Wesley, Reading, MA, USA, 1989.

[GLSW93] P. Gassner, G. Lohman, K. Schiefer, and Yun Wang. Query optimization in the IBM DB2 family. *Data Engineering Bulletin*, 16(4):4–18, December 1993.

[GM93] G. Graefe and W.J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proc. of the Int. Conference of Data Engineering (ICDE)*, pages 209–218, Vienna, Austria, April 1993.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[GW89] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, pages 358–366, Portland, Oregon, USA, May 1989.

[HC93] F.K. Hwang and G.J. Chang. Enumerating consecutive and nested partitions for graphs. DIMACS Technical Report 93-15, Rutgers University, 1993.

[Hel94] J.M. Hellerstein. Practical predicate placement. In *Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, pages 325–335, Minneapolis, Minnesota, USA, May 1994.

[Hel98] J. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems*, 23(2):113–157, June 1998.

[HS93] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, pages 267–277, Washington, D.C., USA, May 1993.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, USA, 1979.

[IC91] Y.E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, pages 268–277, Denver, Colorado, USA, May 1991.

[IK84] T. Ibaraki and T. Kameda. Optimal nesting for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, 1984.

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

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. of the Int. Conference on Very Large Data Bases (VLDB)*, pages 128–137, Kyoto, Japan, 1986.

[KM94a] A. Kemper and G. Moerkotte. *Object-oriented Database Management: Applications in Engineering and Computer Science.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1994.

[KM94b] A. Kemper and G. Moerkotte. Query optimization in object bases: Exploiting relational techniques. In J.C. Freytag, D. Maier, and G. Vossen, editors, *Query Processing For Advanced Database Systems*, pages 63–94. Morgan Kaufman Publishers, San Mateo, CA, USA, 1994.

[KMPS94] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, pages 336–347, Minneapolis, Minnesota, USA, May 1994.

[KMS92] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object bases. In *Proc. of the Int. Conference on Very Large Data Bases (VLDB)*, pages 79–90, Vancouver, Canada, 1992.

[KRHM95] B. König-Ries, S. Helmer, and G. Moerkotte. An experimental study on the complexity of left-deep join ordering problems for cyclic queries. Technical Report 95-4, Lehrstuhl für Praktische Informatik III, RWTH Aachen, Aachen, Germany, 1995.

[KZ88] R. Krishnamurthy and C. Zaniolo. Optimization in a logic based language for knowledge and data intensive applications. In *Proc. of the Int. Conference on Extending Database Technology (EDBT)*, pages 16–33, Venice, Italy, 1988.

[Law78] E. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Annals of Discrete Mathematics*, 2:75–90, 1978.

[LCW93] H. Lu, H.C. Chan, and K.K. Wei. A survey on usage of SQL. *SIGMOD Record*, 22(4):60–65, 1993.

[ME92] P. Mishra and M.H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[Min86] M. Minoux. *Mathematical Programming. Theory and Algorithms*. Wiley, New York, NY, USA, 1986.

[MS79] C. Monma and J. Sidney. Sequencing with series-parallel precedence constraints. *Mathematics of Operations Research*, 4:215–224, 1979.

[MS92] Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufman Publishers, San Mateo, CA, USA, 1992.

[OL90] K. Ono and G.M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. of the Int. Conference on Very Large Data Bases (VLDB)*, pages 314–325, Bisbane, Queensland, Australia, 1990.

[Pap94] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, USA, 1994.

[Pea84] J. Pearl. *Heuristics*. Addison-Wesley, Reading, MA, USA, 1984.

[Pel97] J. Pellenkoft. *Probabilistic and Transformation based Query Optimization*. Ph.D. Thesis. Amsterdam, The Netherlands, 1997.

[PGLK96] J. Pellenkoft, C.A. Galindo-Legaria, and M.L. Kersten. Complexity of transformation-based optimizers and duplicate-free generation of alternatives. Technical Report CS-R9639, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands, 1996.

[PGLK97a] A. Pellenkoft, C.A. Galindo-Legaria, and M.L. Kersten. The complexity of transformation-based join enumeration. In *Proc. of the Int. Conference on Very Large Data Bases (VLDB)*, pages 303–315, Athens, Greek, 1997. Extended version available as CWI Technical Report CS-R9639.

[PGLK97b] A. Pellenkoft, C.A. Galindo-Legaria, and M.L. Kersten. Duplicate-free generation of alternatives in transformation-based optimizers. In *Proc. of the Int. Symposium on Database Systems for Advanced Applications (DASFAA)*, pages 117–123, Melbourne, Australia, 1997.

[PHH92] H. Pirahesh, J.M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, pages 39–48, San Diego, CA, USA, June 1992.

[Poh70] I. Pohl. First results on the effect of error in heuristic search. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 219–236. Elsevier, New York, 1970.

[SAC+79] P.G. Selinger, M.M Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, pages 23–34, Boston, Massachusetts, USA, May 1979.

[SL95] S.K. Seo and Yoon-Joon Lee. Optimizing object-oriented queries with path predicates: Genetic algorithm approach. In *Next Generation Information Technologies and Systems (NGITS)*, Naharia, Israel, 1995.

[SL96] Sang Koo Seo and Yoon Joon Lee. Applicability of genetic algorithms to optimal evaluation of path predicates in object-oriented queries. *Information Processing Letters*, 58(3):123–128, 1996.

[SM96] W. Scheufele and G. Moerkotte. Optimal ordering of selections and joins in acyclic queries with expensive predicates. Technical Report 96-3, Lehrstuhl für Praktische Informatik III, RWTH Aachen, Aachen, Germany, 1996.

[SM97] W. Scheufele and G. Moerkotte. On the complexity of generating optimal plans with cross products. In *Proc. of the ACM SIGMOD/SIGACT Conference on Principles of Database Systems (PODS)*, pages 238–248, Tucson, Arizona, USA, May 1997.

[SM98] W. Scheufele and G. Moerkotte. Efficient dynamic programming algorithms for ordering expensive joins and selections. In *Proc. of the Int. Conference on Extending Database Technology (EDBT)*, pages 201–215, Valencia, Spain, March 1998. Lecture Notes in Computer Science1377, Springer Verlag, Heidelberg, New York, Berlin, etc.

[Smi56] W.E. Smith. Various optimizers for single-stage production. *Naval Res. Logist. Quart.*, 3:366–373, April 1956.

[SMK97] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal*, 6(3):191–208, 1997.

[SPMK93] M. Steinbrunn, K. Peithner, G. Moerkotte, and A. Kemper. Optimizing join orders. Technical Report MIP-9307, Universität Passau, Germany, 1993.

[SS91] L. Shapiro and A.B. Stephens. Bootstrap percolation, the Schröder numbers, and the n-kings problem. *SIAM Journal of Discrete Mathematics*, 4(2):275–280, May 1991.

[SS96] M. Stillger and M. Spiliopoulou. Genetic programming in database query optimization. In *1st Int. Conf. of Genetic Programming (GP'96)*, Stanford, July 1996.

[STY93] E.J. Shekita, K.-L. Tan, and H.C. Young. Multi-join optimization for symmetric multiprocessors. In *Proc. of the Int. Conference on Very Large Data Bases (VLDB)*, pages 479–492, Dublin, Ireland, 1993.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume Volume II: The New Technologies. Computer Science Press, Rockville, MD, USA, 1989.

[Van95] B. Vance. Fast exhaustive search of large bushy join plans with cartesian products. Working draft, Oregon Graduate Institute of Science & Technology, June 1995.

[Van98] B. Vance. *Join-order Optimization with Cartesian Products.* Phd. thesis, Oregon Graduate Institute of Science and Technology, January 1998.

[VM96] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, pages 35–46, Montreal, Canada, June 1996.

[YKY+91] K. Yajima, H. Kitagawa, K. Yamaguchi, N. Ohbo, and Y. Fujiwara. Optimization of queries including ADT functions. In *Proc. of the Int. Symposium on Database Systems for Advanced Applications (DASFAA)*, pages 366–373, Tokyo, Japan, April 1991.