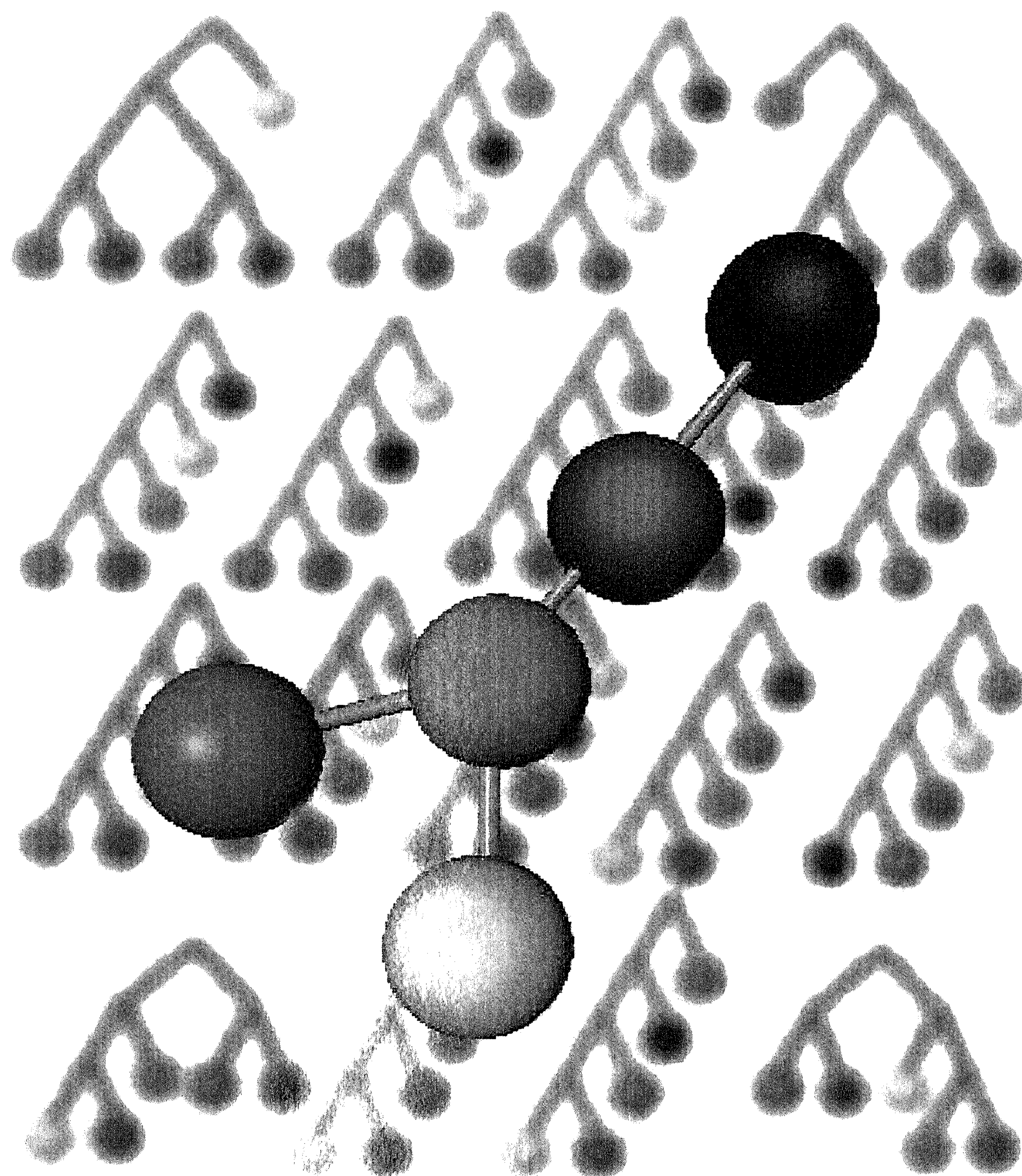
A-33326
=========CWI
Bibliotheek
Centrum voor
Wiskunde &
Informatica
=Amsterdam=

Probabilistic and Transformation based Query Optimization



Jan Pellenkoft

L6

CWI BIBLIOTHEEK
3 0054 00062 2838

Probabilistic and Transformation based Query Optimization

Probabilistic and Transformation based Query Optimization

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor

aan de Universiteit van Amsterdam,

op gezag van de Rector Magnificus

prof. dr J.J.M. Franse

ten overstaan van een door het

college van dekanen ingestelde commissie

in het openbaar te verdedigen

in de Aula der Universiteit

op dinsdag 25 november 1997 te 13.00 uur

door Jan Pellenkoft
geboren te Sleen

Promotor:	prof. dr. M.L. Kersten
Co-promotor:	dr. C.A. Galindo Legaria
Faculteit:	Wiskunde en Informatica

Acknowledgements

I'm glad to express my appreciation here to everybody who has contributed to the research described in this thesis. First of all, I want to thank César Galindo Legaria for introducing me to database research and especially to the area of query optimization. The six months we shared an office at CWI constituted the best start a Ph.D student could ever whish. Your patience and cristal clear explanations of a large variety of subjects made a fruitful research possible. After you left CWI our cooperation continued by e-mail and was indescribably valuable to me. I have learned to know you and your wife as warm and captivating personalities. César and Zandra, Thanks.

I also thank Martin Kersten for offering me a position in his database group at CWI and allowing me to change the subject of my research right from the start. The trust and freedom you gave me made it possible for me to explore the areas which I found most appealing. Together with your enhousiasm it made the four years at CWI a very pleasant and stimulating period.

Also thanks to my colleges — Johan van den Akker, Arno Siebes, Herman Ehrenburg, Marcel Holzheimer, Frank van Dijk, Sunil Choenni, Fred Kwakkel, Chris Thieme and Carel van den Berg — at the AA-department / INS-cluster for their help and interest in my work. I'm particularly honoured that Yannis Ioannidis, Patrick Valduriez, Peter Apers, Peter van Emde Boas, Arno Smeulders and Paul Klint aggreed to be committee members and refereed this thesis.

Verder wil ik mijn vrienden en familie bedanken voor hun steun en belangstelling tijdens de afgelopen vier jaar. In het bijzonder dank aan mijn ouders die me altijd hebben gestimuleerd bij het aangaan van nieuwe uitdagingen.

Alinde, ik weet dat onze eerste jaren in de Bijlmer niet eenvoudig zijn geweest voor jou. Toch ben je me altijd blijven aanmoedigen om het onderzoek voort te zetten. Ik ben er dan ook van overtuigd dat dit proefschrift er niet was geweest zonder jouw liefde en doorzettingsvermogen.

Contents

1 Introduction **1**
1.1 Query processing 2
1.1.1 Query processing architecture 2
1.1.2 Query evaluation plans 2
1.2 Query optimizer 5
1.3 Research problem and objectives 6
1.4 Overview of this thesis 7

2 Join reordering **9**
2.1 Definitions 9
2.1.1 Query graphs and join trees 9
2.1.2 Join tree topologies 11
2.1.3 Structured query graphs 12
2.2 Join tree selection methods 14
2.2.1 Exhaustive search 14
2.2.2 Probabilistic algorithms 15
2.2.3 Non-exhaustive deterministic algorithm 15
2.2.4 Cost models 16

I Theory **17**

3 Complexity of transformation-based join enumeration **19**
3.1 Optimizer Framework 20
3.1.1 MEMO-structure 20
3.1.2 Exploration process 21
3.1.3 Transformation rules 22
3.2 Size of the MEMO-structure 24
3.2.1 Bushy join evaluation orders 24
3.2.2 Left-linear join trees 25

3.3 Duplicates 28
3.3.1 Bushy join trees 30
3.3.2 Linear join trees 32
3.4 Summary 35

4 Duplicate-free join enumeration **37**
4.1 Bushy join trees 38
4.1.1 Completely connected queries 38
4.1.2 Acyclic queries 41
4.2 Linear join trees 44
4.2.1 Completely connected queries 44
4.2.2 Acyclic queries 47
4.3 Experiments 48
4.3.1 Bushy join trees 49
4.3.2 Linear join trees 50
4.4 Summary 51

5 Counting join trees **53**
5.1 Definitions 54
5.1.1 Lists 54
5.1.2 Anchored-list representation 56
5.2 Decomposition and construction of trees 57
5.2.1 Primitive operations 57
5.2.2 Standard decompositions 59
5.3 Counting bushy join trees 63
5.3.1 Recurrence equations 63
5.3.2 Counting standard decompositions 64
5.4 Counting linear join trees 66
5.4.1 Recurrence equations 66
5.4.2 Standard decompositions 67
5.5 Summary 69

6 (Un)ranking and random generation **71**
6.1 Ranking and unranking 71
6.1.1 Mapping trees to integers 71
6.1.2 Local ranking 73
6.1.3 Efficiency of ranking and unranking 79
6.2 Generating random join trees 80
6.2.1 Random join trees 80
6.2.2 Random generation based on unranking 83
6.3 Improved random generation of join trees 83
6.3.1 Efficiency of improved random generation 84

6.4 Summary 85

II Experiments 87

7 **Transformation free optimization** 89
7.1 Definitions 91
7.2 Experimental setup 92
7.2.1 Cost model 92
7.2.2 Database schema, queries and catalogs 92
7.2.3 Cost metrics 93
7.2.4 Performance measure 95
7.3 Cost distribution in search spaces 95
7.3.1 Small search spaces 96
7.3.2 Random sampling 99
7.3.3 Large search spaces 101
7.4 Optimization algorithms 103
7.4.1 Iterative Improvement (II) 103
7.4.2 Simulated Annealing (SA) 104
7.4.3 Transformation free algorithm (TF) 105
7.5 Performance measurements 105
7.6 Summary 107

8 **Hybrid algorithms** 111
8.1 Experimental setup 112
8.1.1 Cost model 112
8.1.2 Database schema, queries and catalogs 113
8.1.3 Transformations rules 114
8.1.4 Factors Considered 114
8.1.5 Performance Characteristics 114
8.2 Results 115
8.2.1 Original Catalogs 116
8.2.2 Enlarged Catalogs 116
8.2.3 Multiple Join Algorithms 119
8.3 Arbitrary set of Transformation rules 120
8.3.1 Experimental results 121
8.4 Hybrid search algorithms 121
8.4.1 Set Based Iterative Improvement 122
8.4.2 Experimental results 122
8.5 Summary 125

9 Conclusions **127**
9.1 Summary 127
9.2 Future research 129

A DBS3 Measurements **131**

B Query graphs **139**

Chapter 1

Introduction

Nowadays, computers play an important role in all parts of life. Many of these computers are used for storing and retrieving large amounts of data in an efficient way. Such systems are called *database systems* and the software that manages the flow of data is called a *database management system* (DBMS). The DBMS facilitates concurrent access to a single database by many users, limits access to data to authorised users only, and recovers from systems failures without loss of data integrity [Ull89a, GV89, Ozk86]. In general, the primary interface to a DBMS is an easy to use high-level query/data-manipulation language — e.g. SQL (Structured Query Language).

Statements in SQL can be issued by the user directly, using a command-line interface, or by an application program. The statements specify *what* answer is expected and not *how* it should be computed. The computation order is generated by the DBMS, specified by a *query evaluation plan* (QEP). In general there are many alternative QEPs that all compute the result required. Each plan, however, has its own *cost* in terms of resource use. It is typically expressed in the number of disk reads and writes and the amount of work for the CPU.

The DBMS's *Query Optimizer* component determines the QEP used for answering a query. It uses a model of the underlying system to select from the large set of candidates an efficient plan as quickly as possible.

This thesis is focused on the techniques employed by the optimizer subsystem to achieve its goal. In Section 1.1 a brief introduction to query processing is given. In Section 1.2 the optimization process is described in more detail. In Section 1.3 the problem statement is exemplified and the research objectives are given. Finally, Section 1.4 provides an overview of this thesis.

1.1 Query processing

Most commercial DBMS systems are based on the relational data-model introduced by E.F. Codd in [Cod70]. An extensive description of this model is beyond the scope of this thesis. We assume the reader is familiar with the basic notions of the relational model and relational algebra. For more information on these topics see [Ull89a, Dat90]. To set the stage we will shortly describe how a query is processed and show the potential cost savings by using efficient QEPs.

1.1.1 Query processing architecture

The process of a query optimizer can be divided into several sub-tasks. Here we follow [GV89] by splitting query processing into three phases, see Figure 1.1.

In the *query decomposition* module, the calculus query — e.g. SQL — is translated into an internal format expressed in relational algebra. The algebraic expression, or operator tree, is the input to the *query optimizer*, which searches for a (quasi) optimal ordering of the algebraic operators. It also selects an algorithm for each operator —e.g. a join operator can be computed using a hash-join, a sort-merge or a nested-loops algorithm — and determines access paths to retrieve data efficiently. The choices are based on the database- and system characteristics, like size of the relations, available memory, etc.

This fully annotated operator tree, the QEP, consists of low level database operations and is passed on to the *query execution* engine where the answer to the query is computed.

1.1.2 Query evaluation plans

Figure 1.2 shows an example relational database which consists of three relations, namely `Person`(ID, CityCode, Name, BirthDate),`City`(CityCode, CityName, Population) and `Car`(OwnerID, type, year). Assume that these relations have the following cardinalities: $|City| = 3000$ tuples, $|Person| = 15 * 10^6$ tuples and $|Car| = 2 * 10^6$ tuples.

Consider the following query:

Give me the name and city information of car owners,.

To illustrate the potentially big difference in cost we analyse two QEPs for this query using a simple *cost model*. In this example the cost of evaluating a QEP is given by the sum of all tuples read and/or written to compute

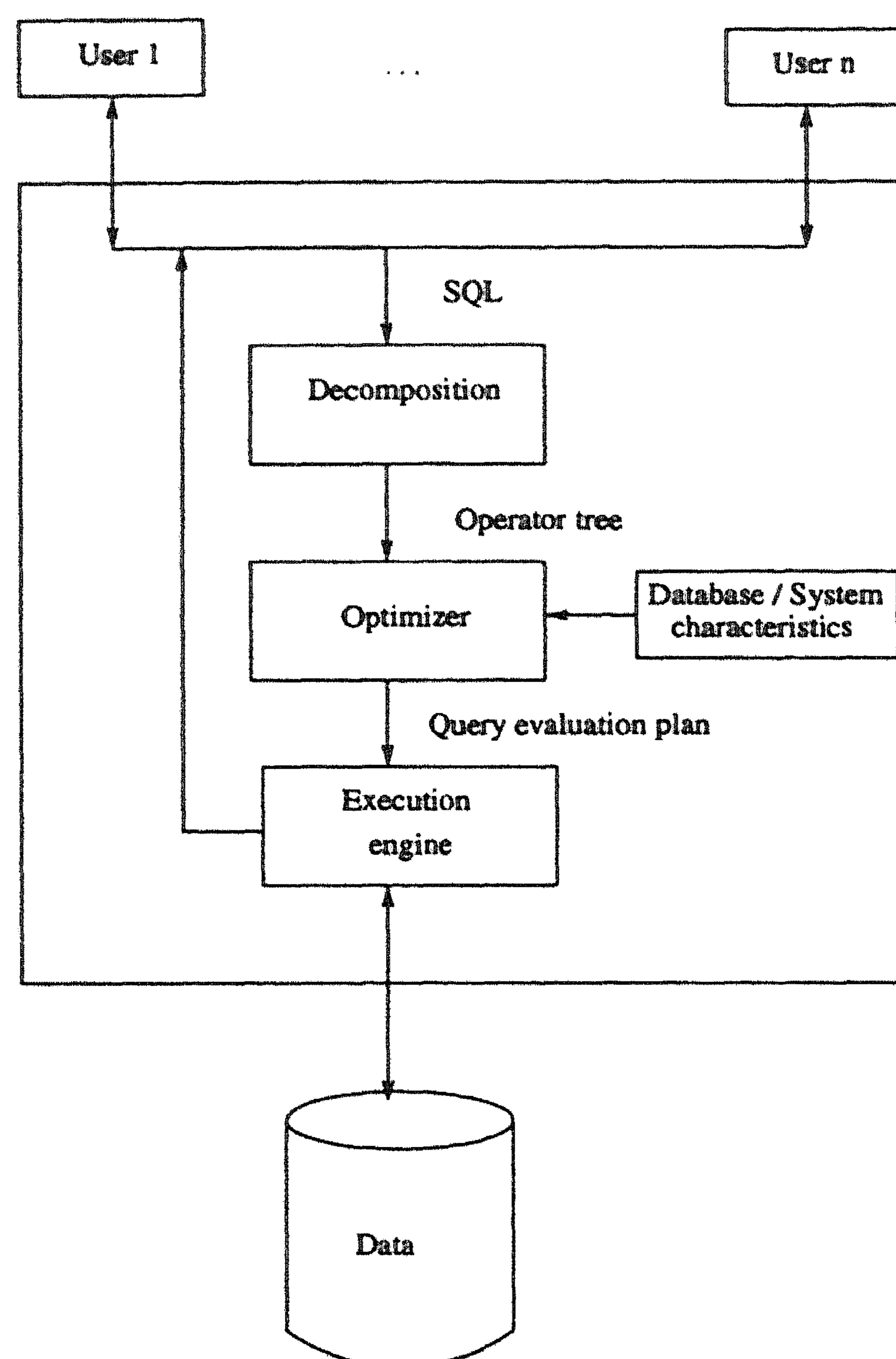


Figure 1.1: Query processing architecture

the result. Furthermore, the cost of *joining* the information of two relations consists of reading the two input relations and writing the result relation[1].

From the set of alternative we consider the two QEPs shown in Figure 1.3, the symbol $\bowtie$ represents the *join*-operator that combines the information of two relations. The QEPs compute the answer as follows.

QEP 1. First, the data from the relations `Person` and `City` is joined to determine for each person the name of the city he or she lives in. This intermediate result is then combined with the relation that holds the identities

[1] For realistic cost models there are many other factors to be taken into account — e.g. availability of different join-algorithms, use of auxiliary access structures, data skew, the effects of caching, etc.

Person:

ID	CityCode	Name	BirthDate
100.6738.678	3265	Jansen	10/12/56
817.6476.345	5897	Vries de	05/03/76
746.6745.634	2526	Pietersen	02/04/78
221.9047.230	7954	Jong de	28/06/95
929.0478.902	2002	Zeilstra	26/09/32
...	...	...	...

City:

CityCode	CityName	Population
1000	Amsterdam	700,000
3000	Rotterdam	600,000
2500	Den Haag	400,000
...	...	...

Car:

OwnerID	Type	Year
432.6544.601	V	1992
817.6476.345	P	1990
221.9047.230	P	1997
921.9403.017	V	1996
...	...	...

Figure 1.2: Example relational database.

of all car owners to compute the final result.

Evaluating QEP 1 requires 49,003,000 reads/writes. The first join requires the relations `Person` and `City` to be read, $15 * 10^6 + 3000$ reads, and the intermediate result to be written, $15 * 10^6$ writes. The size of the intermediate result is $15 * 10^6$, since everybody lives somewhere. Then this intermediate result is read together with the `Car` relation, $15 * 10^6 + 2 * 10^6$ reads. Since each car has one owner the final result contains $2 * 10^6$ entries, which are all written to disk.

QEP 2. By combining the relations `Person` and `Car` an intermediate result containing all car owners is created. This is then subsequently joined to the `City` relation to determine the name of the city for each car owner.

To evaluate QEP 2, 23,003,000 reads/writes have to be performed. The first processing step requires $15 * 10^6 + 2 * 10^6$ reads and $2 * 10^6$ writes. Computing the final result takes another $2 * 10^6 + 3000$ reads and $2 * 10^6$ writes.

This short analysis illustrates that the cost difference between two QEPs can easily be a factor of 2 given a simple cost model. For more realistic queries with more participating relations and an advanced cost model the cost range of QEPs increases even further.

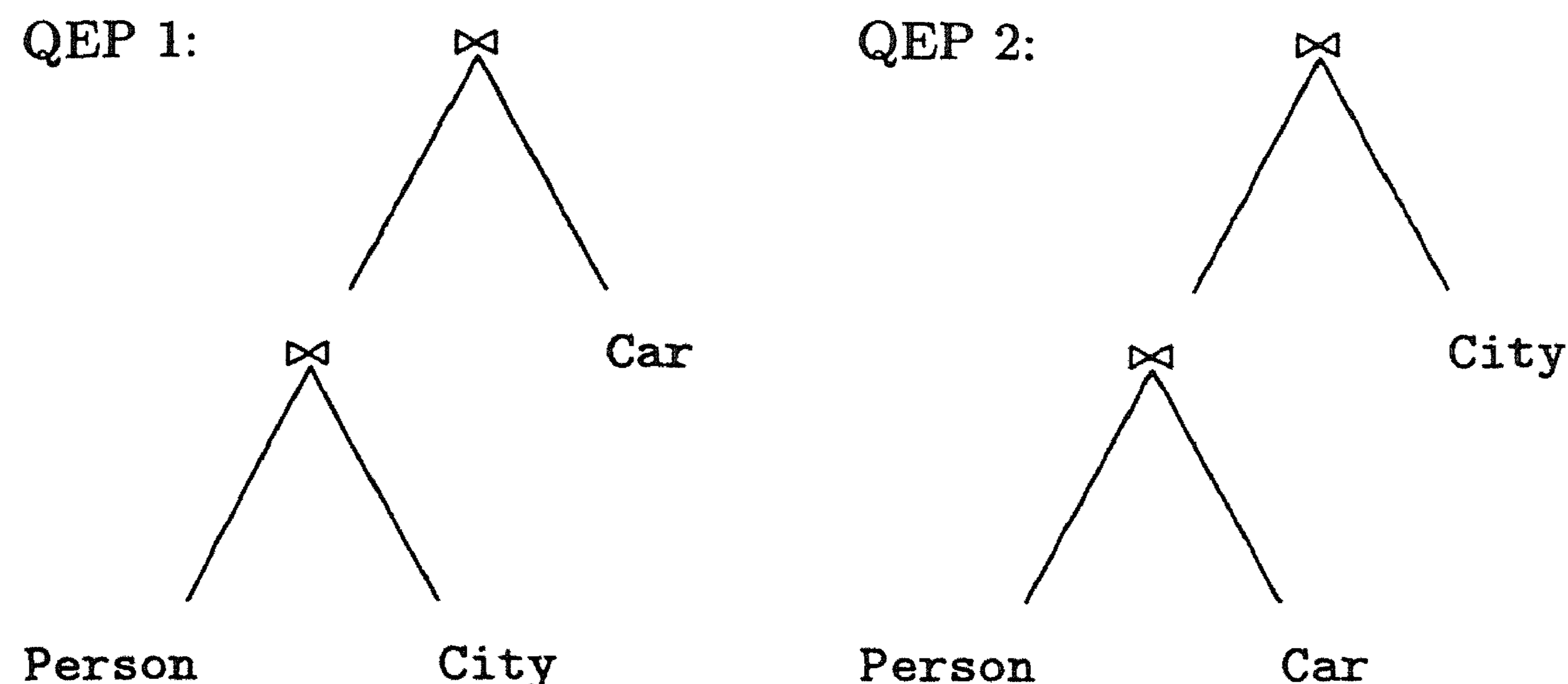


Figure 1.3: Two alternative QEPs

1.2 Query optimizer

Typical optimization goals of an optimizer are *fast response time* and *low resource consumption*. These optimization goals are often conflicting. For example, a QEP which computes the result of a query quickly but requires all available resources (e.g. memory and CPUs) is probably rejected because it would virtually deprive other users from accessing the database.

It has been known for many years that finding good solutions is resource (time) intensive, but can reduce evaluation cost considerably. Clearly, there are trade-offs to be made. Therefore, important properties of a query optimizer are the *quality of the solution* and the *optimization time* — the time it takes to find a good QEP.

As queries get more complex (in terms of the number of relations involved or alternative algorithms for computing an operator) the number of alternative QEPs to consider explodes. The number of alternatives quickly grows in the order of millions, while the difference between the cheapest and most expensive QEP can easily be several orders of magnitude.

As a rule of thumb, for small queries with no more than 4 or 5 relations all QEPs can be generated within a few seconds. In this case the optimization time is often only a fraction of the response time improvement gained. For larger queries, till about 10 relations, an exhaustive search through the alternatives is still feasible, but is often not desired since too much time is spent in optimization.

One of the most time consuming tasks performed by the execution engine is the computation of the join of two relations. To compute the join of N relations, $N-1$ dyadic join operations have to be performed. Since the

size of the relations joined have a big impact on the cost, the order in which all N relations are joined has a big impact on the overall cost of computing the join of N relations.

Unfortunately, finding the optimal join order is an NP-hard problem [IK84, CM95, WM97], while at the same time it is one of the biggest sources of optimization. Only for specific cases, exact and efficient solutions have been found [IK84, KBZ86].

1.3 Research problem and objectives

This thesis is focused on the problem of finding the optimal, or a good, join order and considers several related complexity issues. First, a thorough complexity analysis is given for extensible transformation based optimizers proposed in literature [GD87, GM93, Gra95].

These transformation-based optimizers consist of generating all alternatives reachable from an initial QEP by a set of transformation rules; their estimated cost can be used in choosing one of them. The common view is that the tradeoff of extensible transformation-based optimizers is one of efficiency and extensibility. Since this kind of optimizer is currently used to implement commercial database systems it becomes important to answers the following question:

- *Is it possible to make transformation based optimization as efficient as dynamic programming ?*

Second, although many optimization techniques have been proposed and implemented, the fundamental question of how many alternatives exist for a given join query is yet largely unresolved. Or stated differently:

- *How many join trees exist for a given join query ?*

For combinatorial optimization problems where an exhaustive search is infeasible, probabilistic transformation based algorithms are often used. Two important factors for the performance of these algorithms are random generation and the topology imposed upon the search space by the transformation rules. To identify the impact of randomization on the performance of an optimizer a pure random sampling algorithm is compared to probabilistic transformation based algorithms as developed by [IW87, IK90, SG88]. However, efficient random generation of join trees with a uniform distribution has been a hard problem for many years. So, the third and fourth question to answer are:

- *How can valid join trees be generated efficiently at random with a uniform distribution ?*

- *How does a random sampling algorithm perform compared to probabilistic transformation-based algorithms ?*

Each question raised is addressed in a separate Chapter. The next section gives a more detailed overview of the outline of this thesis.

1.4 Overview of this thesis

This thesis is structured as follows. Chapter 2 describes the primary optimization techniques studied in the research arena and gives definitions of general notions used in this thesis. In Chapter 3 complexity issues for transformation based exhaustive enumeration are derived and in Chapter 4 a technique to reduce the complexity is given, which makes transformation-based join enumeration as efficient as dynamic programming. These last two chapters are based on results previously published in the proceedings of DASFAA'97 and VLDB'97 [PGLK97b, PGLK97a].

Chapter 5 introduces an efficient counting technique for determining the size of search spaces. Based on these results a ranking/un-ranking method for mapping join trees to integers, and visa versa, is developed in Chapter 6. It allows efficient generation of join trees at random with a uniform distribution. Parts of the material presented in Chapter 5 and 6 have been published in the proceedings of ICDT'95 [GLPK95].

In Chapter 7 the random generation of join trees is used to implement a new optimization algorithm, called Transformation Free optimization. The performance of TF is experimentally compared to probabilistic algorithms, like Simulated Annealing an Iterative Improvement, as described in literature. The results published in the proceedings of VLDB'94 [GLPK94] form the basis of Chapter 7.

As a result of the experiments a hybrid algorithm has been implemented and evaluated in Chapter 8. This hybrid algorithm combines the good characteristics of both TF and II. Finally, in Chapter 9 the results obtained in this study are summarized and the directions for future research identified.

Chapter 2

Join reordering

With the introduction of System R [SAC+79] query optimizers became an active research topic within the database community. The common research subject is to improve the Select-Project-Join query. Applying the simple heuristics of ***pushing down selections as much as possible*** and ***perform projections as soon as possible*** reduces the problem of query optimization to finding the right join order. However, it remains a hard combinatorial problem [IK84].

Algorithms for selecting optimal join trees is the focus of this thesis. There are other topics of concern in query optimization, but they are not addressed. Before we describe what has been done with regard to join reordering we will first give several definitions which will be used throughout the rest of this thesis.

2.1 Definitions

2.1.1 Query graphs and join trees

We represent a query by means of a *query graph.* Nodes of such a graph are labeled by relation names, and edges are labeled by predicates. An edge labeled p exists between the nodes of two relations, say R_i, R_j, if p references attributes of R_i and R_j. The *result* of a query graph $G = (V, E)$ is defined as a Cartesian product followed by relational selection: $\sigma_{p_1 \wedge \cdots \wedge p_n}(R_1 \times \cdots \times R_m)$, where $\{p_1, \ldots, p_n\}$ are the labels of edges E and $\{R_1, \ldots, R_m\}$ are the labels of nodes V.

Join trees are used to evaluate queries, instead of the straight definition of product followed by selection given above. A join tree is an operator

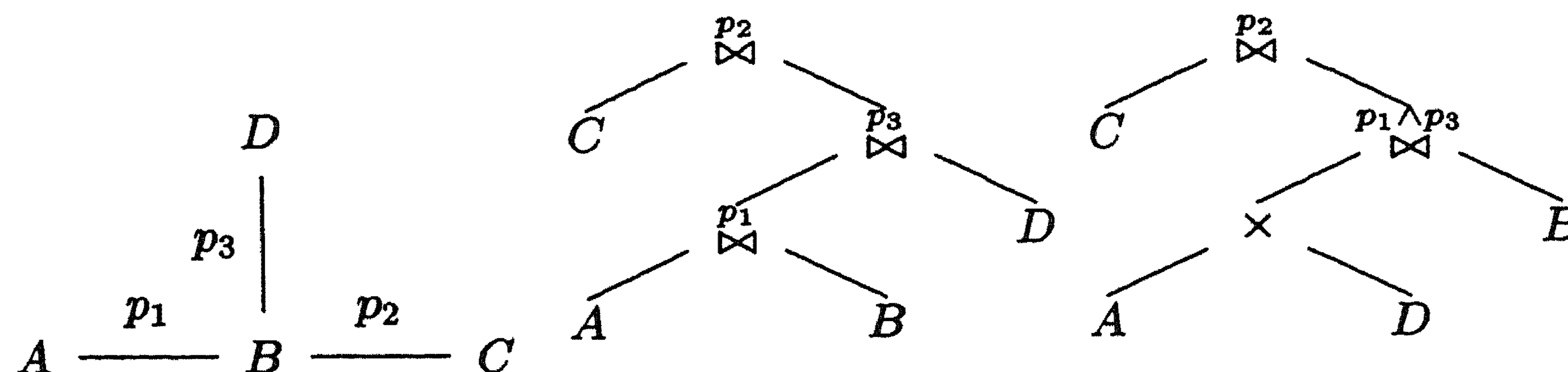


Figure 2.1: Query graph and operator trees.

tree whose inner nodes are labeled by join operators and whose leaves are labeled by relations. The *result* of a join tree is computed bottom-up in the usual way. Join trees also include annotations on the join-algorithm to use —e.g. nested loops, hash, sort merge, etc.— when several are available. Not every binary tree on the relations of the query is an appropriate join tree, because some may require the use of Cartesian products.

The two representations of queries — join trees and graphs — emphasize different aspects of the query. A query graph presents a collection of relations and the predicates that connects them, but it does not impose an evaluation order. A join tree specifies unambiguously the inputs to each operator, and how it is evaluated.

Figure 2.1 shows a query graph, and two operator trees to answer the query. The graph shown corresponds to the query $\{(a, b, c, d) \mid a \in A, b \in B, c \in C, d \in D, p_1(a, b), p_2(b, c), p_3(b, d)\}$, where A, B, C, D are the database relations and p_1, p_2, p_3 are the binary predicates.

The first operator tree requires only relational joins, while the alternative requires a Cartesian product. For a description of relational operators and query graphs, see, for example, [Ull89a, CP85, KRB85]. The reason for using a Cartesian product in the second tree is that we start by combining the information from relations A and D, for which there is no constraining predicate — i.e. there is no edge between A and D in the query graph. Figure 2.2 shows all 6 operator trees for this query involving only joins.

Definition 2.1 *An unordered binary tree T is called a* valid join tree *of query graph $G = (V, E)$ when it satisfies the following recursive definition:*

- *The leaves of T correspond one-on-one with the nodes of G; and*
- *every subtree of T is a join tree for a connected subgraph of G.*

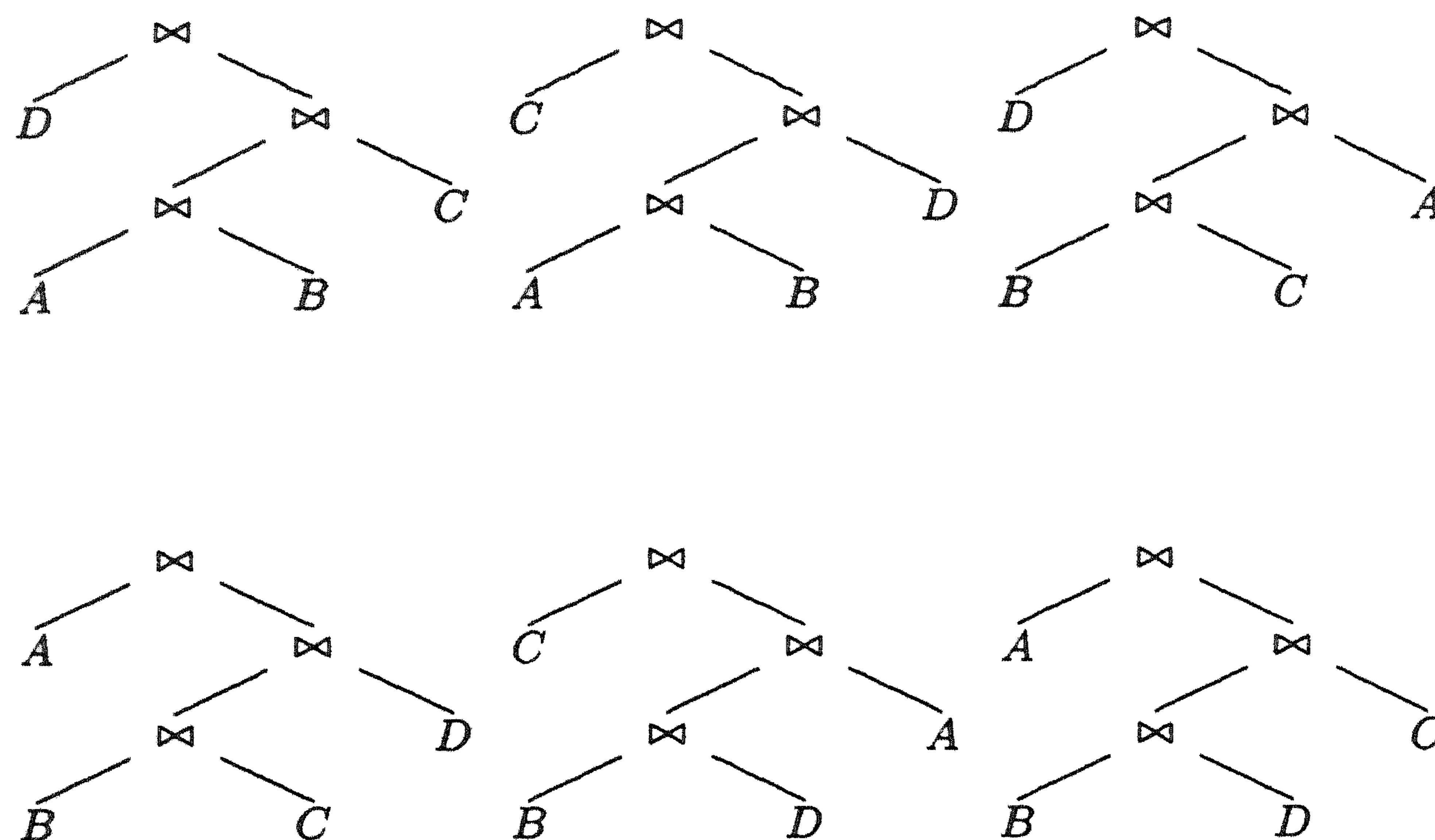


Figure 2.2: All join trees of the query graph.

Operator trees which also contain Cartesian products are called *invalid join tree*. If it is clear from the context then we will refer to valid join trees simply by *join trees*. In general the use of Cartesian products results in operator trees which have a high cost, therefore those operator trees are typically avoided to reduce the number of alternatives. However, special cases exist in which the use of Cartesian products results in the cheapest operator tree [Tay90, Mor92].

2.1.2 Join tree topologies

For join trees we distinguish between *linear* and *bushy* join trees. If for each join operator at least one input is a base relation it is called *linear*, otherwise it is called *bushy*. Accordingly, a set of join trees is called a *bushy search space* if there is no restriction on the topology of the trees. In a *linear search space* the topology of the trees is restricted to linear join trees. Note that the space of linear join trees is a subset of the space of bushy join trees.

From Definition 2.1 it follows that join trees are unordered —i.e. they do not distinguish left from right subtrees. To change an unordered tree of n leaves into an ordered tree a binary choice for each of the $n-1$ internal nodes has to be made. Each unordered tree on n leaves then maps to 2^{n-1} ordered trees.

If there are several implementations available for each join operator - i.e. hash-join, nested-loop or merge-scan, then the total number of trees to be considered increases even further. Suppose we have a query graph in which n relations participate, and there are m alternative implementations available at each join. For each previously unordered tree we then have to consider m^{n-1} ordered trees.

2.1.3 Structured query graphs

For arbitrary query graphs the number of valid join trees is difficult to compute. If, however, the topology of the query graph is "structured", the exact number of join trees can be computed using standard combinatorial methods [LVZ93]. In [Knu68, p. 389] the number of unlabeled binary trees is derived, also known as the Catalan number. For the three query graph topologies— *string, completely connected* and *star* [OL90] —the exact number of unordered linear and bushy join trees is given in Figure 2.3.

Search Space	String query	Completely connected	Star
Linear join trees	2^{n-2}	$n!$	$(n-1)!$
Bushy join trees	$\frac{(2n-2)!}{n!(n-1)!}$	$\frac{(2n-2)!}{(n-1)!}$	$(n-1)!$

Figure 2.3: Number of valid join trees for "regular" query graphs.

String Query. String queries are often used in foreign key traversal and object-oriented environments. The query graph for a string query of n relations has two nodes with degree 1 and $n-2$ nodes with degree 2, see Figure 2.4.

Figure 2.4: A string query

Completely connected Query. The completely connected query, or clique, is not used in real-life applications often. However, it is often used as a test case for its large number of evaluation orders. The query graph for a completely connected query of n relations, has n nodes of degree $n-1$. These types of graphs are called *complete-* or $n-1$ *regular* graphs. See Figure 2.5 for a completely connected graph on 4 nodes.

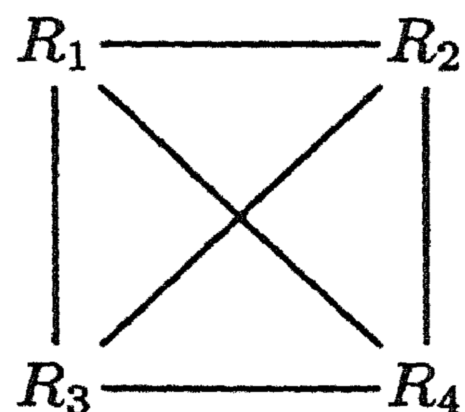


Figure 2.5: A completely connected query on 4 nodes

Star Query. The star query is often found in OLAP applications. The query graph of a star query of n relations, has $n-1$ nodes with degree 1 and one central node with degree $n-1$. See Figure 2.6 for a star on 5 relations. For a star query all valid join trees consists of join operators of which at least one operator is a base relation, therefore the number of bushy and linear join trees are the same.

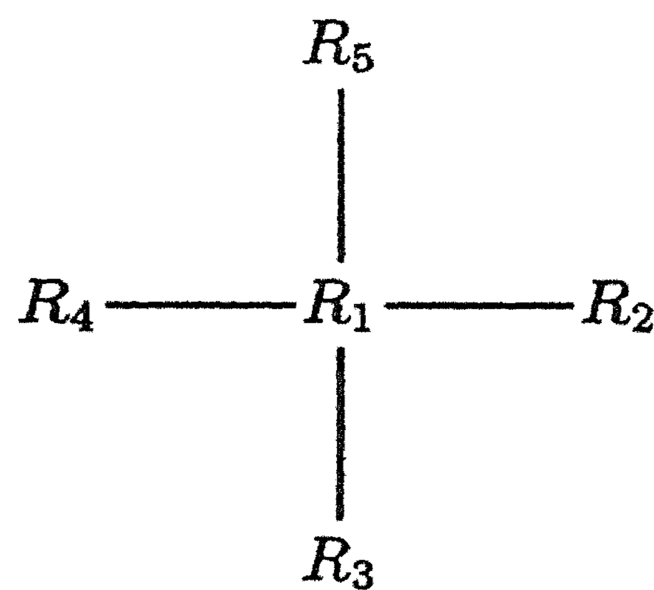


Figure 2.6: A star query

2.2 Join tree selection methods

Finding the "optimal" join tree can be modelled as a search problem. It consists of three components, the *search space*, the *cost model* and the *search strategy*. The search space contains all valid join trees. The cost model is used to annotate each of these trees with a cost and the search algorithm describes how the join trees are explored.

2.2.1 Exhaustive search

If search spaces are not too big ($<$ 10 relations) an exhaustive search is feasible, with the advantage of returning the global optimal join tree according to the cost model. For performing an exhaustive search, either a *dynamic programming* [SAC+79] or an exhaustive application of transformation rules is commonly used [McK93]. These two methods are also called the *bottom-up* and *top-down* approach. Top-down is actually a misnomer, since early in the exploration process alternatives at the bottom of the operator trees are generated as well, due to recursion.

Dynamic programming. In System R [SAC+79] and Starburst [OL90, HCL+90, HP88] dynamic programming is used to find optimal access paths and join trees. The bottom-up algorithm starts by considering the individual relations and adds new relations until all relations are joined. For each subset of joined tables only the cheapest alternative is maintained.

In [VM96] Vance and Maier presented an efficient implementation of an $O(3^n)$ dynamic programming like exhaustive algorithm, which generates all operator trees involving both Cartesian products and join operators.

Transformation-based. Transformation-based exhaustive search can be found in Volcano [Gra89, McK93] as well in Cascades [Gra95] and operates as follows. First, all available transformation rules are applied on the initial operator recursively. Then, the transformation rules are applied on all of the newly generated join trees. This process is repeated until no new join trees are generated. To determine which join trees have already been generated, an efficient structure is used for storing the (partially) explored search space. See Section 3.1.2 for a detailed description of the generation algorithm.

2.2.2 Probabilistic algorithms

In very large search spaces, it is often infeasible to search for the globally optimal join tree. Such spaces are typically the play ground of probabilistic algorithms. Instead of finding the best join tree, the goal is to avoid the worst join trees. Probabilistic algorithms, like *Iterative Improvement* (II) [SG88], *Simulated Annealing* (SA) [IW87] and their variations *Toured Simulated Annealing* (TSA) [LVZ93] and *Two Phase Optimization* (2PO) [IK90] rewrite join trees in order to generate alternatives. These rewrites are based on the algebraic properties of the join operator, i.e. commutativity and associativity.

The probabilistic algorithms start by generating an arbitrary initial join tree. Then, transformation rules are applied to derive new join trees. Depending on the cost of the new join tree and the characteristics of the search algorithm, this new join tree is accepted as the *current solution* on which transformation rules are applied again. This process is repeated until a stopping condition is met. For example, the optimization can stop after a fixed time, or if the cost improvement drops below a certain threshold.

The algorithm TSA performs SA several times, each time with a new initial starting point. The 2PO algorithm first runs II for a short period, the result of the first phase is the initial state for the second SA phase in which the final result is fine tuned. In Chapter 7, the SA and II algorithms are described in more detail.

In [SG88] the following variants were studied also: *Perburtation Walk* (PW) and *Quasi-random Sampling* (QS). The PW algorithm fits within the algorithm framework described above. However, while keeping track of the join tree with the lowest cost, each join tree generated is unconditionally accepted as the new current state. The QS algorithm is similar to PW, but instead of moving to adjacent join trees a new "random" join tree is generated. The join trees are not generated with a uniform distribution and the algorithm is therefor called quasi-random.

2.2.3 Non-exhaustive deterministic algorithm

In [IK84], an efficient algorithm was presented which generates optimal linear join trees for acyclic queries when block-wise nested-loop join algorithms are used. The algorithm presented in [KBZ86] relaxed the use of a specific join algorithm. However, there are still some restrictions on the cost function of the join algorithm. And unfortunately, not all join algorithms comply to it — i.e sort-merge.

Using the KBZ algorithm as a basic building block, [SI92a] propose their AB algorithm which relaxes most of the restrictions of the original KBZ

algorithm by introducing randomness and spanning trees. This makes the AB algorithm probabilistic and increases its complexity.

2.2.4 Cost models

All optimization algorithms need to estimate the cost of (partial) solutions in order to determine which solutions are good. During the optimization many solutions are considered and for all of them a cost is estimated. The estimated cost must be as accurate as possible in order to obtain good solutions. In the design of a cost model a tradeoff between accuracy and speed has to be made.

Another issue related to query optimization and cost models is the fact that a database changes over time and that there might be some time between the optimization of a join tree and its evaluation. A join tree might be "optimal" at optimization time while it can be sub-optimal at execution time due to changes in the system's characteristics or the database contents.

The design of robust and accurate cost models is an important issue and a research area by itself [Sha86, Ull89b]. In this thesis, the cost model is considered to be a black box which, given an evaluation order, returns a cost. The "optimal" solution found by an optimization algorithm is always with respect to the cost model.

Part I

Theory

Chapter 3

Complexity of transformation-based join enumeration[1]

To generate the complete space of alternatives using transformation rules, the naive algorithm is to maintain a set of *visited* join trees. All transformation rules are applied on join trees visited, adding the results to the set if they are new. When no new join trees can be generated, we have explored the complete search space (provided the set of transformation rules is complete). In general, the same join tree can be derived through different sequences of transformation rules, leading to *duplicates*. Every time a duplicate is found, the time to generate it and then to find it in the set of visited join trees is part of the overhead of a naive transformation-based search algorithm.

Duplicates are not an issue for optimization strategies that explore only a small fragment of the search space, especially if the join trees are generated probabilistically, because it is unlikely that the same join tree be generate twice [SG88, IK90, IK91, GLPK94] . However, for optimizers that generate the complete space of alternatives, dealing with duplicates is crucial.

How frequently are we generating duplicates? How expensive is the overhead? Consider the following simple graph model to get a sense of the magnitude of the problem. The number of duplicates generated depends on the size of the search space, n, and the number of neighbors, b_i, of each state s_i (a state s_j is a neighbor of s_i if there is a transformation rule that generates s_j from s_i). Trying each transformation results in generating:

[1]Parts of this chapter have been published in the *Proceedings of the International Conference on Very Large Databases, Athens, 1997* [PGLK97a]

$\sum_{i=1}^{n} b_i$ states. Assuming the number of neighbors for each state is the same ($b = b_1 = \ldots = b_n$) we get $b * n$ generated states. Since there are only n states, the number of duplicates generated is $n * (b - 1)$. Only 1 out of every b states generated is new, and $(1 - \frac{1}{b})$ of the states generated — i.e. most of them as b increases — are duplicates. A considerable efficiency improvement can be achieved by avoiding the generation of those duplicates. We refine our analysis of duplicates later on, for a more realistic optimization framework.

The remainder of this chapter is organized as follows. Section 3.1 describes the relevant parts of the optimizer framework we adopted from the Volcano system [GM93]. Then, Section 3.2 describes the complexity of transformation based join enumeration and Section 3.3 identifies and quantifies the problem of duplicates. Section 3.4 concludes with a summary.

3.1 Optimizer Framework

Before the complexity of transformation based join enumeration is addressed we will first explain the relevant parts of a Volcano-style optimizer. A key component in these optimizers is the *MEMO-structure*, introduced in [GM93], that efficiently stores information about all alternatives explored.

3.1.1 MEMO-structure

The main idea of the MEMO-structure is to avoid replication of subtrees by using *shared copies* only [GCD+94]. It is organized as a network of *equivalence classes* (or simply classes). Each class is a set of operators which all generate the same (intermediate) result. The inputs for the operators are classes which can be interpreted as "any operator of that class can be used as input".

In Figure 3.1 a simplified MEMO-structure is shown which encodes two alternatives for joining the tree relations A, B and C. The two alternative join trees encoded are $(A \bowtie B) \bowtie C$ and $(B \bowtie A) \bowtie C$. Class ABC consists of a single join operator whose left input is an operator of class AB and whose right input is an operator of class C. For joining the relations A and B there are two alternatives, $A \bowtie B$ or $B \bowtie A$, for Class C there is only one alternative so the total number of join trees encoded is two.

In the sequel of this chapter a shorter representation of a MEMO-structure is used in which we omit the classes which reference a single relation. Also the classes are labeled by the relations they combine and instead of using arrows a class is referenced by its label. For example, the

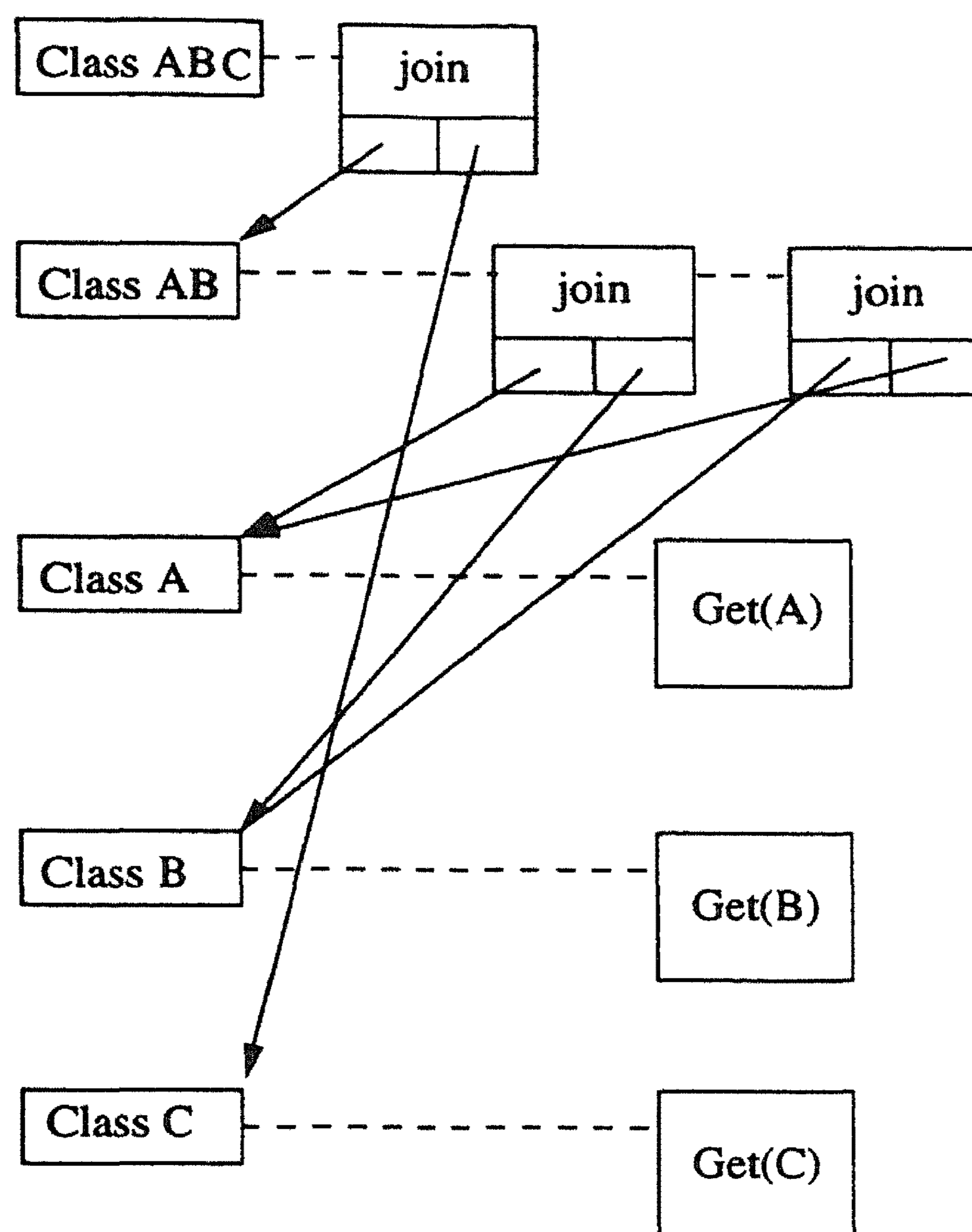


Figure 3.1: A simplified Memo-structure containing two alternatives for 'A join B join C'.

MEMO-structure for the 12 alternatives of a query whose fully connected graph is $Q = \{A - B, A - C, B - C\}$ is shown in Figure 3.2. It has 4 equivalence classes, namely "abc", "ab", "bc", "ac", with the first class containing 6 join operators. The first join operator of class "abc" has as input the classes "ab" and "c".

An operator tree is obtained from a MEMO-structure by choosing a specific operator at each level. For example, the tree to solve query Q that is shown in Figure 3.3, is extracted from the MEMO-structure in Figure 3.2 by *always selecting the first operator from a class.*

3.1.2 Exploration process

A complete MEMO-structure — encoding a complete space — is constructed by recursively exploring the roots of operator trees, starting with an initial join evaluation order. Exploring an operator is done by exhaustively applying all transformation rules to generate all alternatives. This

abc $=[ab] \bowtie [c]; [a] \bowtie [bc]; [b] \bowtie [ac]; [c] \bowtie [ab]; [bc] \bowtie [a]; [ac] \bowtie [b]$
ab $=[a] \bowtie [b]\ ; [b] \bowtie [a]$
bc $=[b] \bowtie [c]\ ; [c] \bowtie [b]$
ac $=[a] \bowtie [c]\ ; [c] \bowtie [a]$

Figure 3.2: The complete MEMO-structure for $\{A-B, A-C, B-C\}$

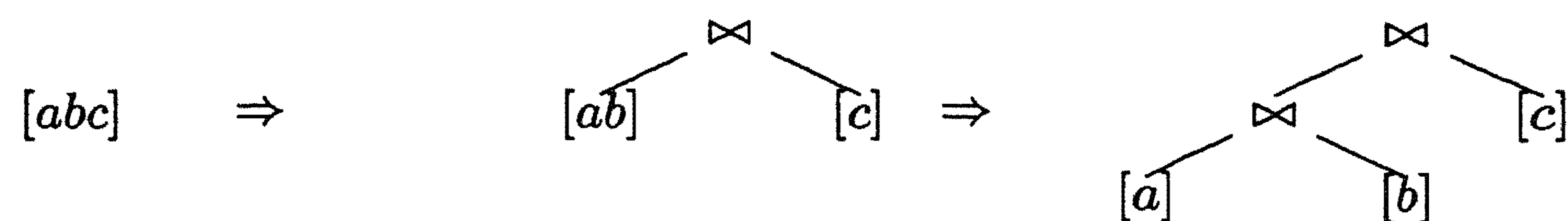


Figure 3.3: Operator tree extraction from a MEMO-structure.

method is similar to the naive algorithm as described in the introduction of this Chapter.

Figure 3.4 shows the exploration algorithm. The initial MEMO-structure is created by walking down a join tree and creating a class for each join operator. This join tree is selected arbitrarily from the space of valid join trees. To start the exploration we call `EXPLORE-CLASS(`C`)`, with C being the root class of the initial MEMO-structure.

In general, the application of a transformation rule can generate an operator which is already present in the MEMO-structure. For example, applying the commutativity rule twice reproduces the original operator. So, before inserting a new operator into the MEMO-structure we have to make sure it is not already present. A hash table is used to speed-up the detection of duplicates.

3.1.3 Transformation rules

To generate the two most commonly used search spaces, the space of *bushy* join trees and the space of *left-linear* join trees, the following sets of transformation rules are used. For generating the space of bushy join trees one may use the following rule set. This set was also used in [BMG93, IW87, IK91, Kan91].

```
EXPLORE-CLASS(C) {
   while not all operators in C have been explored {
      pick an unexplored operator e ∈ C
      EXPLORE-OPERATOR(e);
      mark e explored;
   }
}

EXPLORE-OPERATOR(e) {
   EXPLORE-CLASS(left-child(e));
   EXPLORE-CLASS(right-child(e));
   for each rule R {
      for each partial tree ê such that
            ê is extracted from the MEMO-structure;
            the root of ê is e; and
            ê matches the pattern of R
         x̂ := apply R on ê;
      if x̂ ∉ MEMO-structure
         add x̂ to MEMO-structure;
         (place the root of x̂ in the same class as e)
   }
}
```

Figure 3.4: Exploration algorithm

Rule set **RS-B0**:

- Right Associativity: $(A \bowtie B) \bowtie C \rightsquigarrow A \bowtie (B \bowtie C)$.
- Left Associativity: $A \bowtie (B \bowtie C) \rightsquigarrow (A \bowtie B) \bowtie C$.
- Commutativity: $A \bowtie B \rightsquigarrow B \bowtie A$.

This set is redundant, because we can drop Right Associativity (or Left Associativity) and still generate the same space. We use here the minimal set **RS-B1**, which contains only Left Associativity and Commutativity. As will be shown later on, the addition of redundant transformation rules will have an impact on the performance of the join enumerator.

For generating the space of left linear join trees for completely connected queries a rule set based on [SG88] is used.

Rule set **RS-L1**:

- Swap: $(A \bowtie B) \bowtie C \leadsto (A \bowtie C) \bowtie B$.
- Bottom Commutativity: $B_1 \bowtie B_2 \leadsto B_2 \bowtie B_1$, for base tables B_1, B_2.

3.2 Size of the MEMO-structure

To determine the number of join operators needed to encode a complete search space using the MEMO-structure a few well known query graph topologies are considered, namely: *string, star* and *completely connected* queries (See Section 2.1). For these topologies the size of the MEMO-structure is determined for the cases that *ordered bushy* or *left-linear* join trees are generated. Note that, since the number of join operators are computed, the operators that load a base relation are not counted.

3.2.1 Bushy join evaluation orders

Theorem 3.1 *The maximum number of join operators needed to encode all alternative evaluation orders for a query of n relations is:*
$3^n - 2^{n+1} + 1, n > 1$

Proof. The upper bound on the size of the MEMO-structure is determined by considering a query topology with the largest number of alternative evaluation orders: A completely connected query on n relations. First, we compute the number of equivalence classes. Since each possible non-empty subset of base relations will occur as intermediate result the number of equivalence classes is: $\sum_{k=1}^{n} \binom{n}{k} = 2^n - 1$.
An equivalence class for k base relations describes all possible root operators for these k relations. Every partition of the set of the k relations into left/right non-empty subsets corresponds to an operator in this class, so the number of operators in a class is $2^k - 2$, for $k > 1$. If $k = 1$ the class contains no join operator, but a load-operator which access the base relation. Now the number of join operators in the MEMO-structure is the sum of all operators of all classes which contain join operators, which is: $\sum_{k=2}^{n} \binom{n}{k} (2^k - 2) = 3^n - 2^{n+1} + 1, n > 1$. □

Theorem 3.2 *The maximum number of join operators needed to encode all alternative bushy evaluation orders in case of acyclic queries of n relations is:* $(n-2)2^n + 2, n > 1$

Proof. In case the query is acyclic, every sub query is also acyclic. The number of operators in a class with k, $k > 1$, relations is $2(k-1)$. If $k = 1$ the class only contains a load-operator and no join operators.
Since the exact topology of the acyclic query graph is unknown and bushy join trees are allowed we assume that all sub graphs exists in the MEMO-structure. This leads to the following summation for the total number of operators: $\sum_{k=2}^{n} \binom{n}{k} 2(k-1)$. Rewriting results in: $(n-2) * 2^n + 2, n > 1$ □

For acyclic queries with a "structured" topology the size of the MEMO-structure can be made more precise. In the following Theorem we give the size of the MEMO-structure for the number of operators needed to encode all bushy trees for string queries.

Theorem 3.3 *The complete space of bushy join trees for string queries can be encoded in a MEMO-structure using* $(n^3 - n)/3, n > 1$, *join operators.*

Proof.
As in the proof of Theorem 3.2, a class with k relations contains $2(k-1)$ operators, for $k > 1$. For a string query with n relations there are exactly $n-k+1$ possible substrings of size k. For each of these substrings there is a class in the MEMO-structure.
So the total number of operators in the MEMO-structure is $2\sum_{k=2}^{n}(n-k+1)(k-1)$. Rewriting results in: $(n^3 - n)/3, n > 1$ □

In [OL90] these query graphs and join tree topologies were also considered in the context of the Starburst join enumerator. Our results coincide with their findings, however there is a difference which is caused by the fact that we considered ordered join trees while they considered unordered join trees. To map their results to ours their formulas have to be multiplied by a factor two.

3.2.2 Left-linear join trees

Now we will determine the size of the MEMO-structure if only left-linear join trees are generated. The query graph topologies considered are: *completely connected, string* and *star*.

When counting the number of left-linear join trees we assume that the join operators do not distinguish between the left and right input — i.e. we count unordered linear trees. However, the join operator at the "bottom" of a join tree does distinguish between its two inputs [LVZ93]. So,

for a completely connected query on the relations a, b and c there are 6 different left-linear join trees, namely: $(c \bowtie b) \bowtie a$, $(b \bowtie c) \bowtie a$, $(a \bowtie c) \bowtie b$, $(c \bowtie a) \bowtie b$, $(a \bowtie b) \bowtie c$, $(b \bowtie a) \bowtie c$.

To determine the number of join operators needed to represent all valid left-linear join trees for acyclic query graphs, we use the following observation. In each join operator of class C at least one of the inputs is a base relation. Also this base relation has to be a "terminal" in the subgraph associated with class C, otherwise the join operator can not be valid.

Theorem 3.4 *The number of join operators needed to encode all left-linear join trees for a completely connected join query on n relations is* $n2^{n-1} - n, n > 1$.

> **Proof.** A class contains operators such that the right hand operand consists of a single base relation. So the number of operators in a class which references k relations is k. The n classes referencing 1 relation contain no join operators, but only a load-operator.
> The number of classes with k relations in the fully explored MEMO-structure table is $\binom{n}{k}$.
>
> Summing the join operators in each class we get $\sum_{k=2}^{n} \binom{n}{k} k$. Rewriting the summation results in: $n2^{n-1} - n, n > 1$. □

Theorem 3.5 *The number of join operators needed to encode all left-linear join trees for a string query is* $n^2 - n, n > 1$.

> **Proof.** In each complete class that references at least two relations there are 2 operators, since the two terminal relations appear once as right input of a join operator. If any other relation of the query is used as right input, the resulting join tree would be invalid.
> Each sub-string of the complete string query appears as the left input. Summing the operators of each class gives the total number of operators. The number of possible sub-strings is $n - k + 1$ with k the number of relations in the sub-string and n the number of relations in the complete string. Summing, we get: $\sum_{k=2}^{n} 2(n - k + 1) = n^2 - n, n > 1$. □

Theorem 3.6 *The number of join operators needed to encode all left-linear join trees for a star query is* $(n-1) * 2^{n-2} + n - 1, n > 2$.

Proof. In a class which references k relations there are $k-1$ possible join operators, $k > 2$. Hence, each of the $k-1$ "terminal" relations is used once as the right input for a join operator. The classes which reference two relations contain 2 operators, since the bottom operator distinguishes between the left and right input. The classes that reference one relation contain no join operator.

The number of classes which reference k relations is $\binom{n-1}{k-1}$. The reason is that for each operator the right input is a "terminal" relation and the left input is formed by the remaining $k-1$ relations which are selected from the $n-1$ original relations. Note that the central relation of the star query is always part of the left input. Summing the number of operator in each class we get: $\sum_{k=3}^{n} \binom{n-1}{k-1}(k-1) + 2\binom{n-1}{1}$. Rewriting results in $(n-1) * 2^{n-2} + n - 1, n > 2$. □

In [OL90] also the number of operators needed to encode all left linear trees for string and star queries were derived. These results coincide with our findings except for a minor deviation, caused by the fact that the "bottom" join operator we use, distinguishes between the left and right input while [OL90] counted only one operator in those cases.

Rel.	Bushy				Left-linear					
	cc		string		cc		string		star	
	trees	ops.	trees	ops.	trees	ops.	trees	ops.	trees	ops.
2	2	2	2	2	2	2	2	2	2	2
3	12	12	8	8	6	9	4	6	4	6
4	120	50	40	20	24	28	8	12	12	15
5	1680	180	224	40	120	75	16	20	48	35
6	30240	602	1344	70	720	186	32	30	240	85
7	665280	1932	8448	112	5040	441	64	42	1440	265

Figure 3.5: Number of join trees and operators for various query graphs and join trees.

The table in Figure 3.5 shows how many operators are needed to encode a specific search space using a MEMO-structure. All the combinations of query graphs and join trees discussed in this section are described. For computing the size of the search space — number of join trees — the formulas of Figure 3.6 were used. Note that for star query graphs there are no "real" bushy join trees. In the tables "cc" is short for completely connected.

	Bushy	Left-linear
cc	$\frac{(2n-2)!}{(n-1)!}$	$n!$
string	$\frac{(2n-2)!}{n!(n-1)!}2^{n-1}$	2^{n-1}
star	-	$2(n-1)!$

Figure 3.6: Formula's for the size of search spaces.

3.3 Duplicates

Before quantifying the effect of duplicate generation, we walk through the construction of a MEMO-structure for a completely connected query, on relations a, b and c. Even in this small example the number of duplicates is relatively large.

Example 3.1 *For the completely connected query on the relations "a, b, c," Figure 3.7 shows the MEMO-structures before and after exploring operator* $[ab] \bowtie [c]$. *In the "before" MEMO-structure all children, "ab" and "c", have been fully explored. The transformation rules* **RS-B1** *(see Section 3.1) generate the following new operators, when applied to operator* $[ab] \bowtie [c]$.

Commutativity: $([ab] \bowtie [c])$ *creates* $([c] \bowtie [ab])$ *which is added to the class "abc".*

Associativity: $([ab] \bowtie [c])$ *does not match, the left child is a class and should be a tree. This is resolved by extracting partial trees for the left class "ab." — i.e. replacing the expression "ab" by one of the operators of that class.*

$[a] \bowtie [b]$: *First tree* $(([a] \bowtie [b]) \bowtie [c])$ *is extracted. Now the rule matches and is applied. The new tree* $([a] \bowtie ([b] \bowtie [c]))$ *is generated, and added to the MEMO-structure in class "abc". The subexpression* $([b] \bowtie [c])$ *starts a new class "bc" since it didn't appear in the earlier MEMO-structure.*

$[b] \bowtie [a]$: *Second tree* $(([b] \bowtie [a]) \bowtie [c])$ *is extracted. It matches the rule, so it is applied. The new tree* $([b] \bowtie ([a] \bowtie [c]))$ *is generated and added to the MEMO-structure. The subexpression* $[a] \bowtie [c]$ *starts a new class "ac".*

Before	*After*
$abc{=}[ab] \bowtie [c]$	$abc{=}[ab] \bowtie [c]; [c] \bowtie [ab]; [a] \bowtie [bc];$ $[b] \bowtie [ac]; [bc] \bowtie [a]; [ac] \bowtie [b].$
$ab = [a] \bowtie [b]; [b] \bowtie [a]$	$ab = [a] \bowtie [b]; [b] \bowtie [a]$
	$bc = [b] \bowtie [c]; [c] \bowtie [b]$
	$ac = [a] \bowtie [c]; [c] \bowtie [a]$

Figure 3.7: MEMO-structure before and after exploration.

The exploration process is continued by applying transformation rules to the newly created operators. Now, duplicates are generated. Before the new operators ($[c] \bowtie [ab]$, $[a] \bowtie [bc]$, and $[b] \bowtie [ac]$) of the root class "abc" can be explored, all their children ("a","b","c" "ab","bc" and "ac") must be fully explored. This results in two new operators, $[c] \bowtie [b]$ and $[c] \bowtie [a]$, which are added to the appropriate classes.

Commutativity on the new operators produces $[ab] \bowtie [c]$, $[bc] \bowtie [a]$ and $[ac] \bowtie [b]$, out of which $[ab] \bowtie [c]$ already exists. The new operators are added to the MEMO-structure and explored. Both associativity and commutativity can be applied to the operators $[bc] \bowtie [a]$ and $[ac] \bowtie [b]$, which results in 6 operators. All these operators were already stored in the MEMO-structure. So, during the exploration of class "abc", 5 new operators and 7 duplicates were generated. In Figure 3.8 the generation graph for class "abc" is shown. The bold operators are duplicates.

As illustrated by the previous example, the straight forward application of transformation rules results in the generation of operators which are already in the MEMO-structure —duplicates. The generation of duplicates affects the efficiency of the join enumeration process considerably. For each operator generated the MEMO-structure has to be searched to determine if it already exists. The search and the time needed to generate duplicates are part of the generation cost.

In the introduction, where a simple graph model was assumed, we derived a factor of $b - 1$ of duplicate elements over new elements. The naive model used there, however, did not take into account the MEMO-structure used by the Volcano-type optimizers. The next Theorems deal with the details of the new structure.

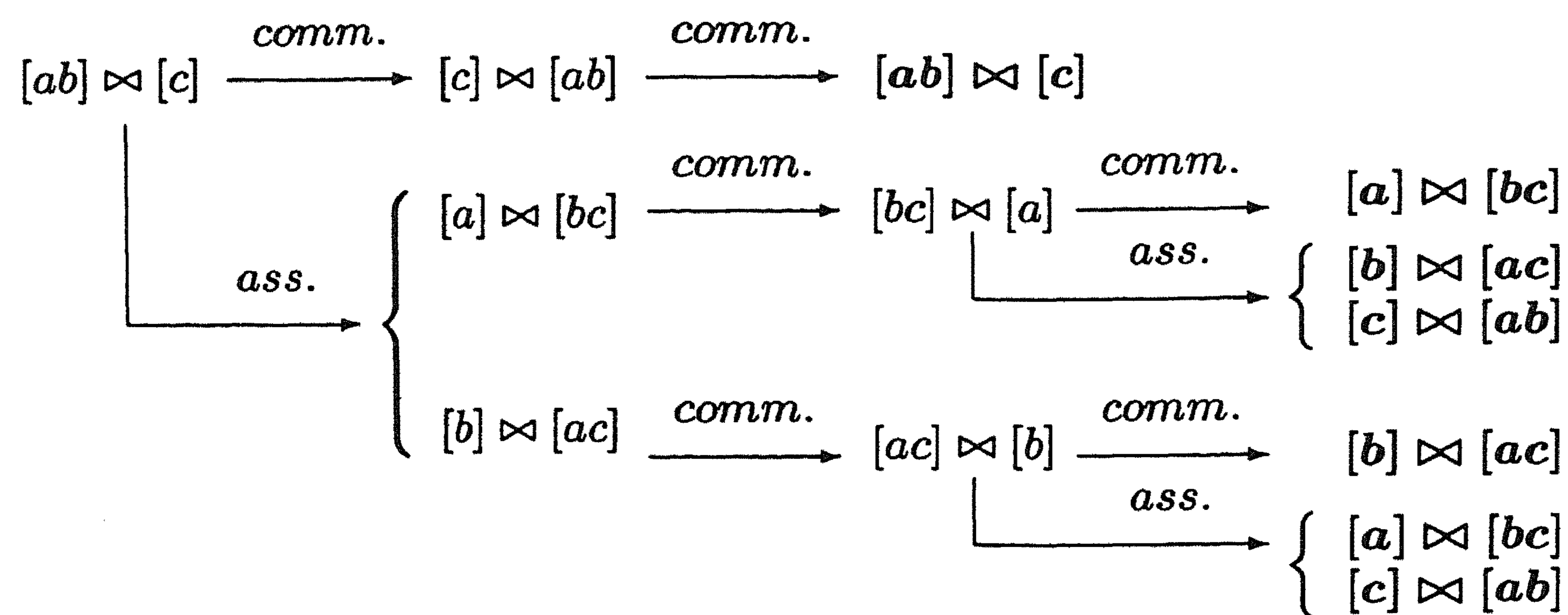


Figure 3.8: Operator generation graph.

3.3.1 Bushy join trees

The following theorems and lemmas quantify the number of duplicates generated when exploring the search space of bushy join trees, for completely connected and acyclic query graphs. It assumes a minimal set of unidirectional join associativity and commutativity rules, see Section 3.1. If associativity is enabled in both direction, as is commonly suggested, we simply end up generating even more duplicates.

Completely connected query

Lemma 3.1 *The number of duplicates generated by* **RS-B1** *during the exploration of a class that combines k relations, on a completely connected graph, is:* $3^k - 3 * 2^k + 4$.

> **Proof.** In a class that combines k relations, take an operator $[L] \bowtie [R]$, with l the number of relations in $[L]$ and $k - l$ the number of relations in $[R]$ $(2 \leq k \leq n)$. Applying commutativity and associativity on this operator, we generate $(2^l - 2) + 1$ alternatives.
> To ensure that all operators are generated within a class, all transformation rules are applied on each operator. Thus, the total number of operators generated in the class is $\sum_{l=1}^{k-1} \binom{k}{l} (2^l - 1)$. Rewriting, the summation becomes $3^k - 2^{k+1} + 1$. But the number of unique operators

in such a class is $2^k - 2$ and of these operators the initial operator is given instead of being generated. Therefore the number of duplicates generated in the class is: $3^k - 2^{k+1} + 1 - (2^k - 2 - 1) = 3^k - 3 * 2^k + 4$.
□

Theorem 3.7 *The number of duplicates generated by* **RS-B1** *during the construction of a MEMO-structure encoding all bushy join trees for a query with n relations, on a completely connected graph, is:*
$4^n - 3^{n+1} + 2^{n+2} - n - 2$.

Proof. The MEMO-structure consists of $\binom{n}{k}$ classes that combine k relations, $2 \leq k \leq n$. Using Lemma 3.1 the total number of duplicates generated is $\sum_{k=2}^{n} \binom{n}{k} (3^k - 3 * 2^k + 4)$. Rewriting results in: $4^n - 3^{n+1} + 2^{n+2} - n - 2$. □

Acyclic query For acyclic queries the number of classes required for the MEMO-structure to encode a fully explored search space depends on the number or participating relations as well as on the topology of the acyclic query. This makes it hard to give a formula for the general case of acyclic queries. Therefore we consider a string query for which the exact number of duplicates generated can be computed.

To determine the number of duplicates generated during the construction of a MEMO-structure we use the following lemma which states the number of duplicates generated during the exploration of a single class.

Lemma 3.2 *The number of duplicates generated by* **RS-B1** *during the exploration of a class that combines k relations, on an acyclic query graph, is:* $k^2 - 3k + 3, k > 1$

Proof. In a fully explored class that combines k relations there are $2(k-1)$ valid joins in case the query graph is acyclic and bushy threes are generated. On all these joins the commutativity rule can be applied and results in the generation of $2(k-1)$ join operators.
If the left associativity rule is applied on an operator $[L] \bowtie [R]$, with l the number of operators in $[L]$ and $k - l$ the number of operators in $[R]$, then $l - 1$ operators are generated. The reason is that, although class L consists of $2(l-1)$ operators only half of them will result in valid operators, since the query graph is acyclic. If all operators would result in valid operators both left and right inputs of operators in class L would

share a predicate with R, which means that the query graph would have to be cyclic.
When applied on the mirrored operator, $[R] \bowtie [L]$, the number of operators generated is: $k-l-1$. In total, the number of operators generated for each (non-mirrored) operator is $k-2$.
The number of non-mirrored operators is $k-1$ so $(k-1)(k-2)+2(k-1) = k(k-1)$ operators are generated for a class that combines k relations. The fully explored class consists of $2(k-1)$ operators of which the initial one is already given, so the number of duplicates generated is: $k(k-1) - (2(k-1)-1) = k^2 - 3k + 3, k > 1$. □

Theorem 3.8 *The number of duplicates generated by* **RS-B1** *during the construction of a MEMO-structure encoding all bushy join trees for a string query with n relations is:* $\frac{1}{12}(n^4 - 2n^3 + 5n^2 - 4n), n > 1$

Proof. From Lemma 3.2 we know that the number of duplicates generated per class of k relations is: $k^2 - 3k + 3, k > 1$. For a string query of n relations there are $n-k+1$ of k relations so the number of duplicates generated during the exploration of a MEMO-structure is: $\sum_{k=2}^{n}(n-k+1)(k^2-3k+3)$. Rewriting results in: $\frac{1}{12}(n^4-2n^3+5n^2-4n), n > 1$
□

3.3.2 Linear join trees

Completely connected queries. The following lemma and theorem show how many duplicate join operators are generated when constructing the MEMO-structure for all left linear join trees.

Lemma 3.3 *The number of duplicates generated by* **RS-L1** *during the exploration of a class that combines k relations, on a completely connected graph, is* $k^2 - 2k + 1$.

Proof. In a class that combines k relations, take an operator $[L] \bowtie [R]$, with $k-1$ the number of relations in $[L]$ and one relation in $[R]$, $2 \leq k \leq n$.
For a class with more than two relations, $k > 2$, only the Swap rule applies to the join operators. For each join operator this rule generates $k-1$ alternative operators. Since a class contains k unique operators, $k(k-1)$ operators are generated in a class. Of the k unique operators one is already given, the initial one, so the number of duplicates generated is: $k(k-1) - (k-1) = k^2 - 2k + 1$.

For the class with exactly two relations, $k = 2$, only the Bottom Commutativity rule applies. This rule behaves the same as the Swap rule so the formula $k(k-1) - (k-1) = k^2 - 2k + 1$ also holds for $k = 2$. □

Theorem 3.9 *The number of duplicates generated by* **RS-L1** *during the construction of a MEMO-structure encoding the left linear join trees for a query with n relations, on a completely connected graph, is* $(n^2 - 3n + 4)2^{n-2} - 1$.

Proof. The MEMO-structure consists of $\binom{n}{k}$ classes that combine k relations, $2 \leq k$. Using Lemma 3.3, the total number of duplicates generated during the construction of the MEMO-structure is: $\sum_{k=2}^{n} \binom{n}{k} (k^2 - 2k + 1)$. Rewriting results in $(n^2 - 3n + 4)2^{n-2} - 1$. □

Acyclic queries. Again it is not possible to compute the exact number of duplicates generated for the general case of acyclic queries. Therefore, we consider two acyclic query graphs, namely star and string queries.

Lemma 3.4 *The number of duplicates generated by* **RS-L1** *during the exploration of a class of a string query, is* 1.

Proof. If the Swap rule is applied on the initial operator $[L] \bowtie [R]$ of a class with $k > 2$ relations, with $l = 2$ the number of operators in class L and R being a base relation, then two new join operators are considered but due to the structure of the query graph only one of them is valid. To this new operator the Swap rule is also applied, which will result in the generation of the initial operator, so only one duplicate is generated. For the class with two relations, $k = 2$, only the bottom commutativity rule can be applied which behaves similarly to the swap rule and also generates exactly one duplicate. □

Theorem 3.10 *The number of duplicates generated by* **RS-L1** *during the construction of a MEMO-structure encoding the left linear join trees for a query with n relations, on a string query, is* $\frac{n^2-n}{2}$.

Proof. A complete MEMO-structure for a string query consists of $n - k + 1$ classes which combine k relations. Using Lemma 3.4 the number of duplicates generated during the creation of a complete MEMO-structure is: $\sum_{k=2}^{n}(n - k + 1) = \frac{n^2-n}{2}$. □

Lemma 3.5 *The number of duplicates generated by* **RS-L1** *during the exploration of a class that combines* k *relations, on a star query, is:* $(k-2)^2, k > 2$.

Proof. In a class that contains all left linear join operators for a star query, the number of join operators is $k-1$, with k the number of relations that are combined. Each of these operators, $[L] \bowtie [R]$, results in $k-2$ valid join operators when the Swap rule is applied, since L consists of $k-2$ operators. So the number of join operators generated is: $(k-1)(k-2), k > 2$.
Since there are only $k-1$ unique join operators and the initial operator is already given the number of duplicates generated is: $(k-1)(k-2) - (k-1-1) = (k-2)^2, k > 2$. □

Theorem 3.11 *The number of duplicates generated by* **RS-L1** *during the construction of a MEMO-structure encoding the left linear join trees for a query with* n *relations, on a star query, is:* $(n-1)(n-2)2^{n-3} - (n-2)2^{n-2} + 2^{n-1} - 1$, $n > 2$.

Proof. For a star query the fully explored MEMO-structure consists of $\binom{n-1}{k-1}$ classes that combine k relations. Using Lemma 3.5 the number of duplicates generated during the creation of a complete MEMO-structure is:
$\sum_{k=3}^{n} \binom{n-1}{k-1} (k-2)^2$. Rewriting results in: $(n-1)(n-2)2^{n-3} - (n-2)2^{n-2} + 2^{n-1} - 1$, $n > 2$ □

	Bushy join trees		Left-linear join trees	
Relations	Operators	Duplicates	Operators	Duplicates
2	2	1	2	1
3	12	10	9	7
4	50	71	28	31
5	180	416	75	111
6	602	2157	186	351
7	1932	10326	441	1023

Figure 3.9: Number of duplicates generated during the exploration of a MEMO-structure for a completely connected query.

Figure 3.9 shows concrete numbers for the size of the MEMO-structure and the duplicates generated (both bushy and linear trees), as a function of the number of relations joined, for fully connected graphs. The second column gives the number of operators needed to encode all bushy trees using the MEMO-structure. The number of duplicates generated during the exploration process is given in column 3. For linear join trees the size of the MEMO-structure and the number of duplicates generated is given in column 4 and 5.

When for completely connected queries the results from Theorem 3.1 and 3.7 are combined it shows that for bushy join trees the ratio of duplicates over new operators is $O(2^{n \log(4/3)})$. For left linear join trees, the ratio of duplicates over new operators is $O(n)$, as can be seen from Theorem 3.4 and 3.9.

Similarly we can determine the ratio of duplicates over new operators in case string or star queries are used. For string queries combining the results from Theorem 3.3 and 3.8 shows that for bushy join trees the ratio of duplicates over new operators is $O(n)$. For left linear join trees the ratio of duplicates over new operators is constant, as can be seen from Theorem 3.5 and 3.10. For star queries no valid bushy join trees exist. The ratio of duplicates over new operators is $O(n)$, if left linear join trees are generated, as can be seen from Theorem 3.6 and 3.11.

3.4 Summary

In this chapter we have analysed the problem of duplicate generation for transformation-based optimizers that explore a space exhaustively. Despite the exponential size of the space, exhaustive search is often used in practice. We are aware of at least two commercial DBMSs under development that are using a Volcano-type optimizer based on exhaustive search one at Tandem in their *NonStop SQL* product and one at Microsoft in their *SQL Server* product [Gra95].

We have shown that duplicates are a serious problem in transformation based optimizers. Even for small queries the number of duplicates exceeds the number of new operators, and it increases dramatically with the size of the query. In particular, for the Volcano-type optimizers the ratio of duplicates over new operators can be up-to $O(2^{n \log(4/3)})$. The detailed complexity analysis developed here is the first that we are aware of, for this type of optimizers.

Chapter 4

Duplicate-free join enumeration[1]

In this chapter, we show that it is possible to generate the space of alternative join trees efficiently — i.e. without generating duplicates — within the framework of an extensible transformation-based optimizer. The technique described is based on conditioning the application of rules on the derivation history of an operator. Each join tree maintains a set of rules that can still be applied on it without generating duplicates.

To determine by which rule a join operator has been generated, a "derivation history" is recorded for each operator. This is done by keeping track of rules that are still worth applying. For example, the application of the commutativity rule will switch the commutativity rule off in the rule set of the resulting operator.

Keeping track of the derivation history requires only a few bits per operator. The applicability of a rule can be encoded using a single bit. Therefore, each operator needs as many bits as there are transformation rules.

For each class of the MEMO-structure a *generation graph* describes for each operator by which application of a transformation rules is was generated. The nodes in the generation graph represent the operators and the directed edges the transformation rules.

The basic idea of the duplicate-free transformation rules described in this chapter, is that the generation graph imposed on the space of operators by the naive transformation rules is transformed into a spanning tree —

[1]Parts of this chapter have been published in the *Proceedings of the International Conference on Database Systems For Advanced Applications, Melbourne, 1997* [PGLK97b].

i.e. from the root of the spanning tree there is exactly one path to every node (operator) in the graph.

In the next sections, we present sets of duplicate-free transformation rules, together with an application schema. The join trees generated are either *left-linear* or *bushy*. The query graph topologies we consider are *acyclic* and *completely connected*.

For bushy join trees, Section 4.1 describes duplicate-free sets of transformation rules for both completely connected and acyclic queries. For left-linear join trees, Section 4.2 describes duplicate-free rules for these queries. Section 4.3 shows experimentally the performance improvement of avoiding duplicates and a summary is given in Section 4.4.

4.1 Bushy join trees

The following two sections describe transformation rules which generate bushy join trees for completely connected and acyclic query graphs.

4.1.1 Completely connected queries

Generating all valid join trees for completely connected queries is similar to generating all *operator* trees for arbitrary query graphs (cyclic or acyclic) that contain both Cartesian products and join operators. Each query graph can be transformed into a completely connected query graph by adding the "missing" predicates. These added predicates are set to *true*, forcing the join operators to compute a Cartesian product. The following rule set generates all bushy join trees for completely connected queries without duplicates.

Rule set $\mathbf{RS\text{-}B_{cc}}$

R_1 : **Commutativity** $x \bowtie_0 y \rightarrow y \bowtie_1 x$
Add $\bowtie_1$ to the class of $\bowtie_0$.
Disable all rules R_1, R_2, R_3, R_4 for application on $\bowtie_1$.

R_2 : **Right associativity** $(x \bowtie_0 y) \bowtie_1 z \rightarrow x \bowtie_2 (y \bowtie_3 z)$
Add $\bowtie_2$ to the class of $\bowtie_1$.
Disable rules R_2, R_3, R_4 for application on $\bowtie_2$.
Start new class with $\bowtie_3$ (new operator) with all rules enabled.

R_3 : **Left associativity** $x \bowtie_0 (y \bowtie_1 z) \rightarrow (x \bowtie_2 y) \bowtie_3 z$
Add $\bowtie_3$ to the class of $\bowtie_0$.
Disable rules $R_2, R_3 R_4$ for application on $\bowtie_3$.
Start new class with $\bowtie_2$ (new operator) with all rules enabled.

R_4 : **Exchange** $(w \bowtie_0 x) \bowtie_1 (y \bowtie_2 z) \to (w \bowtie_3 y) \bowtie_4 (x \bowtie_5 z)$
Add $\bowtie_4$ to the class of $\bowtie_1$.
Disable all rules R_1, R_2, R_3, R_4 for application on $\bowtie_4$.
Start new classes with $\bowtie_3$ and $\bowtie_5$ (new operators) with all rules enabled.

Each transformation rule generates a non-overlapping partition of the space of join operators, and each rule generates each operator within such a partition exactly once. Furthermore the complete space of join operators is the conjunction of all partitions, see Figure 4.1 .

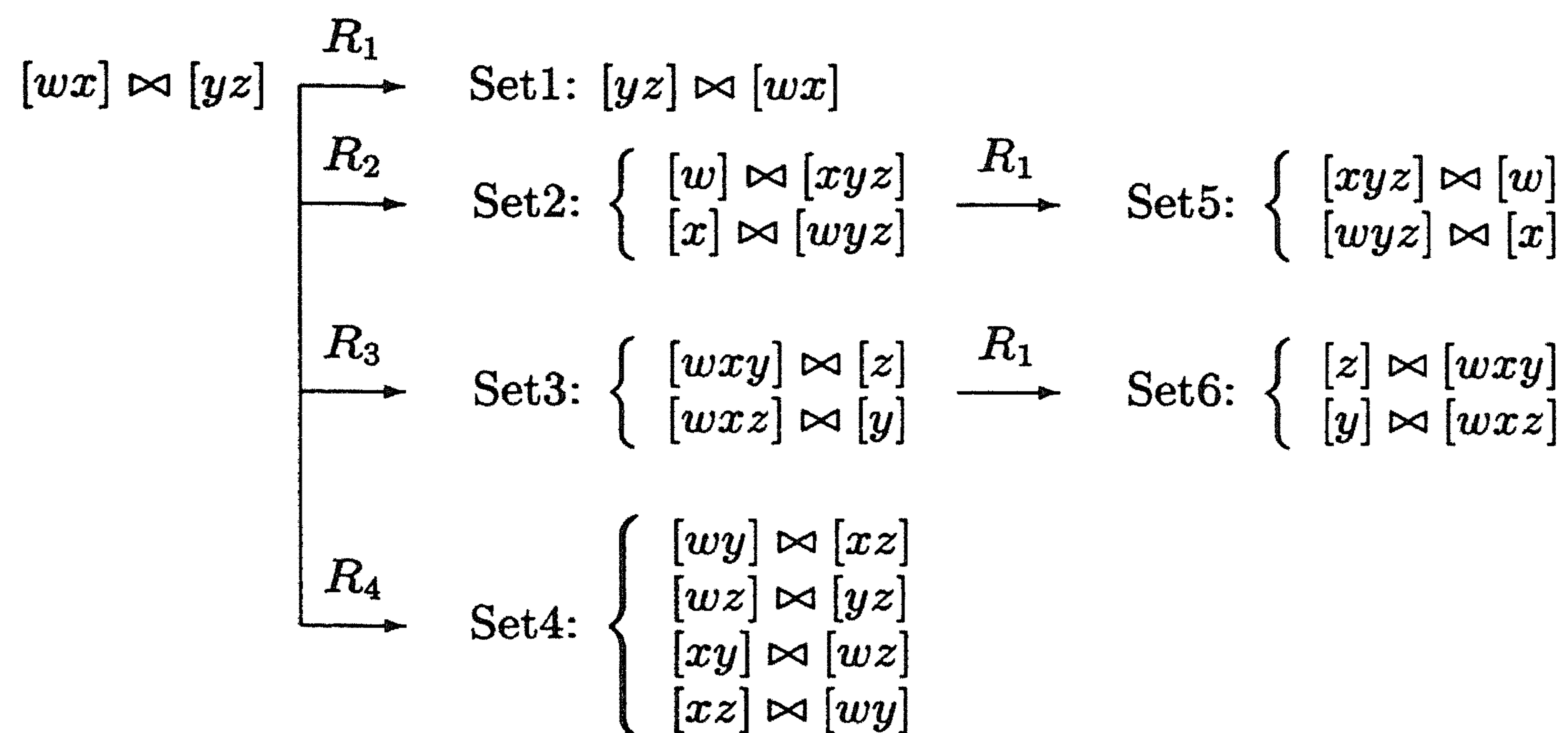


Figure 4.1: Generation graph of rule set $\mathbf{RS\text{-}B_{cc}}$

For an initial join operator, $[L] \bowtie [R]$, with $L = wx$ and $R = yz$ rule R_2 generates the join operators that combine each operator of class $L : \{w \bowtie x, x \bowtie w\}$ with R so the operators $[w] \bowtie [xyz]$ and $[x] \bowtie [wyz]$ are generated. Rule R_3 and R_4 work in a similar fashion. R_1 generates the mirror images for the original operator and all operators generated by rule R_2 and R_3 (since L and R contain all alternatives including the mirror images, rule R_4 automatically generates the mirror images).

Theorem 4.1 *If the transformation rules of rule set* $\mathbf{RS\text{-}B_{cc}}$ *are applied to* $[L] \bowtie [R]$ *then the newly generated operators will be duplicate free if the child classes* L *and* R *are duplicate free.*

Proof. Two operators can not be identical if they are both generated by the same rule (operators of the same set). Namely, rule R_1 is used to generate mirror images of operators; since the left and right operand will never be identical, a duplicate can not be generated. Rule R_2 combines the unique operators of the left child with the right operand of the initial operator resulting in only unique operators. The same holds for rule R_3 and R_4.
Also, no two derivation paths can result in the same operator (operators of different sets). Suppose the application of rule R_2 (Set 2) generated the same operator as rule $R_3; R_1$ (Set 6), then $[w] \bowtie [xyz]$ or $[x] \bowtie [wyz]$ has to be equal to $[z] \bowtie [wxy]$ or $[y] \bowtie [wxz]$. This can not be true since w, x, y, z are disjunct non-empty sets of relations, so $[w] \neq [z] \neq [x] \neq [y]$ and $[xyz] \neq [wxy] \neq [wyz] \neq [wxz]$. A similar argument can be given for any other combination of rules. □

Theorem 4.2 *Given a MEMO-structure which encodes a single join tree, the rules of rule set* **RS-B$_{cc}$** *generates all valid bushy join orders for completely connected queries.*

Proof. In a fully explored class that references n relations the number of join operators is $B_{cc}(n) = 2^n - 2$ (See proof of Theorem 3.1). From the initial operator of a class, say $[L] \bowtie [R]$ the transformation rules generate the following operators. Rule R_2 combines each *operator* of class $[L]$ with $[R]$ resulting in $B_{cc}(|L|)$ new operators, with $|L|$ denoting the number of relations in class L. Similarly rule R_3 generates $B_{cc}(|R|)$ new operators. Rule R_4 combines each *operator* of class L with each *operator* of class R which results in $B_{cc}(|L|)B_{cc}(|R|)$ new operators. Finally rule R_1 generates the mirror images for the initial operator and the operators generated by rule R_2 and R_3. Adding all the newly created operators and the initial operator we get: $2 + 2B_{cc}(|L|) + 2B_{cc}(|R|) + B_{cc}(|L|)B_{cc}(|R|)$. Rewriting shows that $2 + 2B_{cc}(|L|) + 2B_{cc}(|R|) + B_{cc}(|L|)B_{cc}(|R|) = B_{cc}(|L| + |R|)$. From Theorem 4.1 we know that no duplicates are generated, so all valid bushy join trees are generated. □

Example 4.1 *Figure 4.2 shows a partial MEMO-structure for a completely connected query on the five relations $\{a, b, c, d, e\}$, in which the children of operator $[ab] \bowtie [cde]$ have been fully explored. Applying the transformation rules of rule set* **RS-B$_{cc}$** *results in the generation of the following operators.*

Rule R$_2$: *Each operator of class "ab" is combined with the right subtree $[cde]$ to obtain $[a] \bowtie [bcde], [b] \bowtie [acde]$.*

$$\begin{array}{ll} abcde & = [ab] \bowtie [cde] \\ cde & = [c] \bowtie [de]; [de] \bowtie [c]\ ; [d] \bowtie [ce]; \\ & [ce] \bowtie [d]; [e] \bowtie [cd]; [cd] \bowtie [e] \\ ab & = [a] \bowtie [b]; [b] \bowtie [a] \\ cd & = [c] \bowtie [d]; [d] \bowtie [c] \\ ce & = [c] \bowtie [e]; [e] \bowtie [c] \\ de & = [d] \bowtie [e]; [e] \bowtie [d] \end{array}$$

Figure 4.2: Partial MEMO-structure with bushy trees for a completely connected query on five relations.

Rule $\mathbf{R_3}$: *Each operator of class "cde" is combined with the left subtree* $[ab]$ *to obtain* $[abc] \bowtie [de], [abde] \bowtie [c], [abd] \bowtie [ce], [abce] \bowtie [d], [abe] \bowtie [cd], [abcd] \bowtie [e]$.

Rule $\mathbf{R_4}$: *The operators of the class of the left subtree are combined with the operators of the class of the right subtree, resulting in the operators* $[ac] \bowtie [bde], [ade] \bowtie [bc], [ad] \bowtie [bce], [ace] \bowtie [bd], [ae] \bowtie [bcd], [acd] \bowtie [be]$ *and the mirror images* $[bde] \bowtie [ac]$, $[bc] \bowtie [ade]$, $[bce] \bowtie [ad]$, $[bd] \bowtie [ace]$, $[bcd] \bowtie [ae]$, $[be] \bowtie [acd]$.

Rule $\mathbf{R_1}$: *The mirror images of the initial operator and the operators generated by the rules R_2 and R_3.*

During the exploration process 20 new classes were generated and, in turn, fully explored. The completely explored class "abcde" contains 30 $(= B_{cc}(5))$ *operators.*

4.1.2 Acyclic queries

To generate all alternative bushy join operators, without duplicates, for acyclic query graphs the following transformation rules and application schema is used:

Rule set $\mathbf{RS\text{-}B_{ac}}$

R_1 : **Commutativity** $x \bowtie_0 y \rightarrow y \bowtie_1 x$
Disable all rules R_1, R_2, R_3 for application on the new operator $\bowtie_1$.

R_2 : **Right associativity** $(x \bowtie_0 y) \bowtie_1 z \rightarrow x \bowtie_2 (y \bowtie_3 z)$
Disable rules R_2, R_3 for application on the new operator $\bowtie_2$.
Start new class with new operator $\bowtie_3$, with all rules enabled.

R_3 : **Left associativity** $x \bowtie_0 (y \bowtie_1 z) \rightarrow (x \bowtie_2 y) \bowtie_3 z$
Disable rules R_2, R_3 for application on the new operator $\bowtie_3$.
Start new class with new operator $\bowtie_2$, with all rules enabled.

For example, consider a query with predicates between relations (w, x), (x, y), (y, z). Using the initial operator $[wx] \bowtie [yz]$ of a class and the fully explored classes of "wx" and "yz", the three transformation rules generate the following five sets of operators (See Figure 4.3). Sets 1,2 and 3 are generated using the initial operator. Sets 4 and 5 are generated using the operators of set 2 and 3. Join $[wx] \bowtie [yz]$ must be a valid join tree (i.e. no Cartesian products) of an acyclic query graph.

Set 2 is generated by the right associativity rule and contains only one valid result. Since the graph is connected and acyclic, there must be a predicate between yz and either w or x, but not both; say it is between yz and x. A sub-query combining tables wyz would have to use a Cartesian product, which is invalid. Therefore, R_2 generates only one valid alternative. The same argument applies for the left associativity rule used for Set 3.

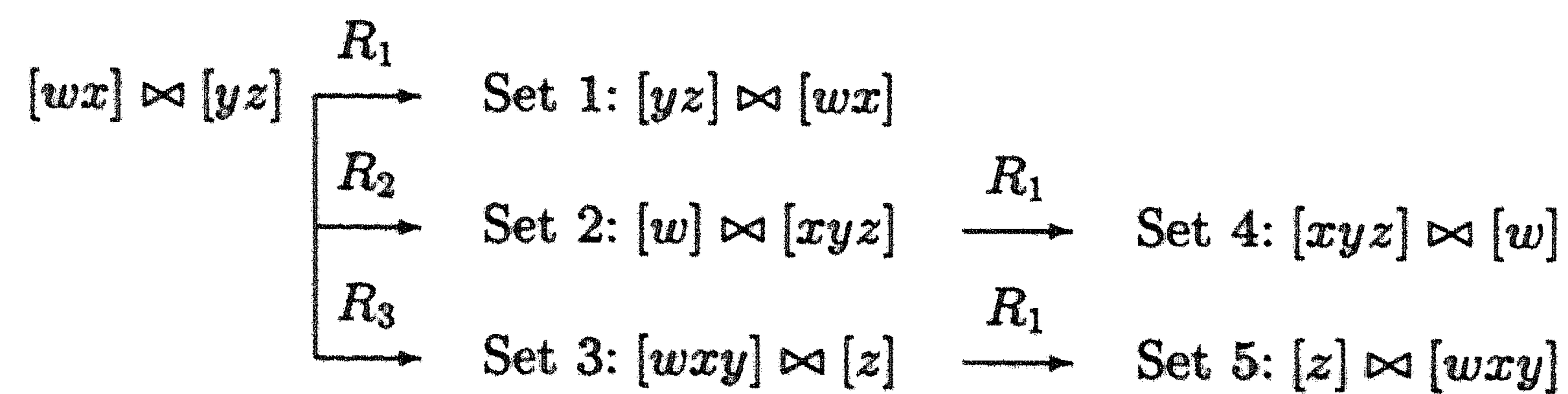


Figure 4.3: Generation graph for rule set $\mathbf{RS\text{-}B_{ac}}$

Theorem 4.3 *No duplicates are generated when rule set* $\mathbf{RS\text{-}B_{ac}}$ *is applied.*

Proof.
The proof is analogous to the proof of Theorem 4.1
□

Theorem 4.4 *For acyclic query graphs rule set* $\mathbf{RS\text{-}B_{ac}}$ *generates all valid bushy join operators.*

Proof. Since the query graph is acyclic, each join operator that can serve as root for a valid join tree corresponds one-on-one to an edge of the query graph. So for a query graph with n relations the number of join operators in the root class is $B_{ac}(n) = 2(n-1)$, when the mirror images are included. Note that each explored class describes all valid roots of the corresponding acyclic sub-graph.
Using the initial operator of a class, say $[L] \bowtie [R]$, the transformation rules generate the following new operators. Rule R_2 combines each *operator* of class L with R; of these new combinations only half are valid since a class contains mirror images and only one can lead to a new valid join operator. This means that rule R_2 generates $B_{ac}(|L|)/2$ new operators, with $|L|$ denoting the number of operators of class L. Similarly rule R_3 generates $B_{ac}(|R|)/2$ new operators. Finally rule R_1 generates for each new operator and the initial operator a mirror image which results in $2(1 + B_{ac}(|L|)/2 + B_{ac}(|R|)/2) = 2 + B_{ac}(|L|) + B_{ac}(|R|)$ operators for the fully explored class.
Now, $2 + B_{ac}(|L|) + B_{ac}(|R|) = B_{ac}(|L| + |R|)$, which is the number of join operators for the fully explored class with $|L| + |R|$ relations. Since, by Theorem 4.3, no duplicate operators were produced, all valid join trees have been generated. □

Example 4.2 *Consider a query* $G = \{\{a, b, c, d, e\}, \{a-b, b-c, c-d, c-e\}\}$ *and a MEMO-structure as shown in Figure 4.4, where the class "abcde" is about to be explored. Further assume that the child classes of the initial operator of the root class "abcde" have been explored exhaustively.*

abcde	$=[ab] \bowtie [cde]$
cde	$=[d] \bowtie [ce]\,;[e] \bowtie [cd]\ ;[ce] \bowtie [d]\ ;[cd] \bowtie [e]$
ab	$=[a] \bowtie [b]\ ;[b] \bowtie [a]$
ce	$=[c] \bowtie [e]\ ;[e] \bowtie [c]$
cd	$=[c] \bowtie [d]\ ;[d] \bowtie [c]$

Figure 4.4: Partial MEMO-structure with bushy trees for an acyclic query.

Applying the transformation rules of rule set $\mathbf{RS\text{-}B_{ac}}$ *to* $[ab] \bowtie [cde]$ *results in generating the following operators.*

Rule $\mathbf{R_2}$: $[a] \bowtie [bcde]$.

The operator $[b] \bowtie [acde]$ is considered by the associativity rule, but rejected, because there is no valid join tree for "acde" (we would be force to introduce a Cartesian product because there is no predicate between a and any of c, d, e).

Rule $\mathbf{R_3}$: $[abce] \bowtie [d], [abcd] \bowtie [e]$.

The operators $[abd] \bowtie [ce]$ and $[abe] \bowtie [cd]$ are considered but rejected because there are no valid join trees for "abd" and "abe".

Rule $\mathbf{R_1}$: $[cde] \bowtie [ab], [bcde] \bowtie [a], [d] \bowtie [abce], [e] \bowtie [abcd]$

The fully explored class "abcde" contains all 8 ($= B_{ac}(5)$) operators. During the exploration process the new classes $[bcde], [abce]$ and $[abcd]$ are created and, in turn, fully explored.

4.2 Linear join trees

Some systems limit the join evaluation orders to linear trees. The following two sections describe transformation rules which generate left-linear join trees for completely connected and acyclic query graphs. As in Section 3.2.2 the left-linear join trees we consider have a "bottom" join operator which distinguishes between its two inputs.

4.2.1 Completely connected queries

To generate all the left-linear join trees for completely connected queries the following transformation rules are used.

Rule set $\mathbf{RS\text{-}L_{cc}}$

R_1 : **Swap** $(x \bowtie_0 y) \bowtie_1 z \rightarrow (x \bowtie_2 z) \bowtie_3 y$.
Disable rule R_1 for application on operator $\bowtie_3$.
x, y and z are classes which reference one or more relations.

R_2 : **Bottom Commutativity**
$(Table_1 \bowtie_0 Table_2) \rightarrow (Table_2 \bowtie_1 Table_1)$.
Disable rule R_2 for application on operator $\bowtie_1$.
$Table_1$ and $Table_2$ are base relation.

All valid join operators in the MEMO-structure have a single base relation as the right operand. For classes with more than two relations the swap rule generates all valid join operators if the class of the left operand is

fully explored. For a class which references two relations the bottom commutativity rule generates a mirror image. This is needed to ensure that in larger classes all base relations appear once as the right input of a join operator.

Alternatively the bottom commutativity rule could be omitted and an exception for classes with two relations could be added to the swap rule, which reduces the size of the MEMO-structure. However, priority is given to clarity over efficiency in describing the generation process of join trees. See Figure 4.5 for the generation graph which clearly shows the two distinct derivation paths.

$$[wx] \bowtie [y] \xrightarrow{R_1} \text{Set 1: } \{[wy] \bowtie [x], [xy] \bowtie [w]\}$$

$$[Table_1] \bowtie [Table_2] \xrightarrow{R_2} \text{Set 2: } [Table_2] \bowtie [Table_1]$$

Figure 4.5: Generation graph of rule set $\mathbf{RS\text{-}L_{cc}}$.

Theorem 4.5 *The transformation rules of rule set $RS\text{-}L_{cc}$ do not generate duplicates if applied to an initial operator $L \bowtie R$, where R consists of a single base relation and the class for L does not contain duplicates.*

Proof. The application schema defines two distinct derivation paths. Either L and R are both base relations and only rule R_2 applies, or L references more than one relation and R is a base relation so only rule R_1 applies. For each case a proof similar to the proof for Theorem 4.1 can be given. □

Theorem 4.6 *For completely connected queries the transformation rules of rule set $RS\text{-}L_{cc}$ generate all valid linear join operators.*

Proof. For a class which references n relations, $n > 2$, there are $L_{cc}(n) = n$ join operators. Each of the n relations appears once as the right operand of a join operator.
For the initial operator $L \bowtie R$ class R is a base relation, $|R| = 1$. Then rule R_1 generates $L_{cc}(|L|)$ new operators. So the total number

$$
\begin{array}{ll}
abcde & = [abcd] \bowtie [e] \\
abcd & = [bcd] \bowtie [a]; [acd] \bowtie [b]; [abd] \bowtie [c]; [abc] \bowtie [d] \\
bcd & = [cd] \bowtie [b]; [bd] \bowtie [c]; [bc] \bowtie [d] \\
acd & = [cd] \bowtie [a]; [ad] \bowtie [c]; [ac] \bowtie [d] \\
abd & = [bd] \bowtie [a]; [ad] \bowtie [b]; [ab] \bowtie [d] \\
abc & = [bc] \bowtie [a]; [ac] \bowtie [b]; [ab] \bowtie [c] \\
cd & = [c] \bowtie [d]; [d] \bowtie [c] \\
bd & = [b] \bowtie [d]; [d] \bowtie [b] \\
bc & = [b] \bowtie [c]; [c] \bowtie [b] \\
ad & = [a] \bowtie [d]; [d] \bowtie [a] \\
ac & = [a] \bowtie [c]; [c] \bowtie [a] \\
ab & = [a] \bowtie [b]; [b] \bowtie [a]
\end{array}
$$

Figure 4.6: Partial MEMO-structure containing linear trees for a completely connected query on five relations.

of operators in the class is $1 + L_{cc}(|L|)$. Rewriting results in $L_{cc}(|L| + |R|)$. Since no duplicates are generated all join operators must have been generated. □

Example 4.3 *Figure 4.6 shows a partial MEMO-structure for a completely connected query on the five relations a, b, c, d, e, in which the children of the initial operator $[abcd] \bowtie [e]$ have been fully explored. Applying the transformation rules of rule set RS-$L_{cc}$to the initial operator $[abcd] \bowtie [e]$ results in the generation of the following operators:*

Rule $\mathbf{R_1}$: $[bcde] \bowtie [a]$, $[acde] \bowtie [b]$, $[abde] \bowtie [c]$ *and* $[abce] \bowtie [d]$.

Rule $\mathbf{R_2}$: *This rule is only active in classes that reference two base relations. For instance when class "ae" is generated, rule R_2 is applicable and will generate a mirror image.*

During the exploration process four new classes are created and, in turn, fully explored. After exploration the total number of operators in the root class is 5 $(= C_{cc}(5))$.

4.2.2 Acyclic queries

The rule set used to generate all linear join trees for acyclic queries is similar to rule set $\mathbf{RS\text{-}L_{cc}}$. The only difference is that the operators generated by the swap rule are not necessarily valid so a test is added to the transformation rule.

Rule set $\mathbf{RS\text{-}L_{ac}}$

R_1 : **Swap** $(x \bowtie_0 y) \bowtie_1 z \rightarrow (x \bowtie_2 z) \bowtie_3 y$.
Disable rule R_1 for application on operator $\bowtie_3$.
x, y and z are classes which reference one or more relations.
Add $\bowtie_3$ only to the class of $\bowtie_1$ if it is a valid operator.

R_2 : **Bottom Commutativity**
$(Table_1 \bowtie_0 Table_2) \rightarrow (Table_2 \bowtie_1 Table_1)$.
Disable rule R_2 for application on operator $\bowtie_1$.
$Table_1$ and $Table_2$ are base relation.

Theorem 4.7 *The transformation rules of rule set RS-L_{ac} do not generate duplicates if applied on an initial operator $L \bowtie R$, where R consists of a single base relation and the class for L does not contain duplicates.*

Proof. The application schema defines two distinct derivation paths. Either L and R are both base relations and only rule R_2 applies, or L references more than one relation and R is a base relation so only rule R_1 applies. For each case a proof similar to the proof for Theorem 4.1 can be given. □

Theorem 4.8 *The transformation rules of rule set RS-L_{ac} generate all valid linear join trees if applied on an initial operator $L \bowtie R$, where R references a single relation.*

Proof. All valid root operators of a class have a single base relation as right input. This base relation is a "terminal"in the associated query graph —i.e. it shares exactly one edge with another relation, otherwise the operator is not be valid.
Assume R is a single relation and class L has been explored completely. Now rule R_1 will combine R with each operator of class L resulting a number of new join operators. Only the operators that are valid will be added to class that is being explored. □

abcde	$=[bcde] \bowtie [a]$
bcde	$=[cde] \bowtie [b]; [bce] \bowtie [d]; [bcd] \bowtie [e]$
cde	$=[ce] \bowtie [d]; [cd] \bowtie [e]$
bce	$=[ce] \bowtie [b]; [bc] \bowtie [e]$
bcd	$=[cd] \bowtie [b]; [bc] \bowtie [d]$
bc	$=[b] \bowtie [c]; [c] \bowtie [b]$
ce	$=[c] \bowtie [e]; [e] \bowtie [c]$
cd	$=[c] \bowtie [d]; [d] \bowtie [c]$

Figure 4.7: Partial MEMO-structure with linear trees for acyclic queries

Example 4.4 *Given a query* $G = \{\{a, b, c, d, e\}, \{a - b, b - c, c - d, c - e\}\}$ *and the MEMO-structure as shown in Figure 4.7, where the class "abcde" is about to be explored. Of the initial operator in class "abcde" the child classes have been explored exhaustively.*

Applying the transformation rules of rule set RS-L_{ac} *results in generating the following operators.*

Rule $\mathbf{R_1}$**:** $[acde] \bowtie [b], [abce] \bowtie [d], [abcd] \bowtie [e]$.
The operator $[acde] \bowtie [b]$ *is discarded because there is no predicate that connects relation* a *to either* c, d *or* e.

Rule $\mathbf{R_2}$**:** *This rule can not be applied for class "abcde" since is is only active for classes with exactly two relations.*

The fully explored class "abcde" contains 3 operators. During the exploration process the new classes $[abce]$ *and* $[abcd]$ *are created and, in turn, fully explored.*

4.3 Experiments

In this section we experimentally verify the efficiency improvement of the duplicate-free join enumeration process. For completely connected queries, from 3 to 8 relations, we generated all bushy and linear join trees using the naive transformation rules (RS-B1, RS-L1) and the duplicate-free rules (RS-B_{cc}, RS-L_{cc}).

The measurements have been performed on a 90 MHz Pentium PC running Windows NT. Its main memory was 64Mb, which was more than

enough to contain the largest MEMO-structure — all bushy trees for a query of 8 relations. The measurements have been performed using the Cascades optimizer, which is a descendent of the Volcano optimizer. However no feature was used that wasn't already present in Volcano. The one modification on the domain-independent, transformation-rule kernel was to add the ability to disable transformation rules. The remainder of the logic is done completely within the domain-specific set of transformation rules.

For each unique "logical" join operator we also generated a single "physical" operator (Nested Loop). For each physical operator some cost estimation was done — i.e. cardinality estimate — which makes the generation of a physical operator more expensive than the generation of a logical operator. However, the estimation cost is constant per physical operator and is only performed for ***unique*** operators and not for duplicates. Avoiding the generation of physical operators would make the improvement factor even bigger.

Each experiment has been performed several times and the graphs represent the averages over these runs. The variation amongst the runs was very small, less then 0.5%,

4.3.1 Bushy join trees

For the naive generation of bushy join trees in case of completely connected queries we used the commutativity and associativity rules (rule set RS-B1) as described in Section 3.1.3. In [GD87] it was already observed that the performance of the join enumerator could be improved by applying the commutativity rule only once. This avoids all generation cycles of length two — i.e. cycles like $(a \bowtie b) \rightarrow (b \bowtie a) \rightarrow (a \bowtie b)$. However the improvement is very small, for 8 relations the improvement is no more than 1%. When generating all bushy trees using the naive set of rules, cycles of length 2 were avoided.

Figure 4.8 shows the (scaled) time required to generate all bushy trees for completely connected queries from 3 to 8 relations. The scaling of the graphs is done using the time to generate all bushy trees for a query of three relations, as a reference.

The experiments show that duplicate free generation of join trees is always faster than generating and discarding duplicates. The performance gain increases from a factor 1.22 for three relations to a factor 5.67 for eight relations. Based on the complexity analysis of the generation algorithms the improvement factor will increase further as queries get larger.

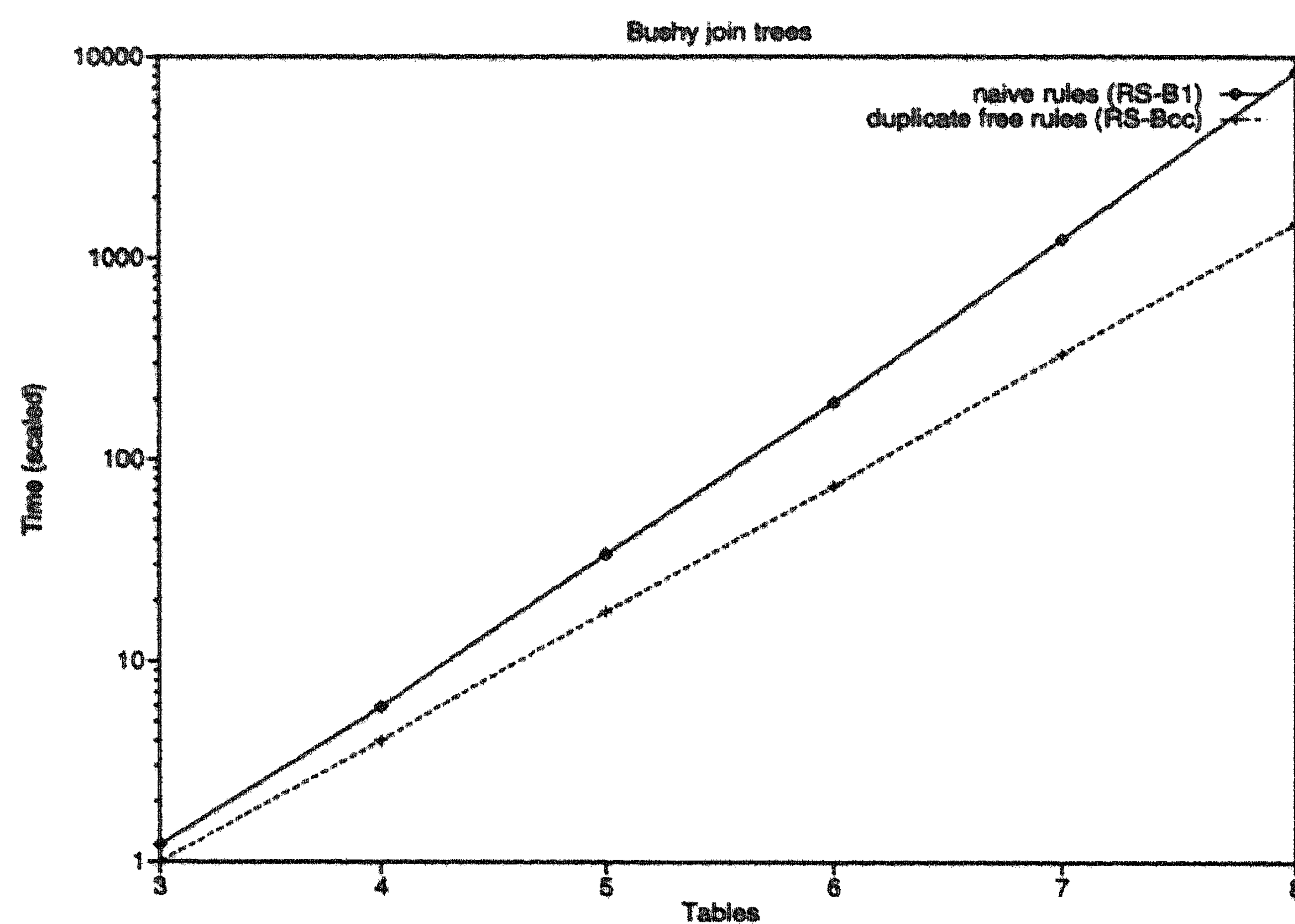


Figure 4.8: Exhaustive generation of bushy trees for completely connected queries.

4.3.2 Linear join trees

The naive method for generating the complete space of linear join trees uses for completely connected queries rule set RS-L1, see Section 3.1.3. As in the naive generation of bushy join trees the commutativity rule is applied only once to avoid cycles of length 2.

Figure 4.9 shows the experimental results for generating all linear trees using the naive method (rule set RS-L1), in which duplicates are generated, and the efficient method that avoids the generation of duplicates (rule set RS-L_{cc}). The time for generating all linear join trees for a query of three relations, using the duplicate-free rules, was used as reference for scaling the graphs. For linear trees, avoiding the generation of duplicates shows a performance improvement of a factor 1.33 to 3.67 for queries from three to eight relations. Again the improvement factor will keep increasing with the number of relations.

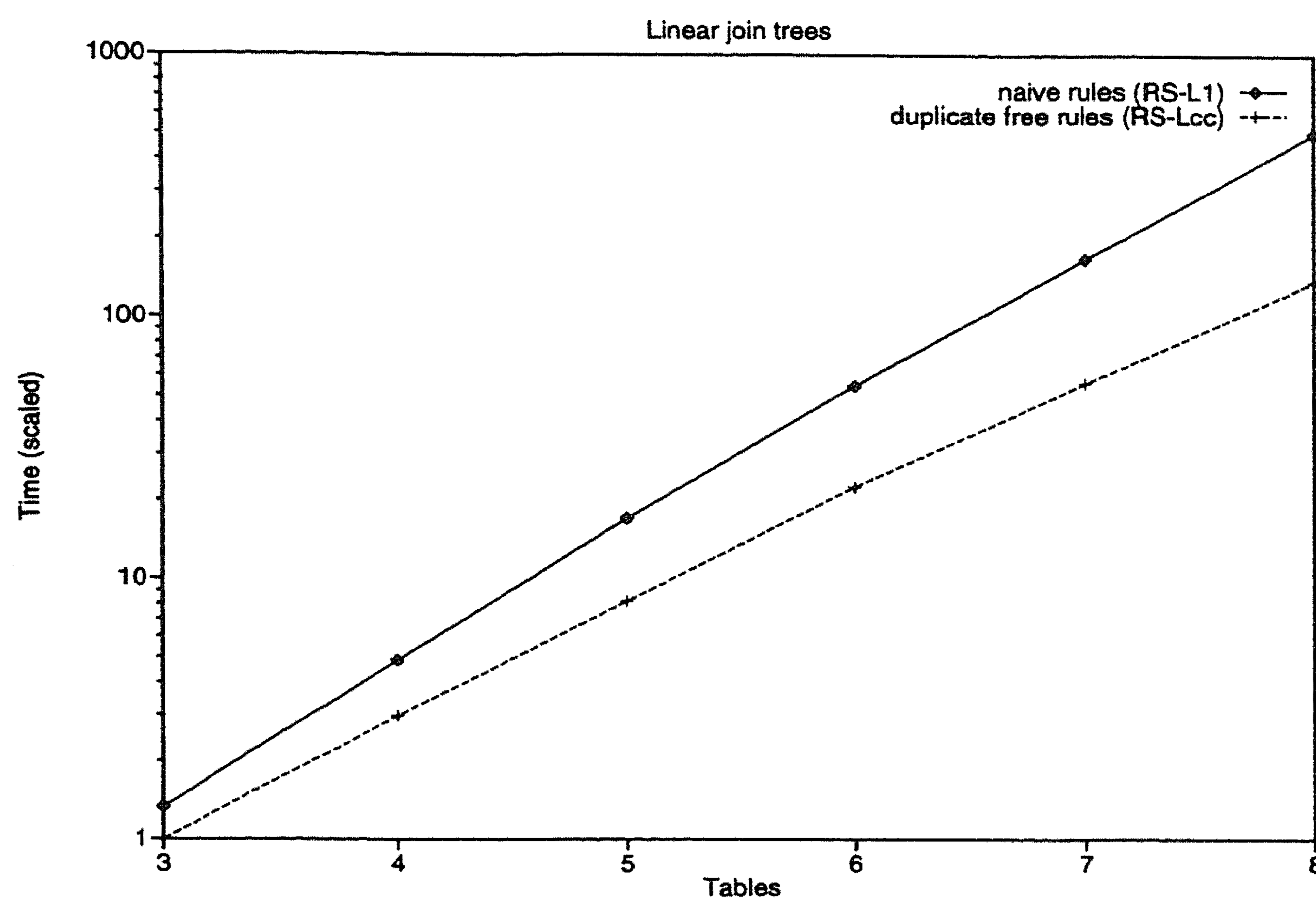


Figure 4.9: Exhaustive generation of linear trees for completely connected queries.

4.4 Summary

In this chapter, we described in detail efficient sets of transformation rules, for several classes of query topologies for both bushy and linear join trees. Our approach to an efficient generation algorithm is to keep track of the transformation rules that can still be applied without generating duplicates at any point in the search space. The overhead of the method consists of a few bits per operator.

The conditioned application of rules can be incorporated easily in the existing framework of modern query optimizers, and preliminary tests corroborate that considerable performance improvements result from the large reduction of generated operators. The performance improvement gained by avoiding duplicate generation is significant in practice, and it should be used whenever possible.

Chapter 5

Counting join trees[1]

For a given query graph the number of valid join trees is, in general, not easy to compute, because the topology of the query graph has to be taken into account. For some query graphs the number of valid join trees can be computed easily, because they are "structured". Fortunately, the upper and lower bounds on the number of join trees are given by such query graphs. The lower bound is set by a *string* graph and the upper bound is set by a *completely connected* graph (See Section 2.1.3).

This chapter answers the question of how to compute the exact size of the space of join trees for *acyclic queries.* The number of join trees is restricted by the relations that can be joined together, and counting them does not reduce, in general, to the enumeration of familiar classes of trees — e. g. binary trees, trees representing equivalent expressions on an associative operator, etc. A variety of techniques is used to enumerate graphs and trees [Knu68, HP73, RH77, VF90]. The scheme we use is similar to that used, for example, in [GLW82], in the sense that an auxiliary structure serves to guide the counting of elements of the space, instead of applying a closed formula.

Previous work has identified restricted classes of queries for which valid operator trees map one-on-one to permutations or unlabeled binary trees — the first class known as *star* queries, and the second as *string* queries, see for example [OL90, IK91]— thus solving the counting and random generation problems for those classes. The work on acyclic queries described in this chapter covers the star and chain queries as particular cases, and provides polynomial time algorithms to count the number of operator trees for a given query.

[1]Parts of this chapter have been published in the *Proceedings of the International Conference on Database Theory, Prague, 1995* [GLPK95]

This chapter is organized as follows. In Section 5.1 we introduce some definitions. In Section 5.2 we describe a standard decomposition graph and primitive operations for constructing join trees. Then in Section 5.3 and 5.4 the recurrence equations for counting bushy and linear join trees are given. A summary is given in Section 5.5.

5.1 Definitions

We assume that query graphs are connected and acyclic, i. e. we deal with ***acyclic queries***. With $\mathcal{T}_G$ we denote the set of join trees of a query graph G, and with $\mathcal{T}_G^{v(k)} \subseteq \mathcal{T}_G$ we denote join trees in which a given leaf v is at level k. For example, for the query graph of Figure 5.1, Figure 5.2 shows that $\mathcal{T}_G$ consists of six trees, $\mathcal{T}_G^{D(1)}$ consists of only two trees, and $\mathcal{T}_G^{B(3)} = \mathcal{T}_G$.

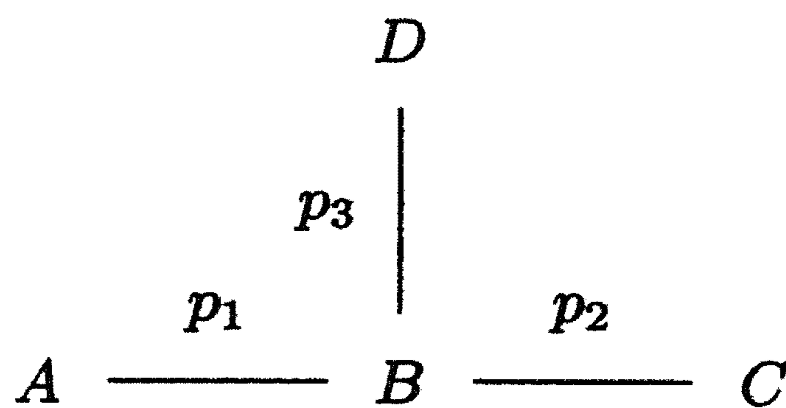


Figure 5.1: Query graph G.

5.1.1 Lists

We introduce some notation and properties of lists that are used later in this chapter. Square brackets are used as lists delimiters, as well as the list construction symbol "|" of Prolog —i. e. $[x|L]$ denotes the list obtained by inserting a new element x at the front of a list L. An array of values $x_0, \ldots, x_n$ in which index i stores value x_i, for $i = 0, \ldots n$, is represented as the list $[x_0, \ldots, x_n]$.

We say that a list L' is the *projection* of a list L on some property P of the elements, if L' contains all the elements of L satisfying P, while also preserving the relative order of L —i. e. if x appears before y in L' then x appears before y in L. We say L is a *merge* of two lists L_1, L_2 without

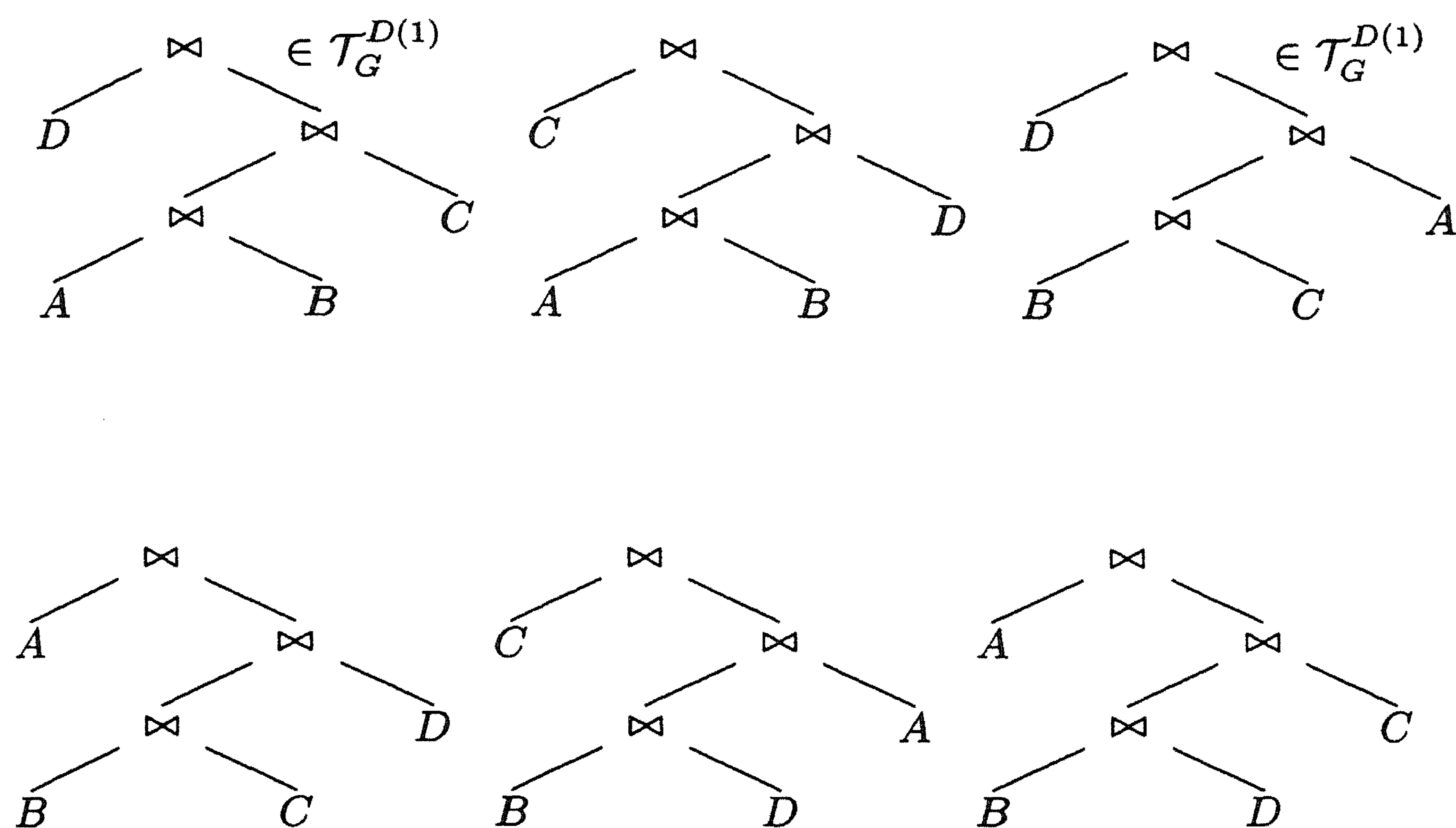


Figure 5.2: All join trees of query graph G.

common elements, if the length of L is the sum of lengths of L_1, L_2, and both L_1, L_2 are projections of L.

Two list L_1 and L_2 of length l_1, l_2, respectively, can be merged in many ways. Each merging is specified by a merging sequence, $\alpha = [\alpha_0, \ldots, \alpha_{l_2}]$. Operationally the merging is as follows. To obtain a merged list L from L_1 and L_2 according the merge sequence $\alpha = [\alpha_0, \ldots, \alpha_{l_2}]$, start with the first α_0 elements of L_1, then use the first element from L_2; now use the next α_1 elements of L_1, then one from L_2. The last α_{l_2} elements of L_1 follow the last element of L_2 in L. Now L is the result of *merging* L_1, L_2 *using* α.

There is a one-on-one mapping between the result of merging L_1, L_2 and the problem of non-negative integer decomposition of l_1 in $l_2 + 1$ —that is, a list of $l_2 + 1$ non-negative integers $[\alpha_0, \ldots, \alpha_{l_2}]$ such that their sum is equal to l_1.

Since the decomposition of n in k can be solved in $\begin{pmatrix} n+k-1 \\ k-1 \end{pmatrix}$ ways

[NW78], there are $M(l_1, l_2) = \binom{l_1 + l_2}{l_2}$ acceptable results of the merge of lists L_1, L_2, each identified with a specific decomposition. Observe that $M(l_1, l_2) = M(l_1, l_2 - 1) + M(l_1 - 1, l_2)$. A table of size $N \times N$ can be constructed in $O(N^2)$ time so that $M(l_1, l_2)$ is found by a simple lookup, for $l_1, l_2 \leq N$.

Example 5.1

nr	α	L
1	$[0,0,0,2]$	$[a,b,c,A,B]$
2	$[0,0,1,1]$	$[a,b,A,c,B]$
3	$[0,1,0,1]$	$[a,A,b,c,B]$
4	$[1,0,0,1]$	$[A,a,b,c,B]$
5	$[0,0,2,0]$	$[a,b,A,B,c]$
6	$[0,1,1,0]$	$[a,A,b,B,c]$
7	$[1,0,1,0]$	$[A,a,b,B,c]$
8	$[0,2,0,0]$	$[a,A,B,b,c]$
9	$[1,1,0,0]$	$[A,a,B,b,c]$
10	$[2,0,0,0]$	$[A,B,a,b,c]$

Figure 5.3: All mergings of lists $[A,B]$ and $[a,b,c]$

Given two list $[A,B]$ and $[a,b,c]$, the table in Figure 5.3 shows all $\binom{2+3}{3} = 10$ merging sequences, α, and the corresponding merging result L.

5.1.2 Anchored-list representation

Since our arguments and constructions often rely on paths from the root of the join tree to a specific leaf, we introduce an *anchored list* representation of trees. Elements of the anchored list are those subtrees observed while traversing the path from the root to some anchor leaf.

Definition 5.1 *Let T be a join tree and v be a leaf of T. The* list anchored on v of T, *call it L, is constructed as follows:*

- *If T is a single leaf, namely v, then $L = []$.*
- *Otherwise, let $T = (T_l \bowtie T_r)$ and assume, without loss of generality, that v is a leaf of T_r. Let L_r be the list of T_r anchored on v. Then $L = [T_l | L_r]$.*

Then we say that $T = (L, v)$.

See Figure 5.4 for an anchored-list representation of join tree $T' = T_1 \bowtie (T_2 \bowtie w)$. Observe that if $T = (L, v)$ is an element of $\mathcal{T}_G^{v(k)}$, then the length of the anchored list L is k.

5.2 Decomposition and construction of trees

5.2.1 Primitive operations

We now describe procedures that relate a join tree of a query graph G with some join trees of subgraphs of G. Applied in one direction, these procedures *construct* a join tree based on smaller join trees; applied in the other direction, they *decompose* join trees.

Leaf insertion. Our first procedure is *leaf insertion*. The idea is that two join trees are related by the insertion/removal of a leaf. The operation is stated as the insertion/removal of a one-leaf tree in the anchored list representation of join trees.

Definition 5.2 *Let* $G = (V, E)$ *be a query graph and* T *be a join tree of* G. *Assume* $v \in V$ *is such that* $G|_{V-\{v\}}$ *is connected, and let* $(v, w) \in E$.

- *Let* $T = ([T_1, \ldots, T_{k-1}, v, T_{k+1}, \ldots, T_n], w)$.
- *Let* $T' = ([T_1, \ldots, T_{k-1}, T_{k+1}, \ldots, T_n], w)$.

We call (T', k) *an* insertion pair *on* v. *We say that* T *is* decomposed into *pair* (T', k) *on* v, *or, equivalently, that* T *is* constructed from *pair* (T', k) *on* v.

Example 5.2

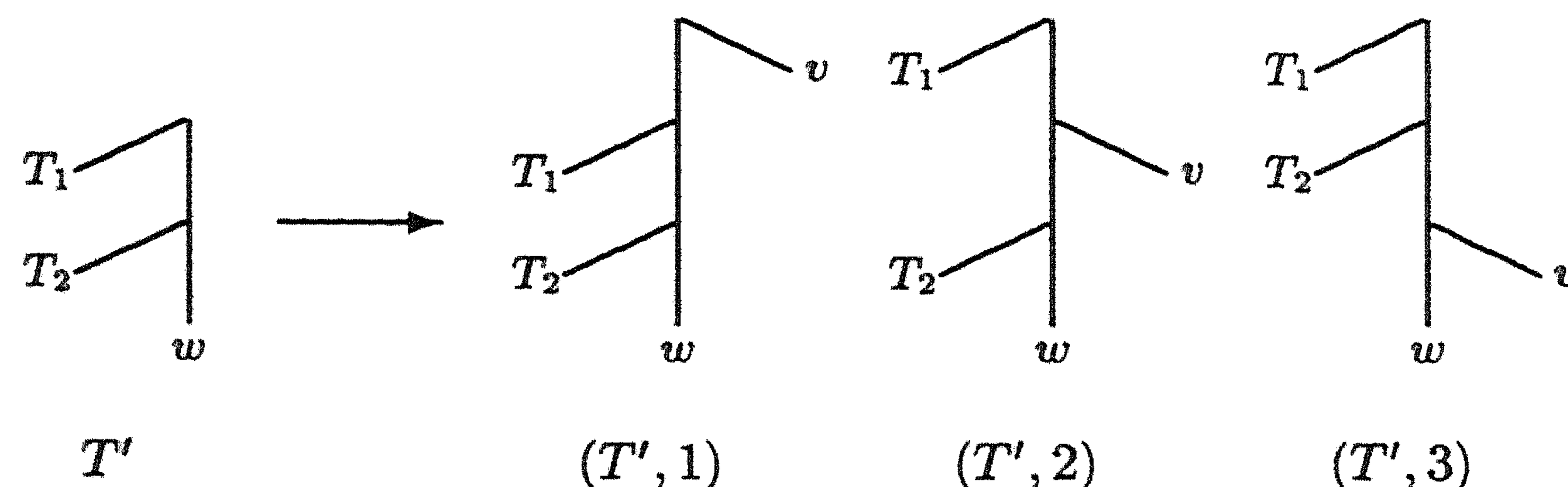


Figure 5.4: Construction by leaf insertion.

Figure 5.4 shows a join tree $T' = ([T_1, T_2], w)$ and the join trees constructed from insertion pairs $(T', 1)$, $(T', 2)$, and $(T', 3)$ on v.

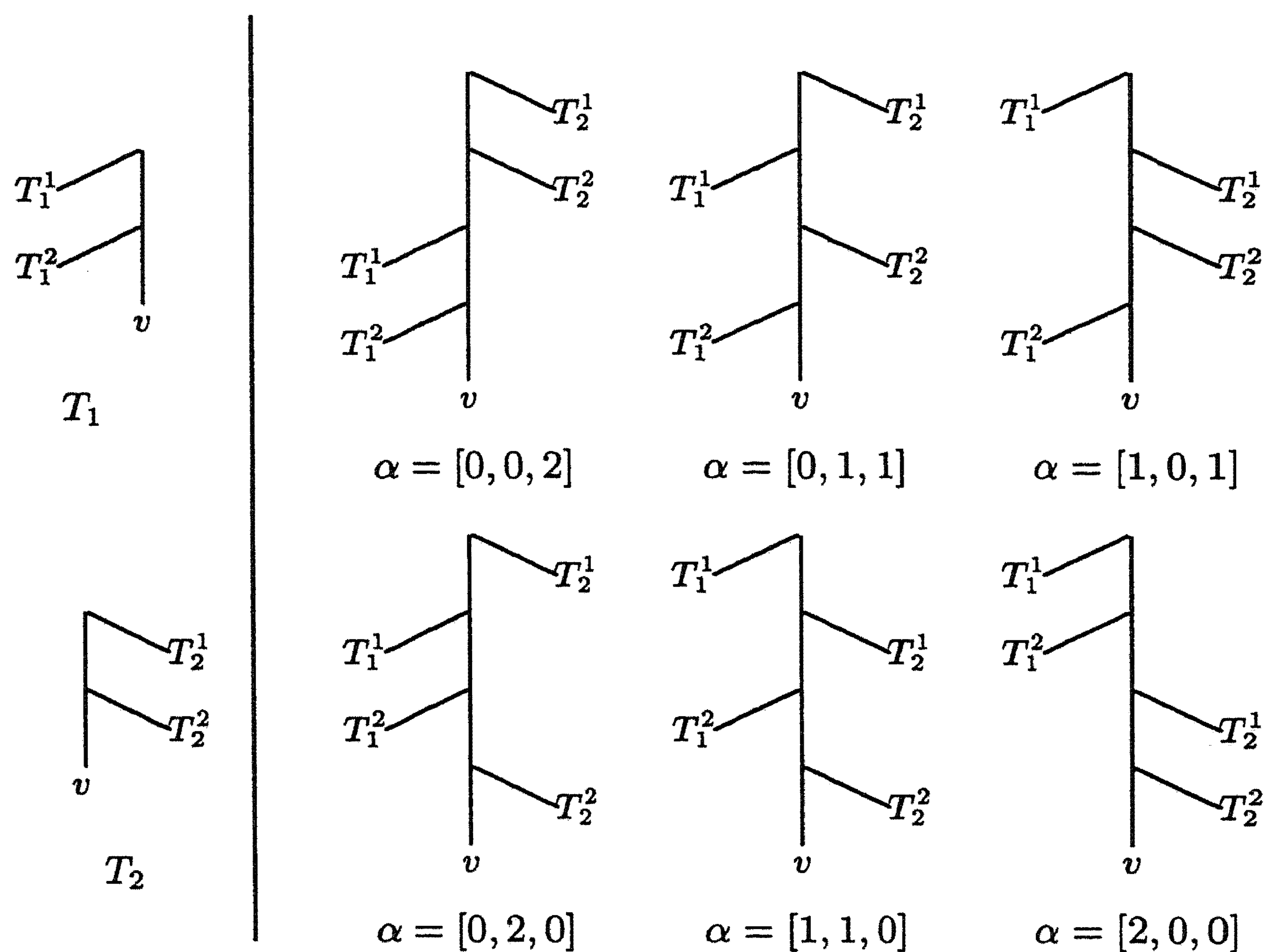


Figure 5.5: All trees resulting from merging T_1 and T_2.

Observation 1 Let $G = (V, E)$ be a query graph with n nodes. Assume $v \in V$ is such that $G' = G|_{V-\{v\}}$ is connected, and let $(v, w) \in E$, and $1 \leq k < n$. The leaf insertion operation defines a one-on-one mapping between elements of $\mathcal{T}_G^{v(k)}$ and insertion pairs on v of the form (T', k), where T' is an element of the disjoint union $\cup_{i=k-1}^{n-2} \mathcal{T}_{G'}^{w(i)}$.

Tree merging. Our second procedure is *tree merging*. The idea is that a join tree can be obtained by merging two smaller join trees. The operation is stated as the merge/projection of the anchored list representation of join trees.

Definition 5.3 *Let $G = (V, E)$ be a query graph and T be a join tree of G. Assume sets of nodes V_1, V_2 are such that $G|_{V_1}, G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{v\}$.*

- *Let $T = ([T_1, \ldots, T_n], v)$.*
- *Define property P_1 (respectively P_2) to be "every leaf of the subtree is in V_1 (V_2)."*
- *Let L_1, L_2 be the projection of L on properties P_1, P_2, respectively. Let α be an integer composition such that L is the result of merging L_1, L_2 using α.*
- *Let $T_1 = (L_1, w)$ and $T_2 = (L_2, w)$.*

We call (T_1, T_2, α) a merge triplet. *We say T is* decomposed into *triplet (T_1, T_2, α) on V_1, V_2, or, equivalently, that T is* constructed from *triplet (T_1, T_2, α) on V_1, V_2.*

Observation 2 Let $G = (V, E)$ be a query graph with n nodes. Assume sets of nodes V_1, V_2 are such that $G_1 = G|_{V_1}, G_2 = G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{v\}$, and let $1 \leq k < n$. The tree merging operation defines a one-on-one mapping between elements of $\mathcal{T}_G^{v(k)}$ and merge triplets on V_1, V_2 of the form (T_1, T_2, α), where $T_1 \in \mathcal{T}_{G_1}^{v(i)}, T_2 \in \mathcal{T}_{G_2}^{v(k-i)}$, and α specifies a merge of two lists of size $i, k - i$ respectively.

Example 5.3 *Figure 5.5 shows the six possible join trees that are the result of merging $T_1 = ([T_1^1, T_1^2], v)$ and $T_2 = ([T_2^1, T_2^2], v)$.*

5.2.2 Standard decompositions

Join trees can be decomposed into a sequence of leaf insertion and tree merging operations, but these decompositions are not unique, in general. A key structure for our algorithms is the *standard decomposition graph*, which is obtained by selecting an arbitrary order of operations to construct the join trees of some graph G. Join decompositions are then unique with respect to the standard order defined.

A standard decomposition graph H of G can be viewed as a generic program (or operator tree) to build join trees of a given query graph. Unary nodes of H, labeled "$+_x$," construct a join tree by inserting a leaf x on its argument; binary nodes of H, labeled "$\times_x$," construct a join tree by merging two trees whose only common leaf is x.

Definition 5.4 *A* standard decomposition graph *H of a query graph $G = (V, E)$ is obtained by modifying G as follows:*

- *Pick a node, say $v \in V$, as* root. *Direct the edges in E from the root v outwards to obtain G'. If there is a directed edge from u to w we say*

```
CONVERT-TO-SDG(v)
   Let x be the label of v.
   Let {w_1, ..., w_n} be the children of v.
   If n = 0
       Label v as "x".
   If n = 1
       Label v as "+_x";
       CONVERT-TO-SDG(w_1).
   If n > 1
       Label v as "×_x";
       create new nodes l, r, with label x;
       delete all edges (v, w_1), ..., (v, w_n);
       create new edges (v, l), (v, r), (l, w_1), (r, w_2), ..., (r, w_n);
       CONVERT-TO-SDG(l), CONVERT-TO-SDG(r).
```

Figure 5.6: Algorithm to obtain a standard decomposition graph.

u is the parent *of w. If there is a directed path (of length zero or more) from u to w we say u is an* ancestor *of w.* Child *and* descendant *are the inverses of parent and ancestor, respectively.*

- *Transform G' using algorithm* CONVERT-TO-SDG*(r), where r is the root chosen earlier. The result of this transformation is H.* CONVERT-TO-SDG *is shown in Figure 5.6.*

The labels of descendants *of a node v in H, denoted $desc(v)$, is the set of node labels $\{w_i\}$ of G that appear in the descendants of v in the form "$\times_{w_i}$," "$+_{w_i}$," or "w_i."*

Example 5.4 *Figure 5.7 shows a query graph and a standard decomposition graph obtained from it. In this case node e was selected as root. The labels of descendants of the node v labeled "$+_b$" in H is $desc(v) = \{a, b\}$. The labels of descendants of "$+_e$" is $\{a, b, c, d, e\}$.*

When an insertion level k is selected at each node labeled "$+_x$," and a merge specification α is selected at each node labeled "$\times_x$," a standard decomposition graph becomes a complete "program" to construct a join tree. The annotations in a graph H necessary to construct T is called the *standard decomposition* of T in H.

Let r be the root of a standard decomposition graph H of G, and let T be a join tree of G. The standard decomposition of T in H is obtained by

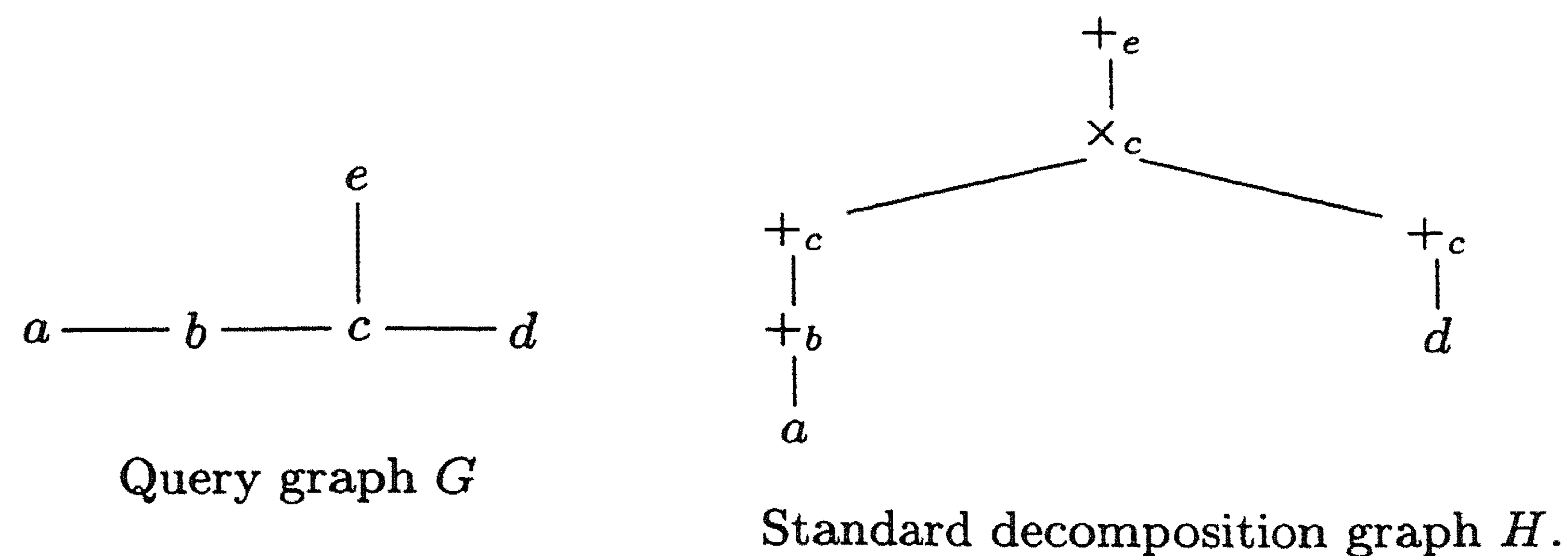


Figure 5.7: Query graph and standard decomposition graph.

applying the procedure DECOMPOSE(T,r), defined in Figure 5.8.

DECOMPOSE(T, v)
  Let $\{w_1, \ldots, w_n\}$ be the children of v in H.
  If $n = 1$
    Let $+_x$ be the label of v.
    decompose T into an insertion pair, say (T', k), on x.
    annotate v as *insert-at* k;
    DECOMPOSE(T, w_1).
  If $n > 1$
    Let V_1, V_2 be the labels of desc(w_1), desc(w_2), respectively.
    decompose T into a merge triplet, say (T_1, T_2, α), on V_1, V_2.
    annotate v as *merge-using* α;
    DECOMPOSE(T_1, w_1), DECOMPOSE(T_2, w_2).

Figure 5.8: Algorithm to obtain the standard decomposition of a tree in H.

Example 5.5 *Figure 5.9 shows the process of obtaining a standard decomposition of a join tree T, using the standard decomposition graph of Figure 5.7. First, the* insert-at *annotation of the root "$+_e$" is 2, because the level of e is* 2 *after being inserted in a smaller tree. This is shown as a label "$+_{e,2}$" in the first row of the figure. Then, the* merge-using *annotation of the node "$\times_c$" is* $[1, 0]$, *because the anchored list on c of the join tree is* $[(a \bowtie b), d]$, *which results from the merge of lists* $[(a \bowtie b)], [d]$ *using* $[1, 0]$. *This is shown as a label "$\times_{c,[1,0]}$" in the second row of the figure. The remanding annotations are obtained similarly.*

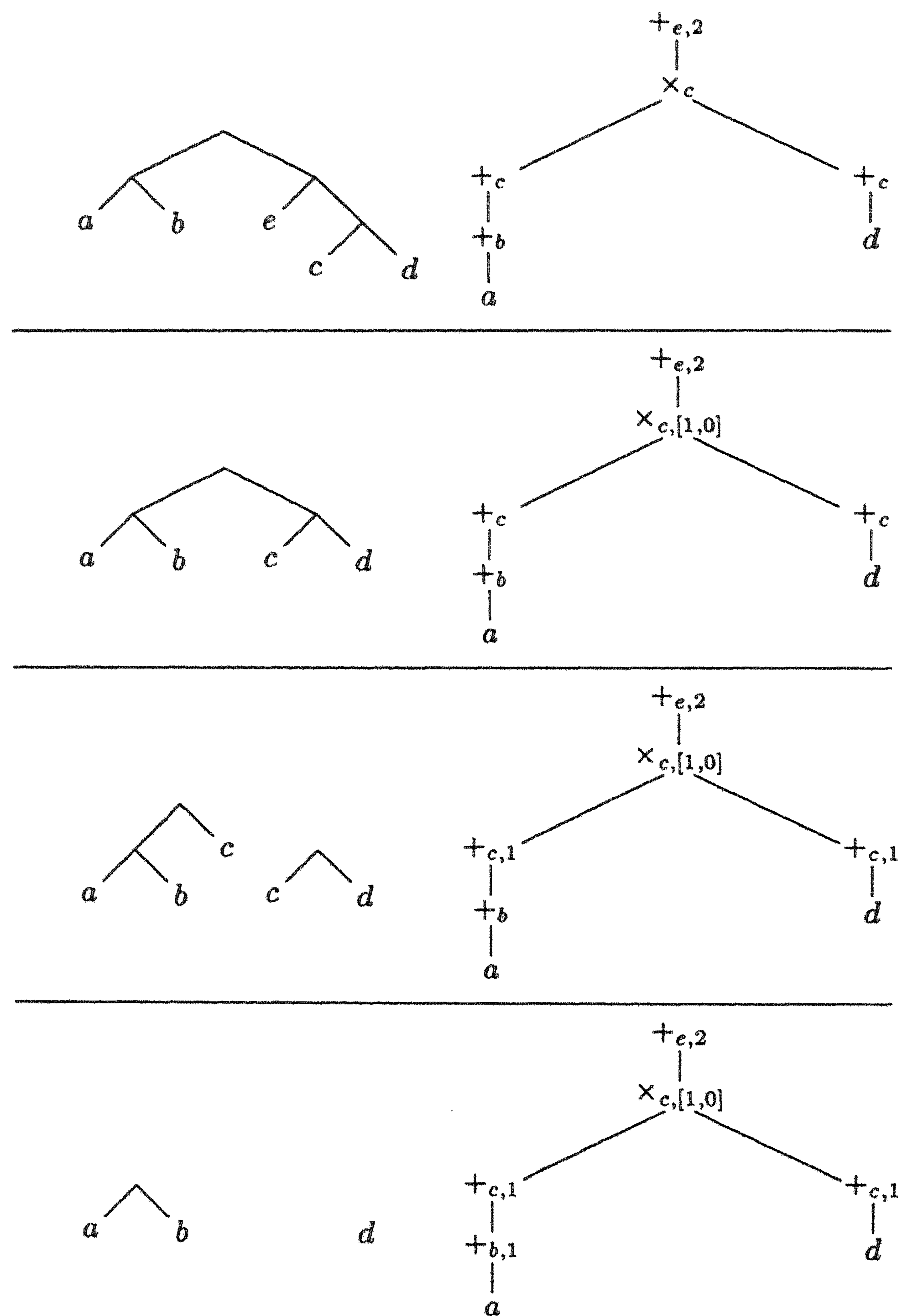


Figure 5.9: Obtaining the standard decomposition of a join tree.

5.3 Counting bushy join trees

Our counting scheme is based on the tree decompositions described in Section 5.2. We first derive recurrence equations relating the number of join trees of a query graph G with the number of trees of subgraphs of G. Then we apply these equations in the context of a standard decomposition graph.

5.3.1 Recurrence equations

Observation 3 The following equations serve as base cases for the computation of the number of join trees of a graph $G = (V, E)$, namely $|\mathcal{T}_G|$. Let $n = |V|$ and $v \in V$.

- If the graph has only one node, then it has only one join tree T, and v is at level 0 in T. That is,

$$|\mathcal{T}_G| = \left|\mathcal{T}_G^{v(0)}\right| = 1, \text{ for } n = 1.$$

- If the graph has more than one node, then it has no join tree in which v is at level 0. That is,

$$\left|\mathcal{T}_G^{v(0)}\right| = 0, \text{ for } n > 1.$$

- There is no join tree in which v is at level greater than or equal to n. That is,

$$\left|\mathcal{T}_G^{v(i)}\right| = 0, \text{ for } i \geq n.$$

- Since v appears at some unique level in any join tree of G, the total number of join trees is

$$|\mathcal{T}_G| = \sum_i \left|\mathcal{T}_G^{v(i)}\right| .$$

Now, the next two lemmas determine the number of join trees that can be constructed using the primitive operations, *leaf insertion* and *tree merging*, as described in Section 5.2.1.

Lemma 5.1 *Let $G = (V, E)$ be a query graph with n nodes. Assume $v \in V$ is such that $G|_{V-\{v\}}$ is connected, and let $(v, w) \in E$, and $1 \leq k < n$. Then*

$$\left|\mathcal{T}_G^{v(k)}\right| = \sum_{i \geq k-1} \left|\mathcal{T}_{G'}^{w(i)}\right| .$$

Proof. The lemma follows from observation 1 in section 5.2.1. □

Lemma 5.2 *Let $G = (V, E)$ be a query graph with n nodes. Assume sets of edges V_1, V_2 are such that $G|_{V_1}, G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{v\}$, and let $1 \leq k < n$. Then*

$$\left|\mathcal{T}_G^{v(k)}\right| = \sum_i \left|\mathcal{T}_{G_1}^{v(i)}\right| \cdot \left|\mathcal{T}_{G_2}^{v(k-i)}\right| \cdot \binom{k}{i} .$$

Proof. The lemma follows from observation 2 in section 5.2.1. □

5.3.2 Counting standard decompositions

The standard decomposition graph defined in Section 5.2.2 is used as an auxiliary structure in the computation of the number of join trees of a query graph. Viewing the standard decomposition graph again as a program (or operator tree), a bottom-up traversal is used to determine how many join trees can be constructed by a given operation, based on the number of trees that its children can construct. At each node we use either Lemma 5.1 for *leaf insertion* or 5.2 for *tree merging* directly to determine the number of trees that can be constructed, and the result is incorporated in the graph as a *count-array* annotation of the node.

For a node u labeled $\odot_v$ (with $\odot \in \{+, \times\}$), the *count-array* annotation has the form $[x_0, \ldots, x_n]$. The interpretation is that node u can construct x_i different trees in which leaf v is at level i. To determine the total number of join trees for a query, just sum all entries of the *count-array* annotation in the root of the standard decomposition graph.

Let r be the root of a standard decomposition graph H. To find the *count-array* annotations of H apply the procedure COUNT-JT(r), defined in Figure 5.10.

Example 5.6 *Figure 5.11 shows the* count-array *annotations on the decomposition graph of Figure 5.7. The total number of different join trees for this query is 18.*

Theorem 5.1 *The number of join trees of a given acyclic query graph G with n nodes can be computed in $O(n^3)$ time.*

Proof. A standard decomposition graph H of G can be constructed in linear time using algorithm CONVERT-TO-SDG in Figure 5.6. The number of nodes of H is linear on n. The *count-array* annotations in H are obtained using algorithm COUNT-JT in Figure 5.10 in $O(n^3)$ time, since the computation required per node is quadratic at worst. Finally, the number of join trees of G is the sum of the $O(n)$ values in the *count-array* of the root of H. □

COUNT-JT(v)
  Let $\{w_1, \ldots, w_n\}$ be the children of v.
  If $n = 0$
    annotate v with *count-array* $[1]$.
  If $n = 1$
    COUNT-JT(w_1);
    let $X = [x_0, \ldots, x_{n_1}]$ be the *count-array* of w_1;
    annotate v with *count-array* $Z = [0, z_1, \ldots, z_{n_1+1}]$,
    where Z =INSERT_B(X)
  If $n = 2$
    COUNT-JT(w_1), COUNT-JT(w_2);
    let $X = [x_0, \ldots, x_{n_1}]$ be the *count-array* of w_1;
    let $Y = [y_0, \ldots, y_{n_2}]$ be the *count-array* of w_2;
    annotate v with *count-array* $Z = [z_0, \ldots, z_{n_1+n_2}]$,
    where Z =MERGE_B(X, Y).

Z=INSERT_B(X)
  let $z_k = \sum_{i=k-1}^{n_1} x_i$, for $k = 1, \ldots, n_1 + 1$.
  With $Z = [0, z_1, \ldots, z_{n_1+1}]$,
    $X = [x_0, \ldots, x_{n_1}]$.

Z=MERGE_B(X, Y)
  let $z_k = \sum_{i=0}^{n_1} x_i \, y_{k-i} \binom{k}{i}$, for $k = 1, \ldots, n_1 + n_2$.[a]
  With $Z = [z_0, \ldots, z_{n_1+n_2}]$,
    $X = [x_0, \ldots, x_{n_1}]$,
    $Y = [y_0, \ldots, y_{n_2}]$.

[a] To simplify the description, we assume that $y_i = 0$ for $i \notin \{0, \ldots, n_2\}$.

Figure 5.10: Algorithm to count the number of join trees.

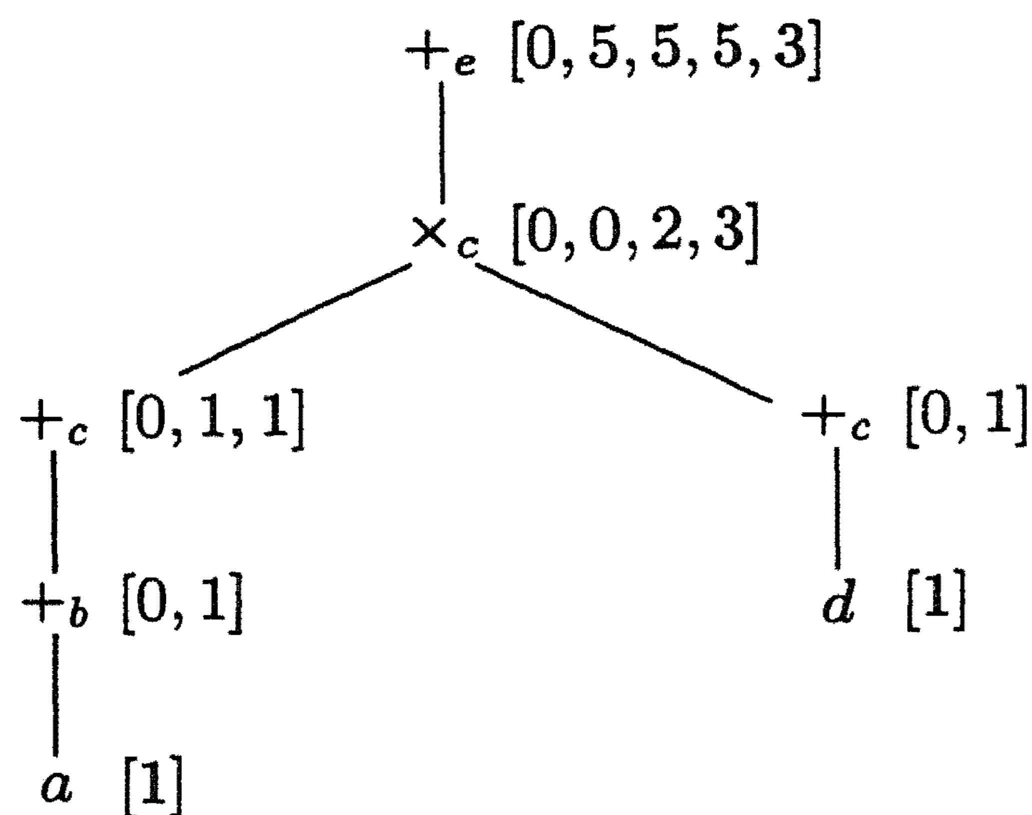


Figure 5.11: Standard decomposition graph with *count-array* annotations.

5.4 Counting linear join trees

The trees considered so-far are all bushy. Some systems reduce the search space by only considering linear join trees in order to reduce the size of the search space or due to limitation of the execution engine. In the next two sections we show how the "bushy" counting algorithms can be modified in order to count linear join trees.

5.4.1 Recurrence equations

The recurrence equations for linear join trees can be derived similarly as for the bushy join trees. First, call $\mathcal{L} \subseteq \mathcal{T}$ the space of linear join trees and note that the base cases for the linear join trees are identical to the base cases of the bushy join trees, observation 3 in Section 5.3.1. For the linear trees, there are additional observations that limit the number of trees that can be constructed by the *leaf insertion* and *tree merging* operations.

Observation 4 Let L be a linear join tree of $G = (V, E)$ with $w \in V$ and $|V| = n$. Then L can be represented as the anchored-list on w as follows: $L = ([L_1, \ldots, L_{k-1}, L_k], w)$. Since L is a linear tree $L_1 \ldots L_{k-1}$ are all leafs, and if w is at level $n-1$ — the bottom of the tree — then also L_k is a leaf, otherwise L_k is a linear sub-tree of at least two leafs.

Observation 5 To construct a linear tree L by merging two linear trees L_1 and L_2, the common leaf v of L_1 and L_2 has to be at the bottom of either L_1 or L_2, otherwise the resulting tree L is not linear.

Now, the next two lemmas give the recurrence equations for determining the number of linear join trees that can be constructed using the primitive operations — *leaf insertion* and *tree merging* — of Section 5.2.1.

Lemma 5.3 *Let $G = (V, E)$ be a query graph with n nodes. Assume $v \in V$ is such that $G|_{V-\{v\}}$ is connected, and let $(v, w) \in E$. Then*

$$\left|\mathcal{L}_G^{v(k)}\right| = \sum_{i \geq k} \left|\mathcal{L}_{G'}^{w(i)}\right| \quad \textit{for } 1 \leq k < n-1$$

And

$$\left|\mathcal{L}_G^{v(k)}\right| = \left|\mathcal{L}_{G'}^{w(k-1)}\right| \quad \textit{for } k = n-1$$

Proof. The lemma follows from observation 1 in Section 5.2.1 and observation 4 in Section 5.4.1. □

Lemma 5.4 *Let $G = (V, E)$ be a query graph with n nodes. Assume sets of edges V_1, V_2 are such that $G|_{V_1}, G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \{v\}$, $n_1 = |V_1|$, $n_2 = |V_2|$ and $n1 \leq n2$.*
Then

$$\left|\mathcal{L}_G^{v(k)}\right| = 0, \quad \textit{for } 0 \leq k < n_1,$$

$$\left|\mathcal{L}_G^{v(k)}\right| = \left|\mathcal{L}_{G_1}^{v(n_1-1)}\right| \cdot \left|\mathcal{L}_{G_2}^{v(k-n_1+1)}\right| \cdot \binom{k-1}{k-n_1}, \quad \textit{for } n_1 \leq k < n_2.$$

And

$$\begin{aligned}\left|\mathcal{L}_G^{v(k)}\right| = & \left|\mathcal{L}_{G_1}^{v(n_1-1)}\right| \cdot \left|\mathcal{L}_{G_2}^{v(k-n_1+1)}\right| \cdot \binom{k-1}{k-n_1} + \\ & \left|\mathcal{L}_{G_1}^{v(k-n_2+1)}\right| \cdot \left|\mathcal{L}_{G_2}^{v(n_2-1)}\right| \cdot \binom{k-1}{k-n_2}, \quad \textit{for } n_2 \leq k < n.\end{aligned}$$

Proof. The lemma follows from observation 2 in Section 5.2.1 and observation 5 in Section 5.4.1. □

5.4.2 Standard decompositions

Counting the number of linear join trees is similar to counting the number of bushy join trees, except Lemma 5.3 and 5.4 are now used. These lemmas compute the *count-array* annotations at the nodes of the standard decomposition graph.

Z =INSERT_L(X)
Let $z_k = \sum_{i=k}^{n_1} x_i$, for $k = 1, \ldots, n_1$
and $z_{n_1+1} = x_{n_1}$.
With $X = [x_0, \ldots, x_{n_1}]$,
$Z = [0, z_1, \ldots, z_{n_1+1}]$.

Z =MERGE_L(X, Y)
$z_i = 0$, for $i = 0, \ldots, n_1 + n_2$;

$$z_{n_1+i} = z_{n_1+i} + x_{n_1} y_i \binom{n_1 + i - 1}{i - 1}, \text{ for } i = 1, \ldots, n_2;$$

$$z_{n_2+i} = z_{n_2+i} + x_i y_{n_2} \binom{n_2 + i - 1}{i - 1}, \text{ for } i = 1, \ldots, n_1.$$

With $X = [x_0, \ldots, x_{n_1}]$,
$Y = [y_0, \ldots, y_{n_2}]$,
$Z = [z_0, \ldots, z_{n_1+n_2}]$.

Figure 5.12: INSERT and MERGE functions for counting the number of linear trees.

For the COUNT-JT algorithm of Figure 5.10 to count the number of linear trees the functions INSERT_B and MERGE_B have to be replaced by their linear variants as defined in Figure 5.12. The number of linear trees is computed by applying the modified COUNT-JT algorithm to the root of a standard decomposition graph and sum the values of the root's *count-array*.

Example 5.7 *Figure 5.13 shows the* count-array *annotations on the decomposition graph of Figure 5.7 when only the linear trees are counted. The total number of different linear join trees for this graph is 14.*

Theorem 5.2 *The number of linear join trees of a given acyclic graph G with n nodes can be computed in $O(n^2)$ time.*

Proof. This proof is analogous to the proof of Theorem 5.1. However, the *count-array* annotations at each node can now be computed in linear time at worst. And the number of nodes in the standard decomposition graph is linear on n, so the *count-array* annotations are computed in $O(n^2)$ time. Finally the total number of trees is computed by summing the $O(n)$ values in the *count-array* of the root. □

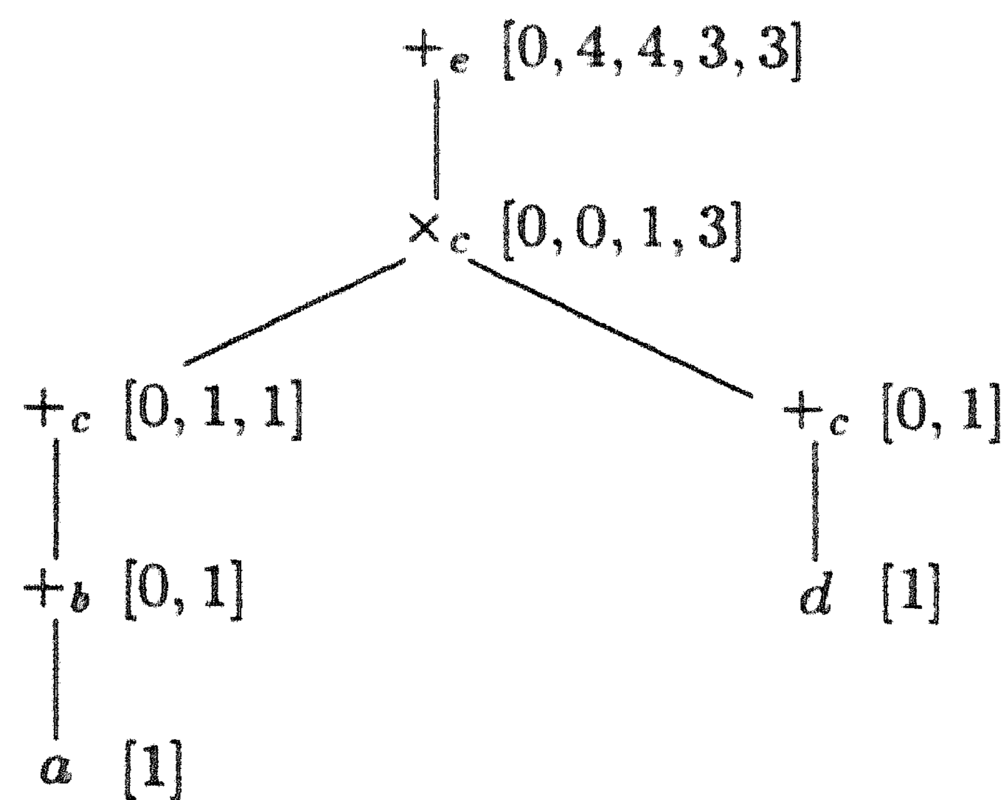


Figure 5.13: Standard decomposition graph with *count-array* annotations for linear trees.

5.5 Summary

This chapter described techniques and procedures for counting the number of bushy and linear join trees that can be used to evaluate an acyclic query. The counting problem results from the fact that there is no one-on-one mapping between join trees and a simple combinatorial structure.

Our concept of a standard decomposition graph provides a supporting structure for counting, because it defines a canonical construction for each tree. In addition, computing an array of values that characterizes the number of canonical constructions can be computed bottom up in an efficient way.

Priority was given to clarity over efficiency when describing the algorithms, and the reader must be aware that there are obvious optimizations. None of those optimizations, however, seems to improve the time bounds stated by the theorems.

Chapter 6

(Un)ranking and random generation[1]

This chapter shows how efficient ranking and unranking functions can be made using the counting techniques of Chapter 5. Together with a source of random bits, the unranking function allows for efficient generation of join trees at random with a uniform distribution.

This has a direct application to randomized query optimization, as selection of a random item in the search space is a basic primitive for most randomized algorithms [SG88, Swa89b, Swa89a, IK90, IK91, Kan91, LVZ93].

This chapter is organized as follows. First in Section 6.1 the mapping of trees to integers is described using the theory developed for counting join trees. Then in Section 6.2 several alternative methods for generating join trees at random are discussed and the straight forward method for generating random join trees using unranking is described. In Section 6.3 an improved method for generating join trees at random is introduced. Finally Section 6.4 concludes with a summary.

6.1 Ranking and unranking

6.1.1 Mapping trees to integers

Our mapping between the N join trees of a query graph and the integers 1 through N is based on the recursive application of the following idea. Assume we want to rank an element $x \in S$, and S is partitioned into sets

[1]Parts of this chapter have been published in the *Proceedings of the International Conference on Database Theory, Prague, 1995* [GLPK95]

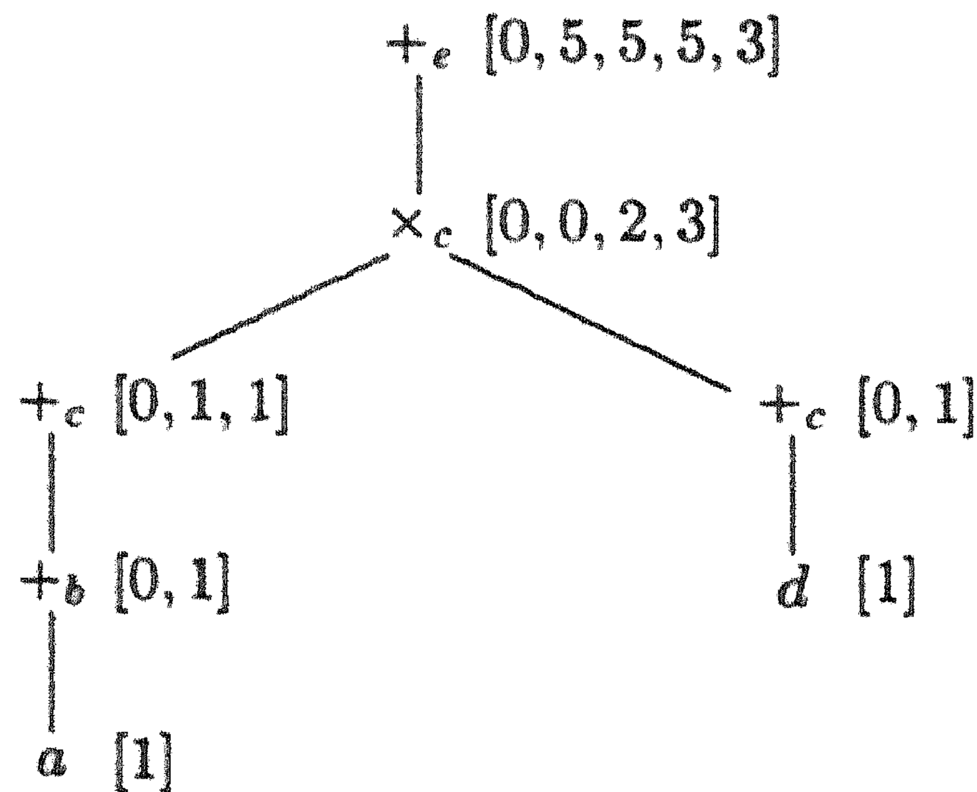


Figure 6.1: Standard decomposition graph with *count-array* annotations.

$S_0, \ldots, S_m$. If $x \in S_k$, for some $k \leq m$, and we can find a *local rank* of x in S_k, then we can set the rank of x in S to be local-rank$(x, S_k) + \sum_{i=0}^{k-1} |S_i|$. Conversely, to unrank some number y under our scheme, first find the set S_k from which the element must be retrieved, where $k = \min_j \; y \leq \sum_{i=0}^{j} |S_i|$. Then find the local rank $y' = y - \sum_{i=0}^{k-1} |S_i|$, and finally unrank-local(y', S_k).

RANK(T)
 Let v be the root of the standard decomposition graph.
 DECOMPOSE(T, v).
 LOCAL-RANK(v).
 Let (r, k) be the *local-rank* of v.
 Let $[z_0, \ldots, z_n]$ be the *count-array* of v.
 Return $r + \sum_{i=0}^{k-1} z_i$.

Figure 6.2: Ranking algorithm.

In the case of join trees of a query of n relations, the set $\mathcal{T}_G$ is partitioned into sets $\mathcal{T}_G^{v(0)}, \ldots, \mathcal{T}_G^{v(n-1)}$, for any given leaf v. For example, for the annotated standard decomposition graph of Figure 6.1, the numbers 1 through 5 are assigned to join trees in which leaf e is at level 1; numbers 6 through 10 are assigned to those in which e is at level 2; numbers 11 through 15 are assigned to those in which e is at level 3; and finally 16 through 18 are assigned to those in which e is at level 4.

Figures 6.2 and 6.3 shows algorithms to rank and unrank trees, based

UNRANK(r)
Let v be the root of the standard decomposition graph.
Let $[z_0, \ldots, z_n]$ be the *count-array* of v.
Let k be $\min_j \; r \leq \sum_{i=0}^{j} z_i$.
Let r' be $r - \sum_{i=0}^{k-1} z_i$.
LOCAL-UNRANK(v, r', k).
The resulting annotations *insert-at* and *merge-using* define the tree whose rank is r.

Figure 6.3: Unranking algorithm.

on a new annotation *local-rank* in the standard decomposition graph, as well as procedures LOCAL-RANK and LOCAL-UNRANK described below.

The procedure LOCAL-RANK works on a standard decomposition graph H of a query graph G, with annotations *insert-at* and *merge-using* that define a tree T. In addition, H must also have annotations *count-array*. The procedure creates annotations *local-rank* on the nodes of the graph. The interpretation of a *local-rank* annotation of the form (r, k) in the root $\odot_v$ of H is that T has local rank r in the set $\mathcal{T}_G^{v(k)}$.

For the same graph H of G, but without *insert-at* and *merge-using* annotations, the procedure LOCAL-UNRANK finds (the *insert-at* and *merge-using* annotations that define) a tree with rank r in $\mathcal{T}_G^{v(k)}$, given r, k as input.

6.1.2 Local ranking

For the local ranking of a tree, we again use the standard decomposition graph and the primitive tree construction operations of section 5.2.1. The summands used to compute $\left|\mathcal{T}_G^{v(k)}\right|$ in Lemmas 5.1 and 5.2 correspond to well-defined subsets of $\mathcal{T}_G^{v(k)}$. The partition defined by those subsets is appropriate for our needs.

Observation 6 Let $G = (V, E)$ be a query graph with n nodes. Assume $v \in V$ is such that $G' = G|_{V-\{v\}}$ is connected, and let $(v, w) \in E$, and $1 \leq k < n$. The set $\mathcal{T}_G^{v(k)}$ can be partitioned into sets $\mathcal{T}_G^{v(k),k-1}, \mathcal{T}_G^{v(k),k}, \ldots, \mathcal{T}_G^{v(k),n-2}$, where $T \in \mathcal{T}_G^{v(k),i}$ if $T \in \mathcal{T}_G^{v(k)}$, the insertion pair on v of T is

(T', k), and $T' \in \mathcal{T}_{G'}^{w(i)}$. The size of each partition is

$$\left|\mathcal{T}_G^{v(k),i}\right| = \left|\mathcal{T}_{G'}^{w(i)}\right| .$$

Observation 7 Let $G = (V, E)$ be a query graph with n nodes. Assume sets of edges V_1, V_2 are such that $G_1 = G|_{V_1}, G_2 = G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{v\}$, and let $1 \leq k < n$. The set $\mathcal{T}_G^{v(k)}$ can be partitioned into sets $\mathcal{T}_G^{v(k),0}, \mathcal{T}_G^{v(k),1}, \ldots, \mathcal{T}_G^{v(k),k}$, where $T \in \mathcal{T}_G^{v(k),i}$ if $T \in \mathcal{T}_G^{v(k)}$, the merge triplet on V_1, V_2 of T is (T_1, T_2, α), and $T_1 \in \mathcal{T}_{G_1}^{v(i)}$. The size of each partition is

$$\left|\mathcal{T}_G^{v(k),i}\right| = \left|\mathcal{T}_{G_1}^{v(i)}\right| \cdot \left|\mathcal{T}_{G_2}^{v(k-i)}\right| \cdot \binom{k}{i} .$$

Just as the annotations *count-array* provide the necessary set partition information in the RANK and UNRANK procedures of Figures 6.2 and 6.3, we use a new annotation *summands* to store information about the partitions introduced in observations 6 and 7.

The *summands* annotation is an array $\sigma = [\sigma_0, \ldots, \sigma_n]$, whose elements in turn are arrays of the form $\sigma_k = [\sigma_{k0}, \ldots, \sigma_{km}]$. If σ is the *summands* annotation of a node whose *count-array* annotation is $[x_0, \ldots, x_m]$, then it holds that $x_k = \sum_{i=0}^{m} \sigma_{ki}$, for $k = 0, \ldots, n$.

Let r be the root of a standard decomposition graph H. To find the *summands* annotations of H apply the procedure SUMMANDS(r), defined in Figure 6.4.

Algorithms LOCAL-RANK and LOCAL-UNRANK are shown in Figures 6.5 and 6.6, respectively. They implement recursively the idea of ranking in terms of set partitions, whose one-level version is the basis of RANK and UNRANK. The necessary information is stored in the *count-array* and *summands* annotations.

The relatively straightforward procedures RANK-DECOMPOSITION and UNRANK-DECOMPOSITION can be implemented efficiently using the lookup table that stores the number of possible mergings of two lists, see Section 5.1.1. The procedures are shown in Figure 6.7 and 6.8.

The procedure RANK-TRIPLET$(a, b, c; A, B, C)$ computes the rank r of a triplet (a, b, c) from the set $\{(x, y, z) \mid 1 \leq x \leq A, 1 \leq y \leq B, 1 \leq z \leq C\}$, and its inverse is UNRANK-TRIPLET$(r; A, B, C)$. These are also straightforward.

SUMMANDS(v)
Let $\{w_1, \ldots, w_n\}$ be the children of v.
If $n = 0$
there is no *summands* annotation in v.
If $n = 1$
SUMMANDS(w_1);
let $[x_0, \ldots, x_{n_1}]$ be the *count-array* of w_1;
annotate v with *summands* $[\sigma_0, \sigma_1, \ldots, \sigma_{n_1+1}]$,
where $\sigma_k = [\sigma_{k0}, \sigma_{k1}, \ldots, \sigma_{kn_1}]$, for $k = 1, \ldots, n_1 + 1$,
and $\sigma_{ki} = \begin{cases} x_i & \text{if } 0 < k \text{ and } k - 1 \leq i; \\ 0 & \text{otherwise.} \end{cases}$
If $n = 2$
SUMMANDS(w_1), SUMMANDS(w_2);
let $[x_0, \ldots, x_{n_1}]$ be the *count-array* of w_1;
let $[y_0, \ldots, y_{n_2}]$ be the *count-array* of w_2;
annotate v with *summands* $[\sigma_0, \sigma_1, \ldots, \sigma_{n_1+n_2}]$,
where $\sigma_k = [\sigma_{k0}, \sigma_{k1}, \ldots, \sigma_{kn_1}]$, for $k = 1, \ldots, n_1 + n_2$,
and $\sigma_{ki} = \begin{cases} x_i \, y_{k-i} \begin{pmatrix} k \\ i \end{pmatrix} & \text{if } 0 \leq k - i \leq n_2; \\ 0 & \text{otherwise.} \end{cases}$

Figure 6.4: Algorithm to compute set partition information.

LOCAL-RANK(v)
  Let $\{w_1, \ldots, w_n\}$ be the children of v.
  If $n = 0$
    annotate v with *local-rank* $(1, 0)$.
  If $n = 1$
    LOCAL-RANK(w_1);
    let (r_1, k_1) be the *local-rank* of w_1;
    let k be the *insert-at* of v;
    let $[\sigma_0, \ldots, \sigma_n]$ be the *summands* of v;
    annotate v with *local-rank* (r, k), where $r = r_1 + \sum_{i=0}^{k_1-1} \sigma_{ki}$.
  If $n = 2$
    LOCAL-RANK(w_1), LOCAL-RANK(w_2);
    let (r_1, k_1) be the *local-rank* of w_1;
    let (r_2, k_2) be the *local-rank* of w_2;
    let $[x_0, \ldots, x_{n_1}]$ be the *count-array* of w_1;
    let $[y_0, \ldots, y_{n_2}]$ be the *count-array* of w_2;
    let k be $k_1 + k_2$;
    let α be the *merge-using* of v;
    let q be RANK-TRIPLET $(r_1, r_2,$RANK-DECOMPOSITION$(\alpha),$
      $x_{k_1}, y_{k_2}, \binom{k}{i})$.
    annotate v with *local-rank* (r, k), where $r = q + \sum_{i=0}^{k_1-1} \sigma_{ki}$.

Figure 6.5: Algorithm for local ranking.

```
LOCAL-UNRANK(v, r, k)
  Let {w_1, ..., w_n} be the children of v.
  If n = 0
      arguments are consistent if r = 1, k = 0;
      there is no additional annotation on v.
  If n = 1
      let [z_0, ..., z_n] be the count-array of v;
      let [σ_0, ..., σ_n] be the summands of v;
      arguments are consistent if k ≤ n, r ≤ z_k;
      let k_1 be min_j r ≤ Σ_{i=0}^{j} σ_{ki};
      let r_1 be r − Σ_{i=0}^{k_1−1};
      annotate v with insert-at k;
      LOCAL-UNRANK(w_1, r_1, k_1).
  If n = 2
      let [z_0, ..., z_n] be the count-array of v;
      let [σ_0, ..., σ_n] be the summands of v;
      let [x_0, ..., x_{n_1}] be the count-array of w_1;
      let [y_0, ..., y_{n_2}] be the count-array of w_2;
      arguments are consistent if k ≤ n, r ≤ z_k;
      let k_1 be min_j r ≤ Σ_{i=0}^{j} σ_{ki};
      let k_2 be k − k_1;
      let q be r − Σ_{i=0}^{k_1−1};
      let (r_1, r_2, a) be UNRANK-TRIPLET(q; x_{l_1}, y_{l_2}, (k choose i)).
      let α be UNRANK-DECOMPOSITION(a, n_1, n_2);
      annotate v with merge-using α;
      LOCAL-UNRANK(w_1, r_1, k_1), LOCAL-UNRANK(w_2, r_2, k_2).
```

Figure 6.6: Algorithm for local unranking.

```
RANK-DECOMPOSITION(α)
    Let α be [α_0, α_1, ..., α_n]
    Let l_1 = Σα; Let l_2 = |α| − 1;
    Let rank = 1;
    For i = 0...n do
        If α_i ≠ 0
            rank = rank + M(l_1, l_2) − M(l_1 − α_i, l_2);
            l_1 = l_1 − α_i;
        l_2 = l_2 − 1;
```

Figure 6.7: Algorithm for RANK-DECOMPOSITION.

```
UNRANK-DECOMPOSITION(r, l_1, l_2)
    Let α_i = 0 for i = 0, ..., n;
    Let i = 0;
    While (r > 1) and (l_1 > 0) do
        c = M(l_1, l_2 − 1);
        If r ≥ c
            r = r − c;
            α_i = α_i + 1;
            l_1 = l_1 − 1;
        Else
            i = i + 1;
            l_2 = l_2 − 1
    α_n = l_1;
```

Figure 6.8: Algorithm for UNRANK-DECOMPOSITION.

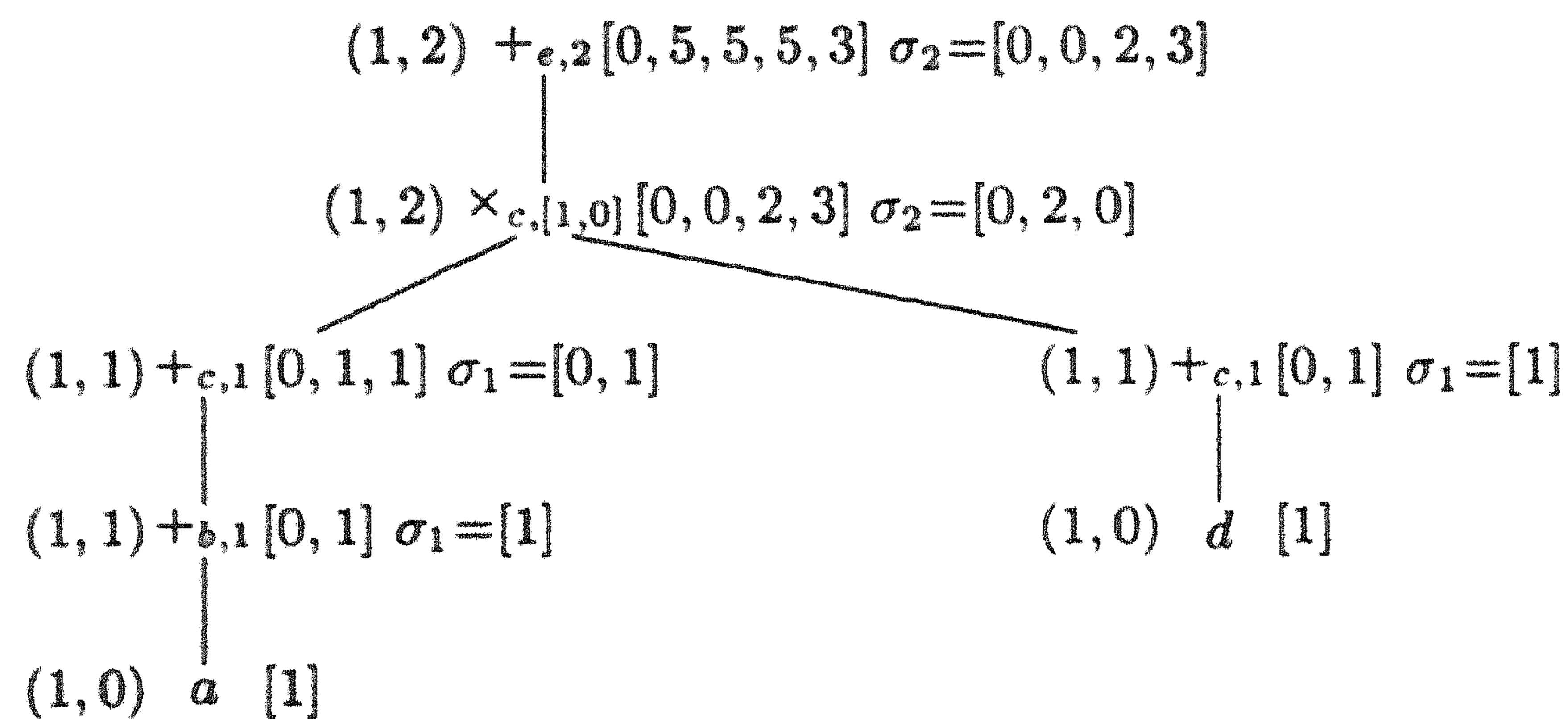


Figure 6.9: Local ranking of (the standard decomposition of) a join tree.

Example 6.1 *Figure 6.9 shows the results of the local ranking for the tree T of example 5.5. The graph contains the annotations of the standard decomposition of T shown in Figure 5.9, the* count-array *annotations of Figure 5.11, and the* summands *annotations computed by* SUMMANDS. *Procedure* LOCAL-RANK *is used to compute annotations* local-rank. *The annotations of a node "$\odot_v$" of the decomposition graph are shown in the figure in the format*

$$(k, l)\ \odot_{vp}\ [x_0, \ldots, x_n]\ \sigma_k = [\sigma_{k0}, \ldots, \sigma_{km}],$$

where p is the insert-at *annotation if $\odot = +$, or else the* merge-using *annotation if $\odot = \times$; $[x_0, \ldots, x_n]$ is the* count-array *annotation; $\sigma = [\sigma_0, \ldots, \sigma_l]$ is the* summands *annotation, but only σ_k is shown. Finally, (k, l) is the local rank computed at the node; that is, the subtree computed at the given node has rank k in set $T_{G'}^{v(l)}$.*

At node $\times_c$ the merge-using annotation *is* $[1, 0]$. *For the purposes of this example we assume that* RANK-DECOMPOSITION$([1, 0]) = 1$ *and* RANK-TRIPLET$(1, 1, 1; 1, 1, 2) = 1$.

Now, applying RANK *on the graph resulting from* LOCAL-RANK, *we determine that the rank of T is 6.*

6.1.3 Efficiency of ranking and unranking

Once the *count-array* and *summands* annotations of a graph are available, ranking and unranking of join trees is based on traversing arrays, for the

most part. Actually, the "bottleneck" of the process is the ranking and unranking of integer decompositions, since each decomposition may have as many as $O(n)$ elements. The relatively simple algorithms we use now take $O(n)$ time to rank and $O(n \log n)$ time to unrank.

Theorem 6.1 *Let G be a query graph on n relations. After a preprocessing step of $O(n^3)$ time, join trees of G can be ranked in $O(n^2)$ time and unranked in $O(n^2 \log n)$ time.*

Proof. By Theorem 5.1, the standard decomposition graph H of G and its *count-array* annotations can be computed in $O(n^3)$ time. To compute the *summands* annotation we use algorithm SUMMANDS, which takes also $O(n^3)$ time, because it requires $O(n^2)$ per node. This completes the preprocessing step of $O(n^3)$ time.
To rank a tree, the most expensive procedure is that of LOCAL-RANK. In the worst case, the time taken per node is $O(n)$ due to the ranking of integer decompositions. The total ranking time, then, is bounded by $O(n^2)$. To unrank a tree, the most expensive procedure is that of LOCAL-UNRANK. In the worst case, the time taken per node is $O(n \log n)$ due to the unranking of integer decompositions. The total unranking time, then, is bounded by $O(n^2 \log n)$ □

6.2 Generating random join trees

One of the basic operations used in probabilistic transformation based query optimizers is the random generation of a join tree. So far the efficient generation of such a join tree with a uniform distribution has been a hard open problem [SG88]. One of the reasons is that there is no one-on-one mapping between join trees and simple combinatorial structures —e. g. a permutation of graph edges— except for query graphs with a very "structured" topology such as star, string, or completely connected.

With an efficient unranking algorithm available, this problem can be solved in a straight forward way as will be shown in Section 6.2.2. First, the random generation algorithms used so far are discussed.

6.2.1 Random join trees

The following two procedures generate *quasi*-random join trees. They are easy to implement, but either do not guarantee uniform probability over the space, or else take a very long time.

Generate and test. A straight forward method is to generate binary trees at random, like proposed in [RH77] and permute the relations to the leaves. Since this method generates all *valid* and *invalid* join trees we need to check if the resulting join tree is either valid or invalid. If it is an invalid join tree it must be discarded, and an other join tree must be generated until a valid join tree is detected.

Number of relations	All join trees	Valid join trees	Fraction
5	40320	576	0.014
10	6.4×10^{15}	1.3×10^{11}	2.1×10^{-5}
15	3.0×10^{29}	7.6×10^{21}	2.5×10^{-8}
20	5.2×10^{44}	1.5×10^{34}	2.8×10^{-11}

Figure 6.10: Fraction of valid join trees

The efficiency of this method depends on both the fraction of *valid* join trees over the total number of *invalid* trees and on how fast the type (valid or invalid) can be detected. Using the definition of join trees in Section 2.1 an efficient recursive algorithm for labeling a tree valid or invalid can be implemented.

The fraction of join trees over the total number of trees decreases fast as the number of relations, n, participating in the query increases. Let us consider a star query on n relations. The number of join trees for this graph is $(n-1)!$ and on n leaves one can construct $\frac{(2n-2)!}{(n-1)!}$ labeled binary trees. Dividing these gives the fraction of join trees over binary trees: $\frac{(n-1)!^2}{(2n-2)!}$. In Table 6.10 the fraction of join trees is computed for several n. From the formula and the table we can see that we have to generate many binary trees before a join tree is generated. This method is therefore only feasible for very small queries.

Random walk. This procedure is based on transformation rules to move from one valid join tree to another. If we start at some join tree in the space and successfully apply transformation rules at random, we get a sample from the space being explored.

Random walks in graphs have been widely studied —see, for example, [GJ74, Ald89, Rag90]. In particular, if all nodes in a graph have equal degree, as is the case in the search space for acyclic queries [Kan91], then we are equally likely to be at any node of the graph after k steps, for a sufficiently large k, regardless of the starting point. In practice, however,

the length of the walk seems too large to be used to generate a single uniformly-distributed join tree. Instead, all join trees visited in a random walk are considered as a random sample of the space, resulting in a non-uniform distribution.

Random-edge selection. Instead of generating complete trees, as done by the previous two methods, random-edge selection builds join trees incrementally. The algorithm uses the relation between join trees and query graphs to generate valid join trees for acyclic query graphs.

The algorithm splits the query graph by selecting an edge at random. This edge maps one-on-one to a join operator and the inputs to the join operator are formed by the two sub-query graphs that result from the split. Since we deal with acyclic query graphs both resulting query graphs are also acyclic and connected. By recursively applying the random edge selection on each sub-graph until each graph consists of a single base relation a valid random join tree is generated.

1^{rst} choice	2nd choice	join tree	probability.
(b, c)	-	$(a \bowtie b) \bowtie (c \bowtie d)$	1/3
(a, b)	(c, d)	$(a \bowtie (d \bowtie (b \bowtie c)))$	1/6
	(b, c)	$(a \bowtie (b \bowtie (c \bowtie d)))$	1/6
(c, d)	(a, b)	$(d \bowtie (a \bowtie (b \bowtie c)))$	1/6
	(b, c)	$(d \bowtie (c \bowtie (a \bowtie b)))$	1/6

Figure 6.11: Generation of quasi random join trees by edge selection.

However, this efficient random generation scheme does not produce equi-probable join trees. Consider the query graph $\{(a, b), (b, c), (c, d)\}$. If we select (b, c) as the first edge to split the graph, then the join tree is already completely specified — assuming that left and right children are not distinguished. If, instead, the first edge selected is either (a, b) or (c, d) we must make a second choice. If choices are made uniformly from the available options, the table in Figure 6.11 shows the probability of generation of each join tree. In principle, it seems that the procedure can be modified to use *weighted* instead of uniform selection at each step, so that the resulting join trees are all equi-probable. But computation of the appropriate weights is difficult, and an efficient solution has not been found yet.

6.2.2 Random generation based on unranking

Uniformly distributed random generation of join trees follows from our results on counting and unranking. To generate random join trees for a given query graph G, first count the number of join trees in the space as described in Chapter 5; say there are N join trees. Now, simply generate a random number r between 1 and N, and unrank the join tree of r as described in Section 6.1, which can be done efficiently.

Theorem 6.2 *Let G be a query graph on n relations. Assuming a source of random bits, join trees for G can be generated at random with uniform distribution in time $O(n^2 \log n)$ per tree, after a preprocessing step of $O(n^3)$ time.*

Proof. To generate random join trees follow the procedure outlined above. Time bounds follow from Theorem 6.1. □

6.3 Improved random generation of join trees

The algorithm for generating join trees at random can be improved by moving the randomization deeper into the algorithm. Instead of generating a single random number and unrank the appropriate tree, we can bias the random generation of subtrees such that complete join trees are still generated with equal probability.

Given a standard decomposition graph with the count-array and summands annotations in place, the random generation starts at the root of this graph, computing *insert-at* x or *merge using* α annotations for each node. The annotations are computed as follows.

If the node in the standard decomposition graph is a leaf insertion $+_v$ with the count array $[x_0, x_1, \ldots, x_n]$, a random number is generated in the range $1, \ldots, \sum_{k=0}^{n} x_k$. This random number corresponds to some element of the count array, say x_i, and determines that leaf v will be at level i of the random tree that is being generated. The annotation *insert-at* i is generated. To be able to insert leaf v at level i, the child leaf w has to be at level $i-1, i, i+1, \ldots, n$. These levels form a sub-count array of the count array of leaf w and identify a set of trees from which one has to be chosen at random.

In case the node of the standard decomposition graph is a tree merging operator, $\times_v$, the level i at which leaf v should end up is determined in the same way as if the operator was a leaf insertion operator. For the two children in the standard decomposition graph, $\odot_{w_1}$ and $\odot_{w_2}$, this means

that if w_1 is at level j then w_2 has to be at level $i - j$ otherwise v can not be at level i. The levels for w_1 and w_2 are selected at random together with a merge specification α. The annotation *merge-using* α is generated. In Figure 6.12 and 6.13, the algorithms for generating join trees at random is given.

RANDOM-TREE(v)
let v be the root of the standard decomposition graph.
let $[z_0, \ldots, z_n]$ be the *count-array* of v.
let L_{low} be 0.
let L_{high} be n.
RANDOM-SUBTREE(v, L_{low}, L_{high}).
The resulting *insert-at* and *merge-using* annotations
define a random join tree.

Figure 6.12: Improved algorithm for generating join trees at random.

6.3.1 Efficiency of improved random generation

Once the preprocessing is done and the *summands* and *count-arrays* annotations are available, the random generation process only consists of traversing arrays. The most expensive operation while unranking a tree was unranking integer decomposition. In the improved random join tree generation algorithm this has been replaced by computing a random integer decomposition which can be done in $O(n)$ [NW78] and therefore speeds up the random generation of join trees.

Theorem 6.3 *Let G be a query graph on n relations. After a preprocessing phase of $O(n^3)$ time, join trees can be generated at random with a uniform distribution in $O(n^2)$ time.*

Proof. By Theorem 5.1, the standard decomposition graph H of G and its *count-array* annotations can be computed in $O(n^3)$ time. To compute the *summands* annotation we use algorithm SUMMANDS, which takes also $O(n^3)$ time, because it requires $O(n^2)$ per node. This completes the preprocessing step of $O(n^3)$ time.
To generate a random join tree the processing per node is $O(n)$ in the worst case. The total generation time, then, is bounded by $O(n^2)$. □

```
RANDOM-SUBTREE(v, L_low, L_high)
   let {w_1, ..., w_n} be the children of v;
   If n = 0
       there is no additional annotation on v;
   If n = 1
       let [z_0, ..., z_n] be the count-array of v
       select r at random such that 1 ≤ r ≤ Σ_{i=L_low}^{L_high} z_i;
       let k = min_j r ≤ Σ_{i=0}^{j} z_i;
       annotate v with insert-at k;
       RANDOM-SUBTREE(w_1, k − 1, n − 1);
   If n = 2
       let [z_0, ..., z_n] be the count-array of v
       let [σ_0, ..., σ_n] be the summands of v;
       let [x_0, ..., x_{n_1}] be the count-array of w_1;
       let [y_0, ..., y_{n_2}] be the count-array of w_2;
       select r_1 at random such that 1 ≤ r_1 ≤ Σ_{i=L_low}^{L_high} z_i;
       let k_1 = min_j r_1 ≤ Σ_{i=0}^{j} z_i;
       select r_2 at random such that 1 ≤ r_2 ≤ Σ_{i=0}^{n} σ_{k_1 i};
       let k_2 = min_j r_2 ≤ Σ_{i=0}^{j} σ_{k_1 i};
       let α be RANDOM-DECOMPOSITION(n_1, n_2);
       annotate v with merge using α;
       RANDOM-SUBTREE(w_1, k_1, k_1);
       RANDOM-SUBTREE(w_2, k_2, k_2);
```

Figure 6.13: Random subtree algorithm.

6.4 Summary

In this chapter we have described how the one-on-one mapping between integers and valid join trees can be done using the RANK and UNRANK procedures. These algorithm are built on the join tree counting theory and techniques developed in Chapter 5.

The UNRANK procedure makes it possible to construct and efficient algorithm which generates join trees at random with a uniform distribution. Also, an improved algorithm for generating join trees at random has been described that reduces the time complexity from $O(n^2 logn)$ to $O(n^2)$.

The integers required by our algorithms described in this chapter, as well as the ones in Chapter 5, can become quite large, as is the case with other

graph counting/generation problems [vL90],section 10.1.5. This eventually limits the applicability of our current results. Nevertheless, the algorithms can be used to a good extent on practical database queries. Using 64 bit integers, queries of about 20 relations can be processed, while 128 bit integers would extend it to queries of about 35 relations.

To extend the applicability of the current result, *reals* could be used instead of integers. This makes it possible to deal with bigger numbers at the cost of precision which is acceptable when generating join trees at random.

Part II

Experiments

Chapter 7

Transformation free optimization[1]

Query optimizers must find an "optimal" join tree from a space of many semantically equivalent alternatives. For join queries, the space of feasible evaluation orders grows very quickly, and to find an optimal join tree, known deterministic search algorithms take exponential time on the number of relations of the query [OL90]. This combinatorial explosion makes heuristics and probabilistic algorithms the prime vehicle for query optimization. *Simulated Annealing* (SA) and *Iterative Improvement* (II) are commonly used as reference points for research in this area [IW87, SG88, Swa89b, Swa89a, IK90, IK91, LVZ93].

The probabilistic search algorithms SA, II, and their variations rely heavily on transformation rules for generating candidate join trees. These transformations are based on properties of the underlying algebra, such as commutativity and associativity of the relational join. The performance of these algorithms depends, in addition to the cost distribution in the search space, on the set of transformations being used. In particular, a *complete* set of transformations —i. e. one that is sufficient to transform an initial join tree into any other join tree in the space— does not guarantee good behaviour, and it is sometimes necessary to add redundant transformations to improve the performance of algorithms [IK90].

Several sets of transformation rules have been studied, but the extent to which they allow rigorous analysis and prediction of the behaviour of transformation-based algorithms is somewhat limited —rather, they serve to provide qualitative insight [IK91]. A question that motivates the work in this chapter is the following:

[1]Parts of this chapter have been published in the *Proceedings of the International Conference on Very Large Databases, Santiago, 1994* [GLPK94]

if we are allowed to explore only a limited, fixed number of join trees, then what is more likely to produce good join trees, the application of transformations or a random selection from the complete space?

The distribution of cost in the search-space of join queries is the focus of [Swa91], which concludes that the proportion of good join trees decreases quickly as the number of relations in the query increases. But even if good join trees were only 1% of the space, random selection of 70 join trees will produce a good one with 50% probability —i. e. a coin toss. The results in [IK91] show that the proportion of good join trees also depends on the relative size of relations, and they give evidence that this proportion is significant.

Then, since there is evidence of a considerable number of good join trees in the space, and given the difficulties in analyzing how transformations lead to them, we study the behaviour of a *transformation-free optimization algorithm*. This algorithm generates a sequence of random join trees and applies a calibrated cost-evaluator to estimate their cost. Then the join tree with minimal cost becomes the preferred join tree of execution.

We start our experiments by exploring exhaustively the search space of small queries, to check the ratio of good join trees —i. e. those within a given factor times the optimal join tree. This CPU-intensive exercise shows the cost distribution over the search space. The results coincide with those of other, similar studies [Swa91].

In order to explore large search space we use the techniques and algorithms as described in Chapter 5 and 6 which make it possible to generate valid join-orders at random with uniform distribution in an efficient way.

Results of our experiments favour a transformation-free optimization algorithm in a direct comparison with the transformation-based SA and II, for the problem of join-order selection. The surprising observation is that our algorithm converges after exploring fewer join trees than the others, finding join trees of comparable cost.

This chapter is organized as follows. Section 7.1 presents basic definitions. Section 7.2 describes the experimental testbed and Section 7.3 shows the cost distribution over search spaces. The optimization algorithms are described in Section 7.4 and the performance measurement are described in Section 7.5. finally, in Section 7.6 presents the conclusions, a comparison with related work, and some directions for future research.

7.1 Definitions

Tree transformations. Our implementation of the traditional transformation-based algorithms uses the tree transformations of [IK90, IK91] for bushy trees, except for algorithm selection, because in our experiments we used one join algorithm within a single tree. The algorithms used are hash-join and nested loops. Only the transformations that lead to a valid join tree can actually be applied on a given tree. The transformation rules are:

Commutativity:	$A \bowtie B \leftrightarrow B \bowtie A$
Associativity:	$(A \bowtie B) \bowtie C \leftrightarrow A \bowtie (B \bowtie C)$
Left join exchange:	$(A \bowtie B) \bowtie C \leftrightarrow (A \bowtie C) \bowtie B$
Right join exchange:	$A \bowtie (B \bowtie C) \leftrightarrow B \bowtie (A \bowtie C)$

Kang showed in [Kan91] that for these transformation rules each join tree with n join operators has $2n - 1$ neighbors.

Search space. The set of all join trees from which the optimizer can choose is called the search space. In transformation based optimization algorithms the search space is mostly modelled by a graph $G = (V, E)$ with nodes V and edges E. The nodes represent join trees and the edges represent transformations between join trees.

A search space can be seen as a "landscape" if we associate the cost of a join tree to its "hight". Ioannidis and Kang [IK91] divided these landscapes into three categories; *cliff, bumpy* or *smooth* and analysed the performance of Simulated Annealing, Iterative Improvement and Two-Phase Optimization in these landscapes.

The size of the search space can be limited by imposing rules on the type or shape of the join trees. Such a smaller search space can speed-up the optimization process, but could at the same time result in finding less optimal join trees. A common way of reducing the search space is by considering only the valid join trees. This search space reduction is based on the assumption that join trees which incorporate a Cartesian Product are not likely to result in a low cost join tree [SAC+79]. However, in specific cases, an operator tree which allows Cartesian products can be of lower cost.

The search space can be reduced even further by only considering linear join trees, but studies presented by [IK91] suggest that the space of bushy join trees has a higher percentage of good join trees. Therefore, in our experiments, we consider the search space of valid bushy join trees.

7.2 Experimental setup

This section describes the conditions in which the experiments were performed. It describes the cost model, schema, catalogs, queries and our measure for *good* join trees.

7.2.1 Cost model

For all our experiments we used the cost function provided by the Analytical Performance Evaluator which provides accurate costs for query plan execution on the DBS3 prototype [ACV91]. The initial set of experiments were conducted using nested-loop joins. Since this is a conservative implementation of a join operator we changed it to hash-join. For the cost function of the hash-join we assumed that the hash-join algorithm sorts its inputs, such that it builds the hash tables on the smallest relation [Gra93].

The cost function used for the hash-join was adopted from [Kan91]:

$(|R| + |S|) * hash + |R| * move + |S| * comp * F$

With $|R|$ and $|S|$ denoting the sizes of the two inputs and $hash$, $move$, $comp$ and F are constants. It is assumed that $|R| < |S|$.

7.2.2 Database schema, queries and catalogs

The experiments foreseen require care in the design and population of the test databases. Traditional benchmark databases, such as Wisconsin and AS3AP, are primarily geared towards performance assessment of the algorithms in relation to the architecture. Moreover, their database statistics do not necessarily reflect real-world applications, which make them less suitable for assessing the quality of an optimizer. On the other hand, designing a new benchmark database complicates comparison with published results.

As in [LVZ93], we run our experiments against the Portfolio Club Experimental Model (PEM) [ACV91]. The database schema, the queries and the catalogs used in [LVZ93] constitute the starting point of our experiments.

Database schema The Portfolio Experimental Model was designed to provide a realistic experimental application base for complex query definition, evaluation, and benchmarking in the EDS project [Val92]. The PEM schema contains several relations join-able through foreign keys, see Figure 7.1 for the database schema.

mkt-sector (ms)	(**mkt-sect-id#** , mkt-sector)
port-holding (ph)	(**investr-id#, portfolio-id#, share-id#**, port-holding, low-limit,high-limit)
dividend (dv)	(**share-id#**, dividend, divd-date, divd-type, yield, p-e-ratio)
mkt-x-act (mx)	(**investr-id #**, portfolio-id, share-id, mx-type, mx-volume, mx-price, mx-log-date, mx-log-txn, mx-comm, mx-tax)
mkt-notify (mn)	(**share-id#** , mktn-date, mktn-code, mktn-note)
inv-prefs (ip)	(**investr-id#** , mk-sect-id, portfolio-id, cap-ratio)
investor (in)	(**inverstor-id#**, investor-cap, investr-name, investr-init, investr-add1, investr-add2, investr-add3, investr-city, investr-pmk, investr-ctry)
share (sh)	(**share-id#**, share-title, **mkt-sect-id#**, share-public, share-market, share-high, share-low, share-price, float-date)
mkt-movement (mv)	(**share-id#**, mkt-log-date, mkt-log-txn, mkt-buy, mkt-sell)
portfolio (pf)	(**portfolio-id#**, investr-id, portf-date)
nominalvalue (nv)	(**share-id#**, nom-note, nom-coin, nom-currency, nom-country)
investorzoom (iz)	(**investr-id#**, investr-name, investr-ctry)

Figure 7.1: Relations and their attributes

Queries Within the database schema described, several acyclic queries containing from 4 to 12 relations are considered. Figure 7.2 shows the query graphs for the queries used. Except for query 4, the topology of the query graphs is neither a star nor a string.

Catalogs The experiments were done for three different catalogs, the sizes are given in Figure 7.3. The catalogs have been chosen such that there is a catalog with low, middle and high variance in their relation sizes.

7.2.3 Cost metrics

The purpose of the first set of experiments (Section 7.3) is to investigate the distribution of cost over the search space, and especially the ratio of *good* join trees. Like Swami and Gupta [Swa89b, Swa91] we classify queries as *good, acceptable* and *bad* according the following criteria, in which join tree is abbreviated to JT.:

```
4:                                   5:
        ph- sh - nv- dv                       ph- sh - nv- dv
                                                   |
                                                   ms

6:                                   7:
   in - ph- sh - nv- dv                 ip - in - ph- sh - nv- dv
             |                                         |
             ms                                        ms

                                                 mv
8:                                   9:            \
mx-ip - in - ph- sh - nv- dv         mx-ip - in - ph- sh - nv- dv
                  |                                    |
                  ms                                   ms

             mv   mn                             mv   mn
10:            \ /                   11:           \ /
mx-ip - in - ph- sh - nv- dv         mx-ip - in - ph- sh - nv- dv
                  |                                |   |
                  ms                               pf  ms

             mv   mn
12:            \ /
mx-ip - in - ph- sh - nv- dv
         |    |   |
         iz   pf  ms
```

Figure 7.2: Queries used in the experiments

good:	$cost(p) \leq 2 * cost(cheapest\ JT)$
acceptable:	$2 * cost(cheapest\ JT) < cost(p) \leq 10 * (cheapest\ JT)$
bad:	$10 * cost(cheapest\ JT) < cost(JT)$

The number of join trees explored is used as a measure for optimization cost [LVZ93]. Although the number of join trees explored does not account for all the resources required by an algorithm, it provides an approximation of an *implementation-independent* measure for optimization cost.

Relation	cat1	cat2	cat3
1 ms	22	22	22
2 ph	5015	107407	25032
3 dv	728	744	6686
4 mx	5015	107407	505
5 mn	1000	1000	1000
6 ip	22249	69632	22249
7 in	505	5000	505
8 sh	1100	1100	10000
9 mv	1100	1100	10000
10 pf	1000	49965	5001
11 nv	1100	1100	10000
12 iz	500	500	500

Figure 7.3: sizes of the relations

7.2.4 Performance measure

The behaviour of an optimization strategy can be represented by a function mapping the number n of join trees explored, to the estimated cost of the best join tree found so-far. For a given algorithm A, we call this cost the ***solution*** after n, and denote it by S_n^A. Formally, using U_n^A as the set of the first n join trees visited by A, the solution after n is:

$$S_n^A = min\{cost(p) \mid p \in U_n^A\}.$$

Since the algorithms are probabilistic, U_n^A is a random subset of size n from the search space, and therefore S_n^A is a random variable. Based on this, we measure the success of these algorithms using the mean and standard deviation of the solution. As n increases, the mean of S_n^A should approach the minimum cost in the search space; while at the same time the standard deviation of S_n^A approaches zero. The second condition ensures that the algorithm, though probabilistic, behaves in a stable way.

7.3 Cost distribution in search spaces

Before the performance of the three optimization algorithms are compared we first perform several experiments to verify the observations of Ioannidis, Kang and Swami [IK91, Swa89a], that a considerable fraction of the

join trees in the search space is *good*, which is an important factor for the transformation free optimization strategy.

First the search spaces for queries from 4 to 8 relations are completely generated so the exact cost distribution can be determined. For queries of 8 to 12 relations an exhaustive exploration of the search space is not feasible, to approximate the cost distributions for these queries a sampling technique is used.

7.3.1 Small search spaces

To generate the complete search space for a given query we used the following algorithm. We consider connected queries with acyclic query graphs. Removing an edge from such a graph disconnects it, leaving two connected graphs. This makes it possible to enumerate the set of all valid join trees efficiently, by recursively splitting a query graph G as follows.

If the graph has one node, then the only join tree is the relation that labels such node; otherwise remove an edge, say labeled p, to disconnect the graph, then find join trees JT_1, JT_2 for the two connected graphs that remain, and finally return $(JT_1 \overset{p}{\bowtie} JT_2)$ as a join tree for G.

When at each recursion level all possible splittings of the graph are considered, all valid join trees are generated. Using the backtracking facility of Prolog, the algorithm takes only a few lines of code.

Experimental results Figures 7.4, 7.5 and 7.6 show the cost distribution of valid join trees for queries having 4 to 8 relations for the three different catalogs. One can observe that as the number of relations increases the number of good join trees decreases. This coincides with observations made by [IK91, Swa89a]. The queries of 6 and 7 relations have fewer good join trees than query 8 under catalog 1, so the shape of the query also has an impact on the cost. The cost distribution is further effected by the type of catalog, catalog 3 (with many big relations) has relatively few good join trees compared to catalog 1 and 2.

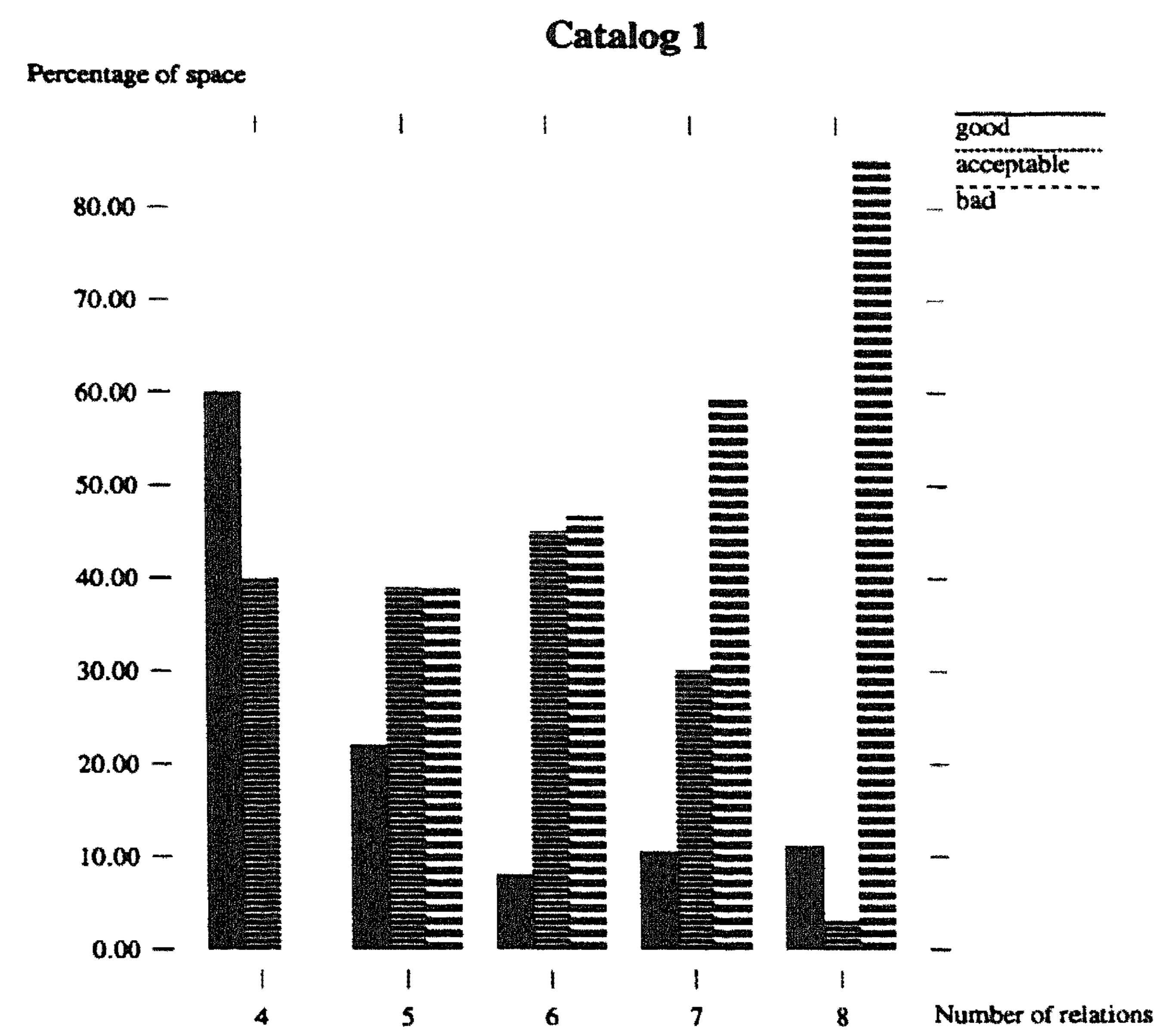


Figure 7.4: Histogram of cost distribution for catalog 1.

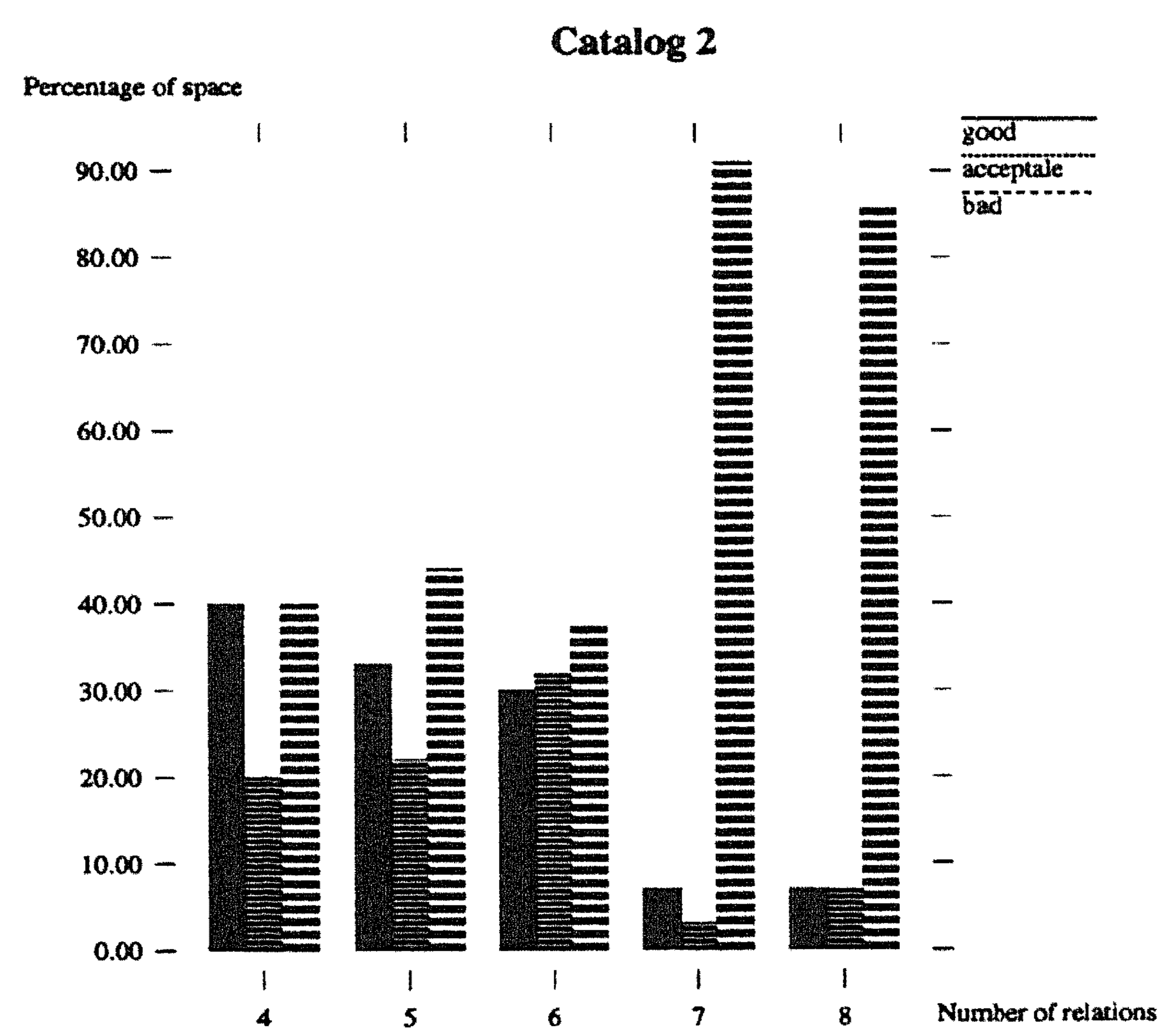


Figure 7.5: Histogram of cost distribution for catalog 2.

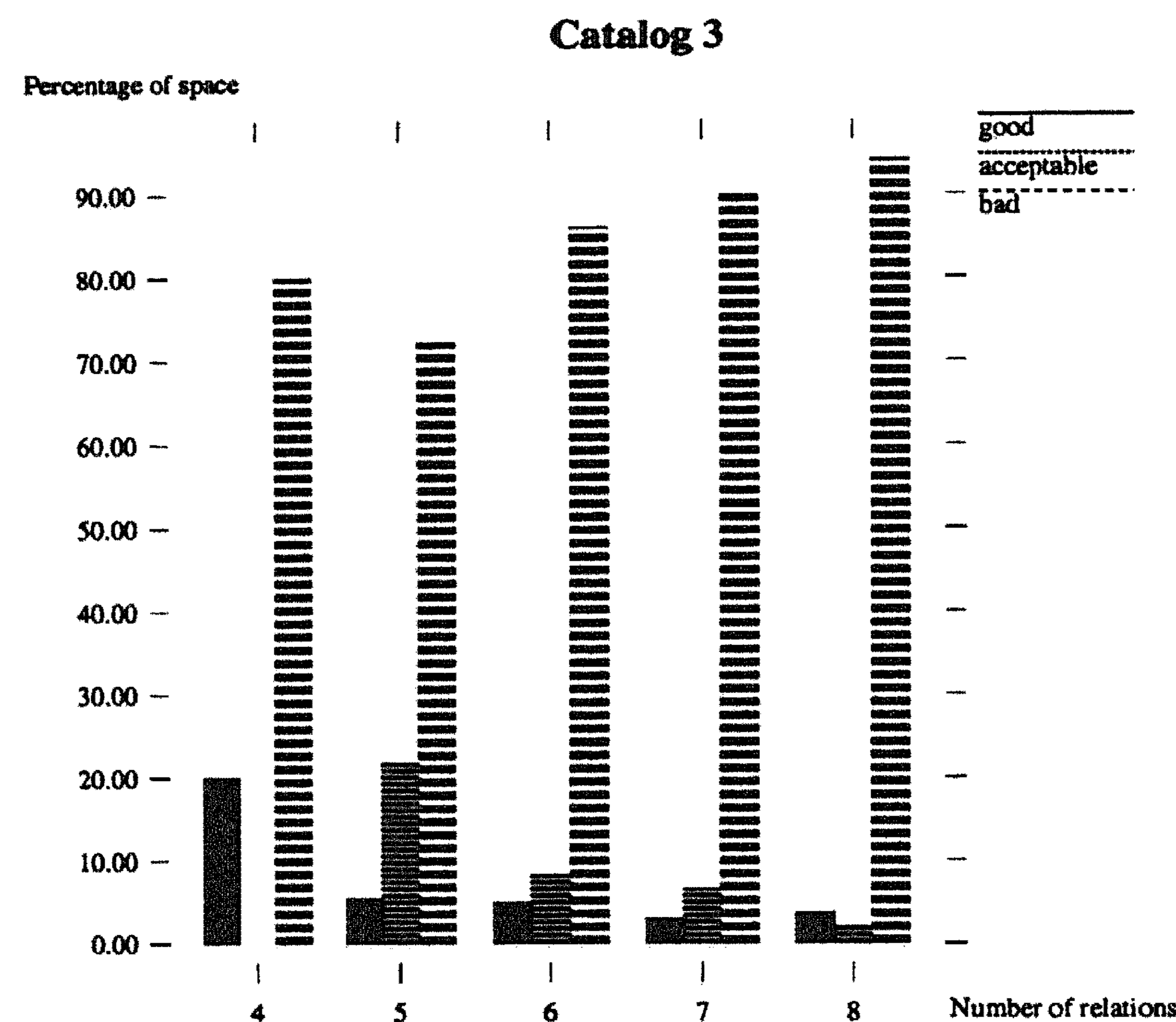


Figure 7.6: Histogram of cost distribution for catalog 3.

For all spaces explored exhaustively the number of good join trees is sufficiently large that a sample of several tens of randomly selected join trees will hit a good one with high probability. Given the fraction of good join trees and the required probability of hitting a good join tree the sample size can be computed as follows.

Lemma 7.1 *Given the ratio of good join trees, P_{good}, and the required probability of hitting a good join tree, P_{req}, the number of join trees n that must be explored is :*

$$n = \frac{log(1 - P_{req})}{log(1 - P_{good})}$$

Proof. Given that the ratio of good join trees is P_{good}, the chance of selecting a *wrong* join tree is $1-P_{good}$. The chance of selecting a sequence of n *wrong* join trees is therefore $(1 - P_{good})^n$. Since the probability of selecting a *good* join tree has to be P_{req}, the probability of selecting a *wrong* join tree must be $1 - P_{req}$. Given the two expressions that tell the probability of selecting a *wrong* join tree we can write the following equation : $1 - P_{req} = (1 - P_{good})^n$. Rewriting this equation leads to the expression: $n = \frac{log(1-P_{req})}{log(1-P_{good})}$. □

Figure 7.7 shows the probability of hitting a good join tree for several ratios of good join trees at increasing sample size. The ratios of good join trees are those for the queries with 4 to 7 relations in combination with catalog 1.

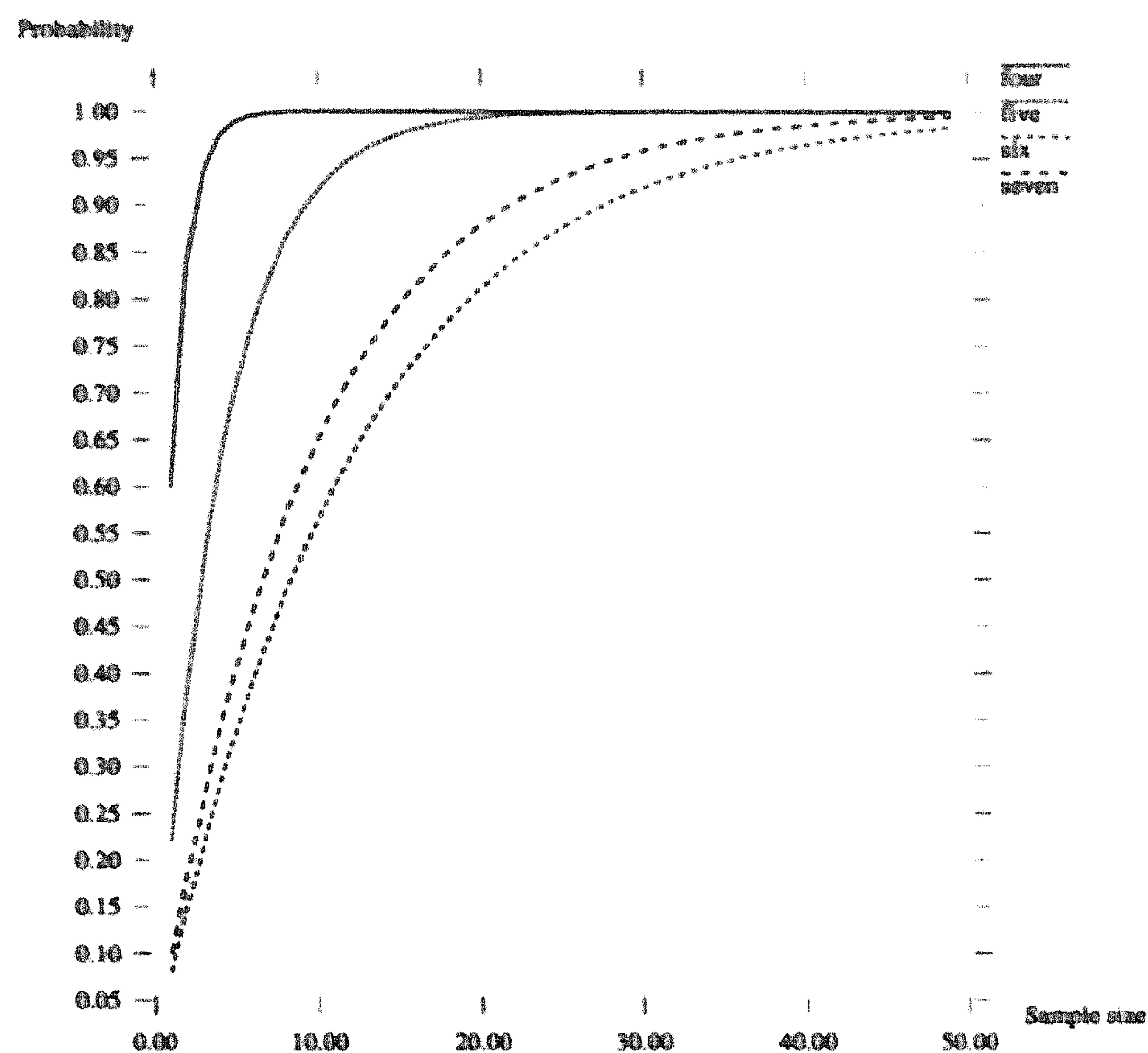


Figure 7.7: Probability of finding a good join tree in a random sample.

7.3.2 Random sampling

To determine the cost distributions of large search spaces a sampling techniques has to be used. To generate a random sample, several algorithms are available; however, only the ones that generate join trees with a uniform distribution are usable if the results should be representative for the complete space.

The three random sampling algorithms we considered are *random-walk*, *random-edge* and *uniform-random*. These algorithms are described in Section 6.2. Before we start sampling large search spaces we first verify the correctness of the sampling algorithms.

Quality of random sampling To evaluate experimentally the quality of a random sampling algorithm, i.e. whether or not the samples are fair, we compare the cost distributions of Section 7.3.1 to the cost distribution of the samples. Basically, the idea is that a fair sample from a space should preserve the cost distribution in that space. Note that we are not evaluating if the sample contains many good join trees, just how fair is the sampling procedure.

We use the accumulated cost distribution as the basis of our analysis. This distribution can be seen as a function on cost c, which gives the percentage of join trees having a cost less or equal to c. Formally, in a space

S, the *accumulated cost distribution* is

$$F_S(c) = \frac{|\{p \mid p \in S, cost(p) \leq c\}|}{|S|} * 100.$$

As samples become larger, they are expected to approximate the real cost distribution function of the space. For spaces of up to eight relations, we obtained the exact accumulated cost distribution in section 7.3.1, which allows us to compute the accuracy of the samples taken. Therefore, we compute the correlation coefficient of the function $F_S(c)$ with $F_{S'}(c)$, where S is the complete search space and S' is a random sample obtained by one of our methods. Figure 7.8 shows the correlation coefficients found for a query of eight relations, for increasing sample sizes.

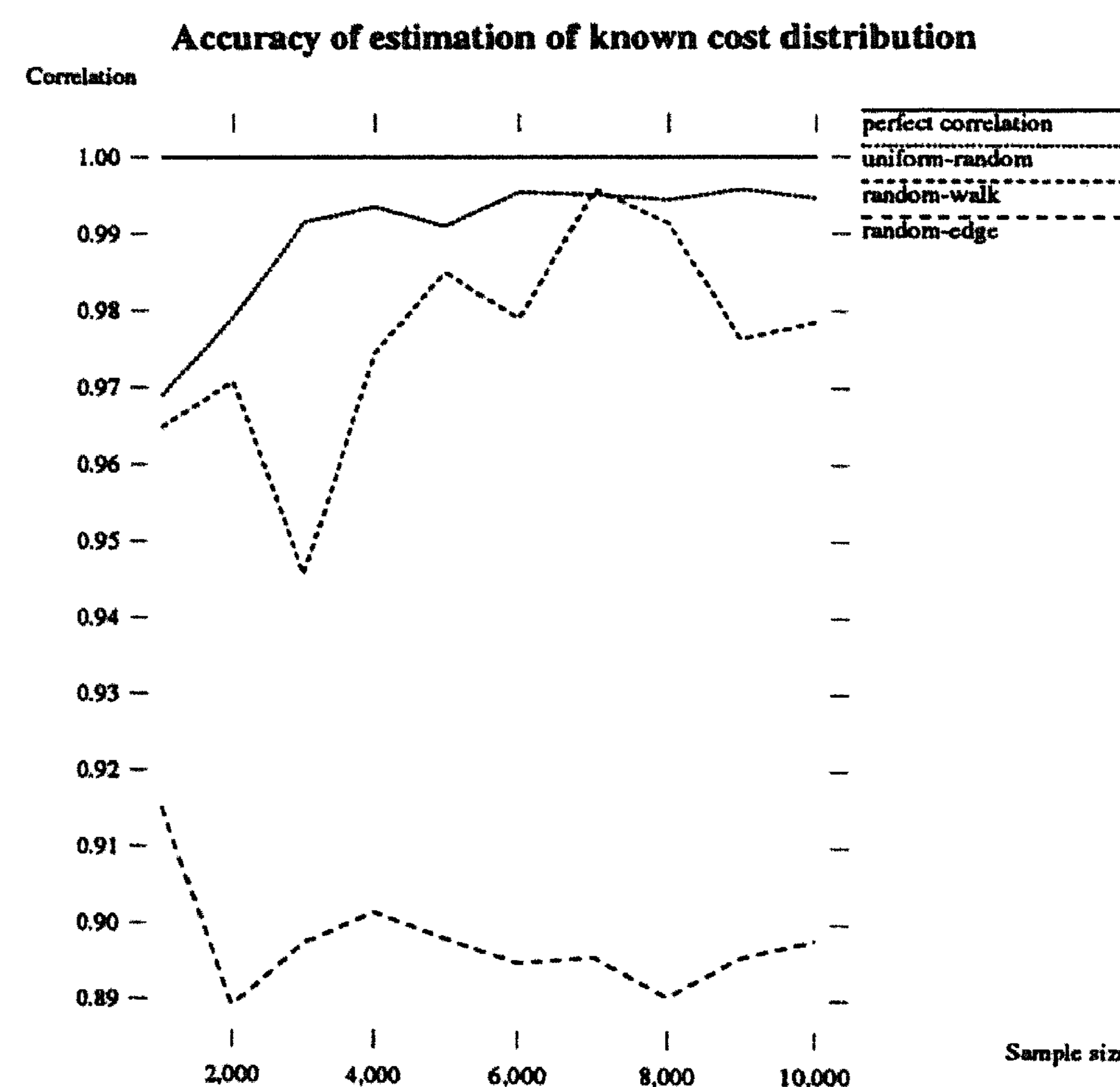


Figure 7.8: Correlation of approximations to cost distribution using random sampling.

The random-edge sampling method does not approach the exact cost distribution, because it favors certain join trees. Since we do not know the quality of the join trees that are favored, the effect of using this method in an optimization strategy is not clear.

The random-walk and uniformly-random method give a more accurate sample and improved their approximation as the sample size increased, but

the uniform-random method converges faster to the known cost distribution.

7.3.3 Large search spaces

The previous sections showed that the uniform-random method is the most appropriate method for sampling large search spaces. Using this method we sampled the search spaces for queries from 9 to 12 relations for catalogs 1, 2 and 3.

Experimental results For each search space a sample of 10.000 join trees was generated and the fraction of good, acceptable and bad join trees was determined. The results are shown in Figure 7.9, 7.10 and 7.11.

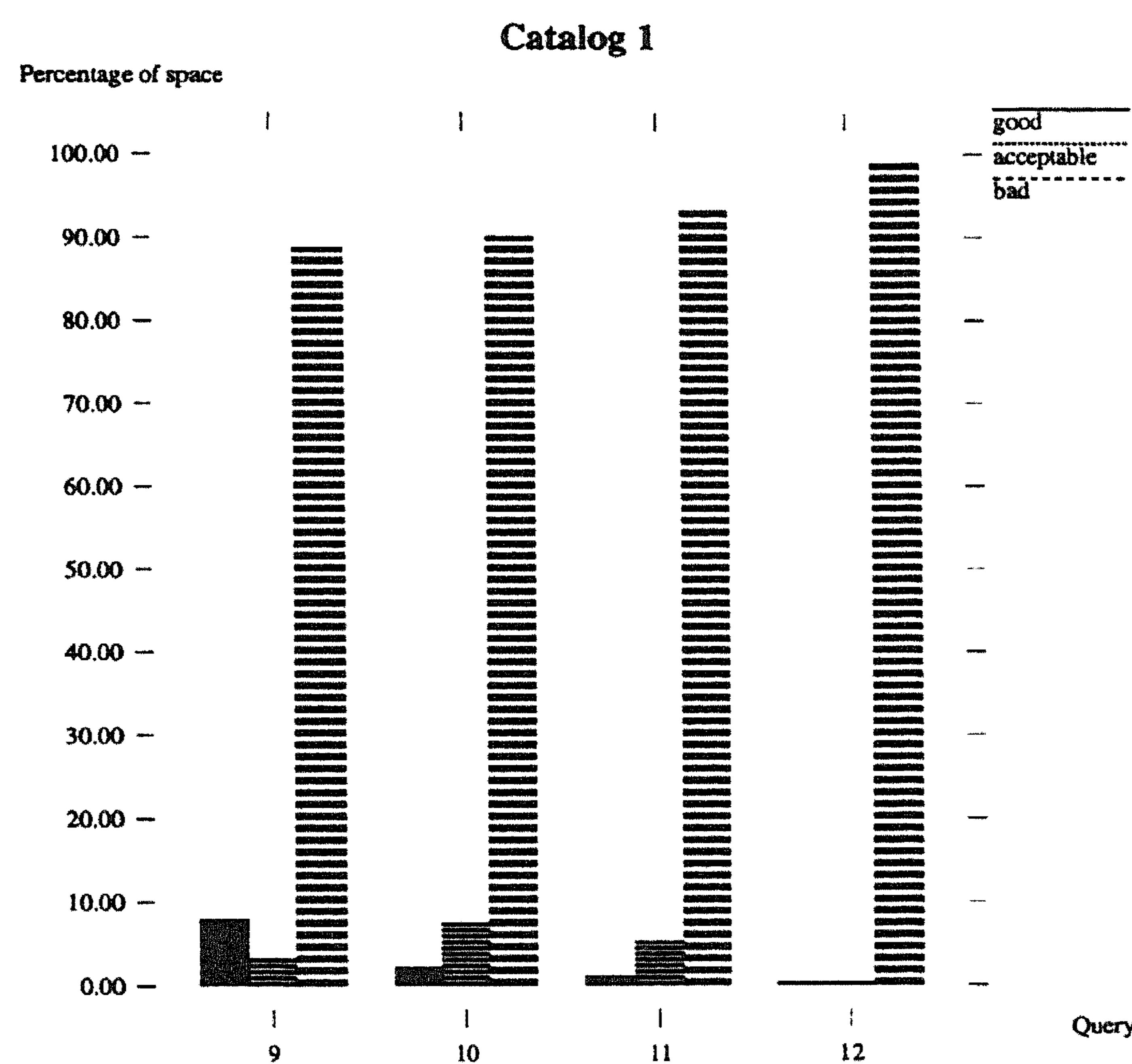


Figure 7.9: Histogram of approximate cost distribution for catalog 1.

These big spaces still show some good join trees but the percentage of good join trees decreased. This is in line with the observations of [Swa89a] for the spaces of linear join trees and with our results of Section 7.3.1. The decreasing number of good join trees implies an increase in the number of join trees required by a transformation-free optimization strategy. But also other optimization strategies need to explore more join trees as the search space increases.

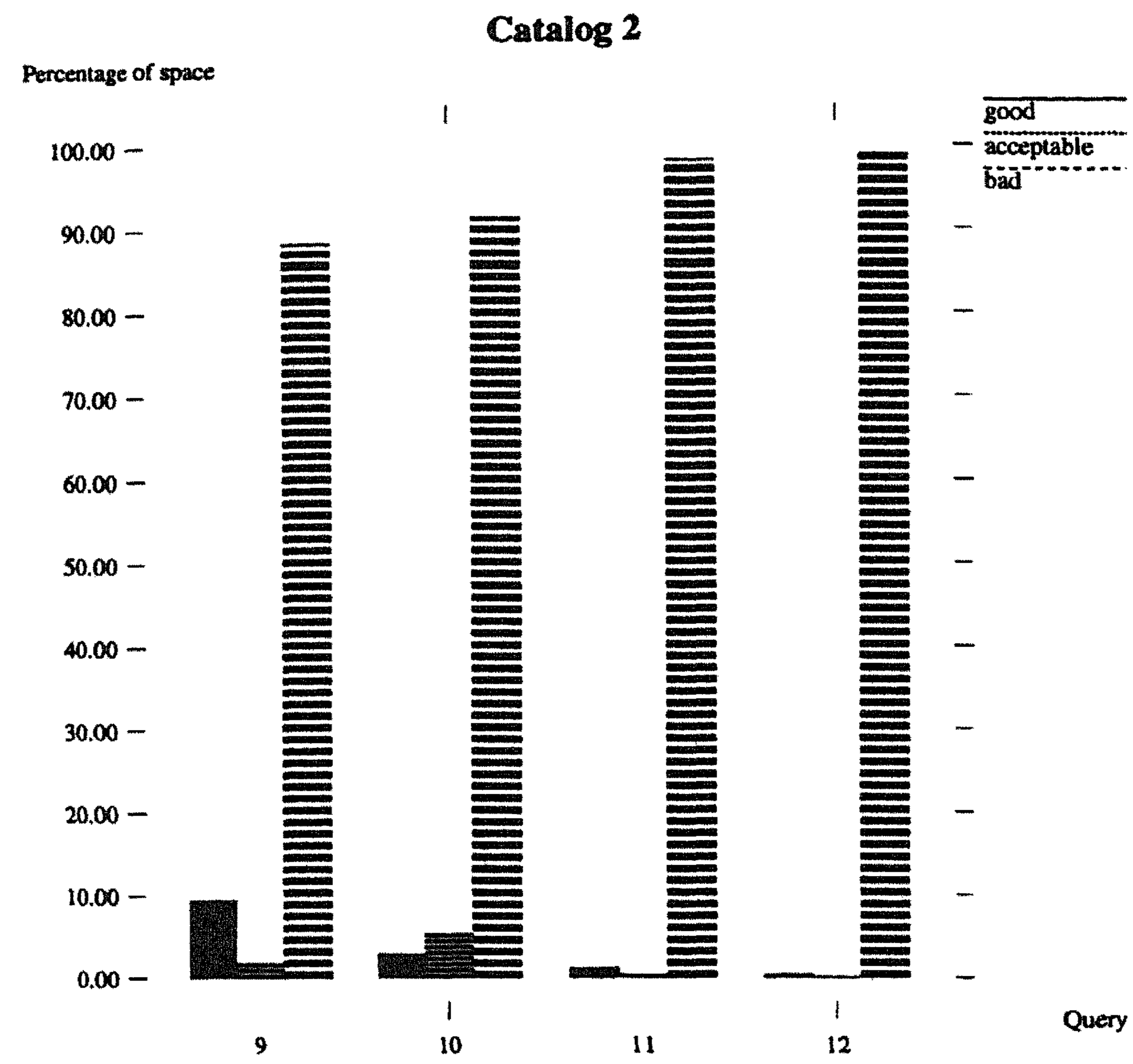


Figure 7.10: Histogram of approximate cost distribution for catalog 2.

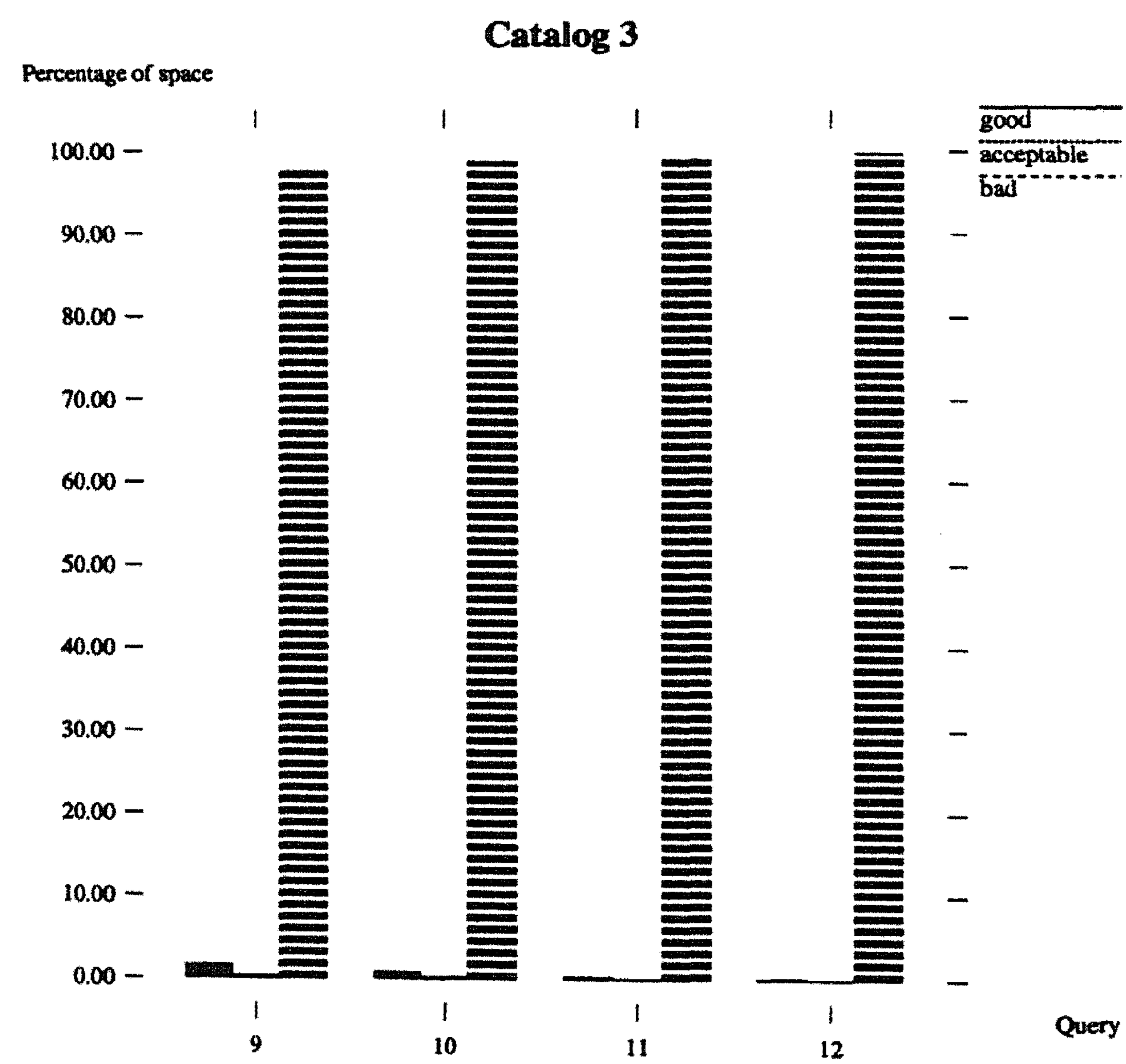


Figure 7.11: Histogram of approximate cost distribution for catalog 3.

7.4 Optimization algorithms

Since the fraction of the search space containing good join trees is sufficiently large, a random sampling algorithm is likely to be efficient. To verify this we compare the performance of the Transformation Free algorithm with the performance of two transformation-based optimization algorithms, *Simulated-annealing* and *Iterative-improvement*.

7.4.1 Iterative Improvement (II)

Iterative Improvement(II) performs a large number of *local optimizations*. A local local optimization starts at a random join tree, accepting only moves which improve the solution. When a local minimum has been reached, the process is repeated until a *stopping condition* is met. The output returned is the local minimum with the lowest cost [IK90, SG88].

Nahar et. al. [NSS86] show that as time approaches infinity, the probability that II visits the global minimum approaches 1. However, given a finite amount of time, the performance of II depends on the characteristics of the cost function over the search space and its connectivity. The stopping condition can both depend on the quality of the output and the effort invested. Figure 7.12 shows the pseudo-code of the II algorithm.

```
PROCEDURE II(){
     minS = infinite;
     WHILE not (stopping_condition) DO {
        S = random state;
        WHILE not (local_minima(S)) DO {
           S' = random state in neighbors(S);
           if cost(S') < cost(S) THEN S = S';}
        IF cost(S)<cost(minS) then minS = S;}
     return(minS);}
```

Figure 7.12: Iterative Improvement

In our experiments we compared the quality of algorithms after exploring a fixed number of join trees. For the II algorithm, this fixed number of join trees is used as the stopping condition. To obtain a random starting point for a local optimization, we used our algorithm that generates join trees at random in a uniform way, (See Chapter 6).

Instead of searching all neighbors of a join tree to detect whether or not it is a local minimum, we used the definition of an *r-local minimum*

[Kan91]. This method classifies a join tree as local minimum if none of r randomly selected (with repetition) neighbors has a lower cost, with r being equal to the number of neighbors of the join tree. Note that since the join trees are selected at random, and repetitions are therefore possible, an r-local minimum is not guaranteed to test all neighbors.

7.4.2 Simulated Annealing (SA)

Simulated Annealing (SA) starts at a random join tree and randomly generates moves to neighboring join trees. If the next join tree is an improvement, the move is accepted; if the move leads to a join tree with higher cost, it is accepted with a certain probability. As time progresses, this probability decreases until it is zero, which ends the optimization sequence. The output is the join tree with the lowest cost. It can be shown that the algorithm converges to the global minimum as temperature approaches zero [IK90, SG88].

Again, given finite amount of time to reduce the temperature, the performance of SA depends on the characteristics of the cost function over the search space and its connectivity, which make it sensitive to the starting point. Figure 7.13 shows the pseudo-code of the SA algorithm. For more detailed descriptions of the SA and II algorithms see [IW87, IK90, SG88, NSS86].

```
PROCEDURE SA(){
 S = S0;
 T = T0;
 minS = S;
 WHILE not(frozen) DO {
  WHILE not(equilibrium) DO {
   S' = random state in neighbors(S);
   deltaC = cost(C') - cost(S);
   IF (deltaC <=0) THEN S = S';
   IF (deltaC > 0) THEN S = S' with probability e^(-deltaC/T);
   IF cost(S)<cost(minS) THEN  minS = S;}
  T = reduce(T);}
 return(minS)}
```

Figure 7.13: Simulated Annealing

Like II, the Simulated Annealing algorithm starts at a random state. In our implementation we used our algorithm that generates join trees at

random in a uniform way, see Chapter 6. For SA-specific parameters, we used the parameters given by [Kan91].

parameter	value
initial temperature T_0	2* cost of initial join tree
frozen	$T < 1$ and cost unchanged for 4 stages
equilibrium	$16 * J$ visited states in current stage
temperature reduction	$T_{new} = T_{old} * 0.95$

As extra stopping condition, (the *frozen* condition), we added the number of join trees explored. The *equilibrium* condition means that the inner loop finishes if n join trees have been explored, with n = 16**Number of joins in the tree.*

7.4.3 Transformation free algorithm (TF)

Transformation Free (TF) generates join trees at random, with replacement, and keeps track of the one with the lowest cost. The algorithm terminates after it has visited n join trees or when the cost of the best join tree found so far is low enough. Like II and SA, if TF is given infinite time it will find the global minimum. Unlike SA and II, if time is finite TFs performance only depends on the cost distribution over the search space and not on its connectivity. Figure 7.14 shows the pseudo-code of the TF algorithm.

```
PROCEDURE TF(){
  minS = infinite;
  WHILE not(stop_condition) DO {
    S = random state;
    IF cost(S)<cost(minS) THEN minS = S}
  return(minS)}
```

Figure 7.14: Transformation Free

In the experiments the number of join trees explored is used as stopping condition.

7.5 Performance measurements

In the experiments, we measured the values of S_n^A for various queries and catalogs, for algorithms II, SA, and TF. In each run, we let each algorithm explore 5,000 join trees. The number of repetitions for each experiment

was 15, each leading to a different observation of the random variables S_n^A. At the end of the experiments, costs were scaled to the best found; for example, a cost of 2 corresponds to a join tree that is twice as expensive as the best found by any method.

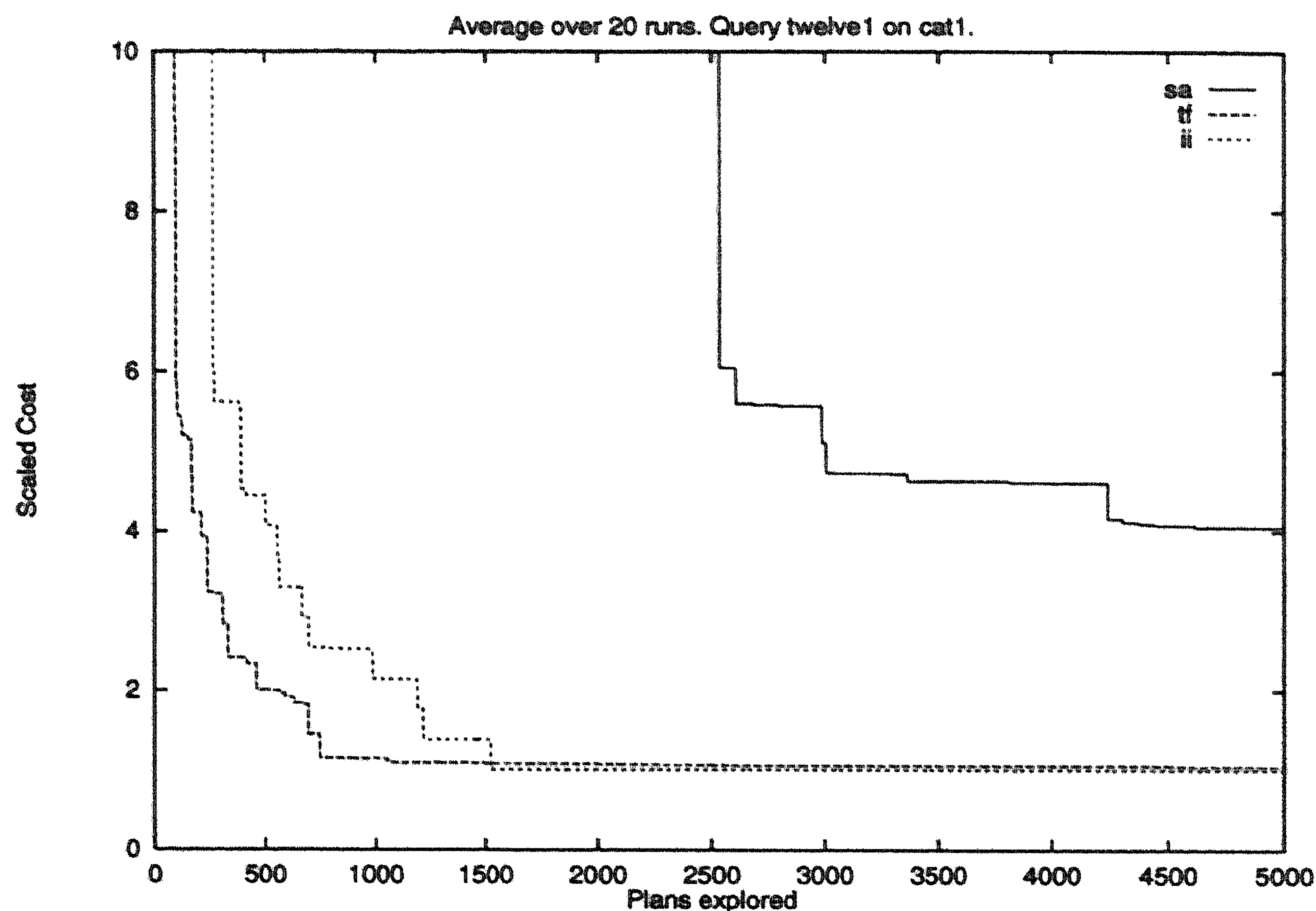


Figure 7.15: Average of cost of solution found.

To analyse the results, we computed the average and standard deviation of the solutions S_n^{II}, S_n^{SA} and S_n^{TF}, for $1 \leq n \leq 5,000$. The result of this analysis, for a query of 12 relations on catalog 1, is shown in figure 7.15 and 7.16. See appendix A for the remaining performance graphs of queries from 9 to 12 relations in combination with catalogs 1,2 and 3.

The average of the solutions found after 5,000 join trees by each algorithm were avg($S_{5,000}^{SA}$) = 4.049, avg($S_{5,000}^{II}$) = 1.000, and avg($S_{5,000}^{TF}$) = 1.048. The standard deviations were std($S_{5,000}^{SA}$) = 4.177, std($S_{5,000}^{II}$) = 0.000, and std($S_{5,000}^{TF}$) = 0.050.

On the average, SA is not able to find a good join tree within 5000 join trees; it finds these only after exploring a few thousand more join trees. On the average, TF finds good join trees faster than II —e. g. TF finds join trees with a scaled cost of 2 after exploring about 500 join trees while II needs about a 1000. When II and TF keep exploring more join trees II will find slightly better join trees than TF. The maximum difference occurs

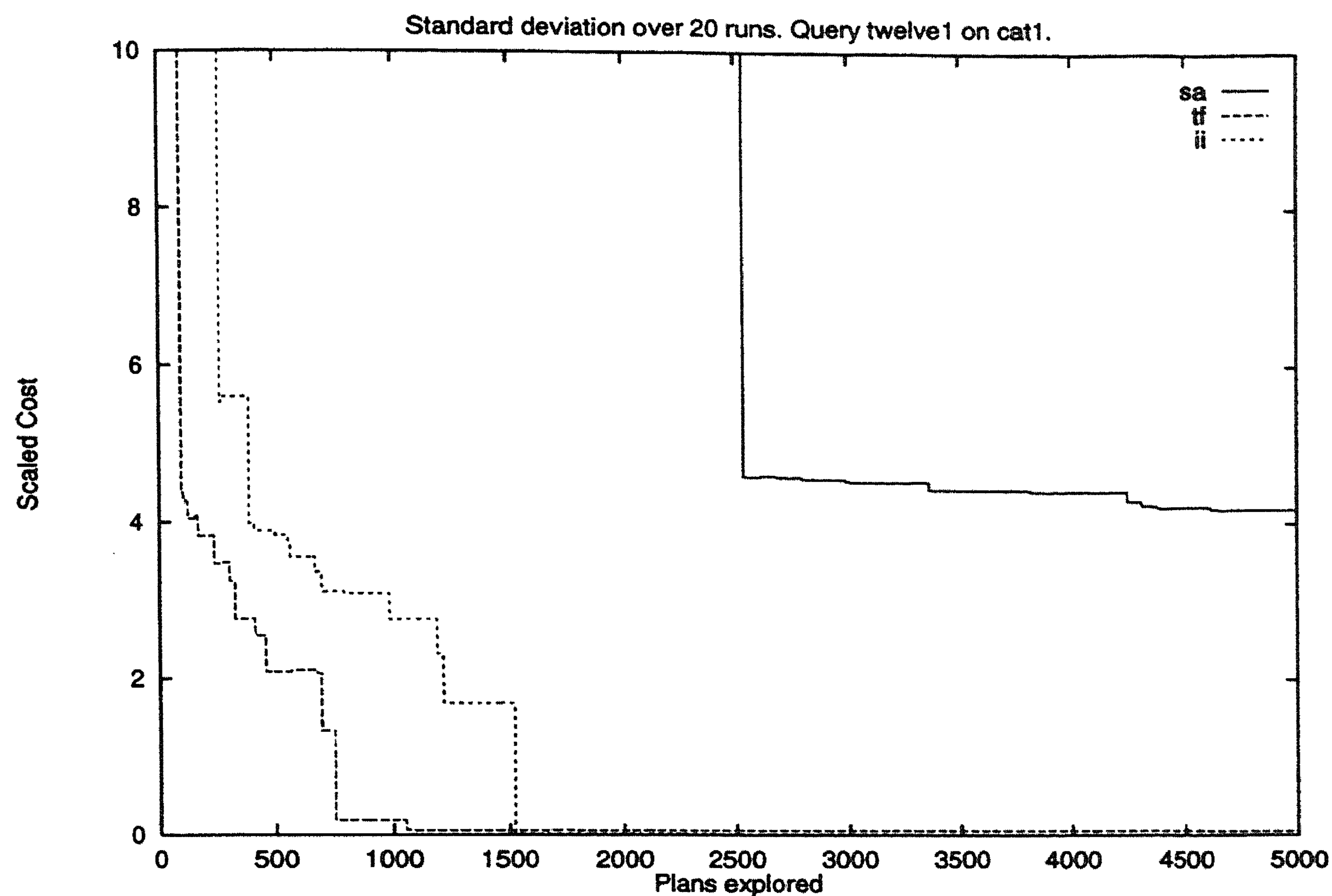


Figure 7.16: Standard deviation of cost of solution found.

after exploring 1800 join trees but is very small (1.01 v.s. 1.08).

Figure 7.16 shows that TF not only finds good join trees fast, on average, but that the quality of the join trees found in different runs are also close together. After about 750 join trees the standard deviation of TF is already 0.19 while that of II is only about 3.11. This leads to consider the ***time of convergence***, defined as the number of join trees required to reach a given maximum standard deviation. Setting the threshold to 0.1, SA doesn't reach the threshold within the first 5000 join trees. II converges after 1524 join trees, finding a solution of cost 1.01; and TF converges after 1056 join trees, finding a solution of cost 1.10.

7.6 Summary

In abstract terms, a set of transformation rules imposes a topology on the search space, but it is difficult to determine how a specific topology affects the performance of search algorithms. For the problem of join-order selection, the thorough studies of Ioannidis and Kang provide an empirical basis to understand the search space [IK90, IK91, Kan91]. Close analysis reveals that our results are not only consistent with their studies, but in

fact complement them. They observe that "...starting at a random state, many downhill moves are needed [by II] to reach a local minimum" [Kan91, p. 65]. In the spaces we considered, II had to explore well over a 100 join trees to reach a local minimum, on average —yet we point out that the expected length of a sequence of random join trees that finds one in the best 1% is 100.

Ioannidis and Kang conclude that SA finds very good solutions but takes a long time, compared with II. Their *two-phase optimization* algorithm uses II to find several local minima that are then used as starting points for SA. A similar multi-phased approach is taken in the *toured simulated annealing* of Lanzelotte, Valduriez, and Zait [LVZ93], where starting points of SA are obtained using a greedy deterministic algorithm. In this context, our result is that *transformation-based optimizers find very good solutions, but take a long time, compared with our transformation-free algorithm.*

A pure transformation-free algorithm has some specific advantages that should not be casually ignored. In our view, a key property of transformation-free optimization is that it has no "knobs to tune." For other algorithms, the setting of parameters for optimum performance is not obvious; results on the sensitivity of the algorithm to these settings are not available; and both optimum setting and sensitivity may depend on the specific cost model and database. Also, in parallel systems, transformation-free optimization can easily take advantage of available processors, achieving nearly optimal speedup —simply replicate the original algorithm in various processors, and add a final phase to determine the best solution found. Transformation-based "walks," on the other hand, are inherently sequential.

The prime novelty presented in this chapter is the transformation-free query optimization scheme, which provides a cheap and effective alternative to transformation-based algorithms. The mechanism relies on both an accurate estimation of query evaluation cost and an efficient mechanism to generate query join trees uniformly distributed over the search space. This leads to a strategy where a random sequence of valid join trees is generated and analysed on their perceived cost. The best join tree within the run is selected for execution.

Exhaustive exploration or sampling of the search space of a class of queries provides a precise measure of the run length required to hit a good join tree. Our results then provide a natural baseline against which the added value of applying transformations and heuristics can be quantified.

Related work. Transformation-based optimization is a general and powerful techniques with applications beyond join-order selection; see, for ex-

ample, [FMV94]. More related to our present work, research on randomized optimization of join queries has been performed by Swami and Gupta [SG88, Swa89b, Swa89a, SI92b], Ioannidis and Kang [IK90, IK91, Kan91], and Lanzelotte, Valduriez, and Zaït [LVZ93]. In contrast to our work, their approach is based mostly on tree transformations. In terms of search space, Swami and Gupta, and Ioannidis and Kang study very large queries (up to 100 relations); Swami and Gupta, and Lanzelotte, Valduriez, and Zaït allow cyclic query graphs; but Swami and Gupta only explore linear join trees. Finally, the cost model of Lanzelotte, Valduriez, and Zaït is that of a parallel database.

Chapter 8

Hybrid algorithms

In the previous chapter we examined a transformation free (TF) optimization scheme that generates join trees uniformly at random and keeps the best solution generated as prime candidate for execution. Our finding was that transformations tend to improve solutions "slowly", while the TF scheme converges faster and finds join trees comparable to those found by transformation based optimizers.

The experiments described in Chapter 7 were based on a calibrated cost model for the DBS3 system [ACV91] —a main memory database whose cost model accounts for CPU only— and considered join trees with hash-joins only. In this chapter we report on experiments to assess the stability of the phenomenon observed. We use the I/O-dominated cost model of the University of Wisconsin[1], which was used in their randomized optimization work [IK90, Kan91]. We examine the impact of changes on the statistical profiles of the catalogs, and the use of different join algorithms.

To study the effect of the search space topology on the behaviour of the transformation based optimization algorithms we considered a set of *arbitrary* transformation rules — i.e. rules that do not use properties of the join evaluation order. It showed that even the topology of the search space imposed by arbitrary transformations can help the search process.

For the problem of selecting a join-order, the size of the space is exponential in the number of relations (see Chapter 5 for the exact size). When, in addition, a join algorithm is selected the resulting search space is the product of two exponentially large spaces. The prime observation is that including the selection of join algorithms has a different effect on the prob-

[1]We are grateful to Yannis Ioannidis for kindly providing us with a copy of their software, and allowing us to modify it for our experiments

lem than changing the cost model or the catalog profiles. In fact, the current experiments show a qualitative difference in the relative performance of optimization algorithms when different join algorithms are allowed. The "high proportion" of good solutions in the space of evaluation orders is for the most part preserved on different catalogs and cost models, but it decreases in the product space of evaluation orders with method selection. At the same time, the transformations used in this product space seem particularly appropriate and lead to good solutions.

We then study a two-phase approach similar to those of [IK90, LVZ93], using TF in the first phase and then transformations. The behaviour of this algorithm combines the fast convergence of random picking with the high quality of solutions of transformation-based search, and it is superior to the other algorithms in all the spaces considered. From the behaviour of this hybrid algorithm, it appears that the neighborhood structure around a given join tree, from the point of view of the transformation-induced topology, depends mostly on the cost of such a join tree.

This chapter is organized as follows. In Section 8.1 the testbed for the experiments is described. Section 8.2 extend our previous work and compares TF directly with II and SA. Section 8.3 explores the use of arbitrary rules and Section 8.4 contains experimental results on the hybrid algorithm. A summary is given in Section 8.5.

8.1 Experimental setup

This section describes under which conditions the experiments were performed. It describes the cost model, the database schema, catalogs, queries, the factors considered and a characterization of the performance graphs.

8.1.1 Cost model

The cost model called CM2 in [Kan91] is the basis for our experiments. This cost model assumes a disk-based database system. Since the cost of evaluating a join tree is dominated by the I/O, the number of pages that are read or written during the evaluation of a join tree is used as cost metric. A large buffer is assumed in the cost model.

The CM2 cost model is able to handle three join algorithms, namely *nested-loop, merge-scan* and *hash-join*. The cost functions for the nested-loops algorithm are *page-level* nested-loops join and *index-scan* nested-loop. The cost of the cheapest alternative is returned as cost for a nested-loop join. The cost of the merge-scan join consist of sorting the inputs, if they

are not already sorted, and by merging the two input streams. The hash-join also has two alternatives of which the one with the cheapest cost is returned. These two alternatives are *simple hash-join* and *hybrid hash-join*. In the computation of the cost for the hash-join it is assumed that the hash table is build on the smallest input.

8.1.2 Database schema, queries and catalogs

The database schema, queries and catalogs used in [IK91] constitute the starting point of our experiments.

Database schema. The database schema consists of 110 relations which all have four attributes. A join predicate can be defined between any two attributes as long as they do not belong to the same relation.

Queries. The queries used in the experiments are generated at random and are acyclic. They range from 4 to 20 joins and all join predicates are equality joins. For each query size, a query is generated and stored for later use. See Appendix B for the query graph topologies.

Catalogs. The queries were optimized for three catalogs with different variance in attribute values and relation size. The catalogs used in [IK91] form our starting point and in the sequel of this chapter these catalogs will be referred to as the *original* catalogs.

The catalogs are generated at random from a profile that specifies an allowed range for relation sizes and uniqueness of attributes. Figure 8.1 gives the profiles for the three types of catalogs used. For example, a catalog of type 2 (or simply catalog 2) uses relation sizes ranging from 1,000 to 100,000 tuples and the uniqueness of the attribute values range from 90% to 100%. This percentage of unique values is used for the computation of the join selectivity in the cost estimation. The ranges are chosen such that the *variance* in catalog 1 is small, and it is increased in catalogs 2 and 3.

Catalog	Cardinality	Percentage of unique values in attribute
catalog 1	1000	[0.9,1.0]
catalog 2	[1000,100000]	[0.9,1.0]
catalog 3	[1000,100000]	[0.1,1.0]

Figure 8.1: Sizes and selectivities of the *original* catalogs

8.1.3 Transformations rules

The transformations used to generate new join trees are the same as the ones used in the experiments of Chapter 7, see 7.1. For join method selection the following transformation rule was added, $A \bowtie_{method_i} B \leftrightarrow A \bowtie_{method_j} B$. Since these transformation rules are based on the algebraic properties of the join-operator, they are *domain-dependent*. In Section 8.3 we also define a set of *domain-independent* transformation rules.

8.1.4 Factors Considered

The factors considered in our study are the following:

- Catalog variance (the difference in relation size and join selectivity).
- Relation sizes (original catalogs or enlarged catalogs).
- Join algorithms (nested-loop, hash-join, merge-scan).

The *enlarged* catalogs are constructed by multiplying the relation sizes in the original catalog (Figure 8.1) by a hundred. These enlarged catalogs were used to study the impact of the large I/O buffer in the cost model and possible non-linear behaviour of the cost functions.

In the experiments discussed in Section 8.2.1 and 8.2.2 there is only one join method available for a single join tree. So all join operators in a join tree are either nested-loop, merge-scan or hash-join. Section 8.2.3 and 8.4 describe experiments in which the join trees considered combine different join algorithms. Each experiment was repeated 20 times.

8.1.5 Performance Characteristics

The graphs shown present the average of solutions found by the various algorithms after exploring a given number of join trees. The y-axis is a linear measure of *scaled cost*, with a scaled cost of 1 for the cheapest individual join tree found by any algorithm, in the given search space.

These graphs have some properties useful for the comparison of search algorithms. A general description of the graph of TF and II is as follows, see Figure 8.2 for skeleton performance graph. Up to a *crossover point* the TF algorithm generates better join trees, and after that the II algorithm finds better join trees. This crossover point marks the solution that is found by both algorithms after exploring the same number of join trees. Note that even if two join trees have the same (estimated) cost, their topology could very well be quite different.

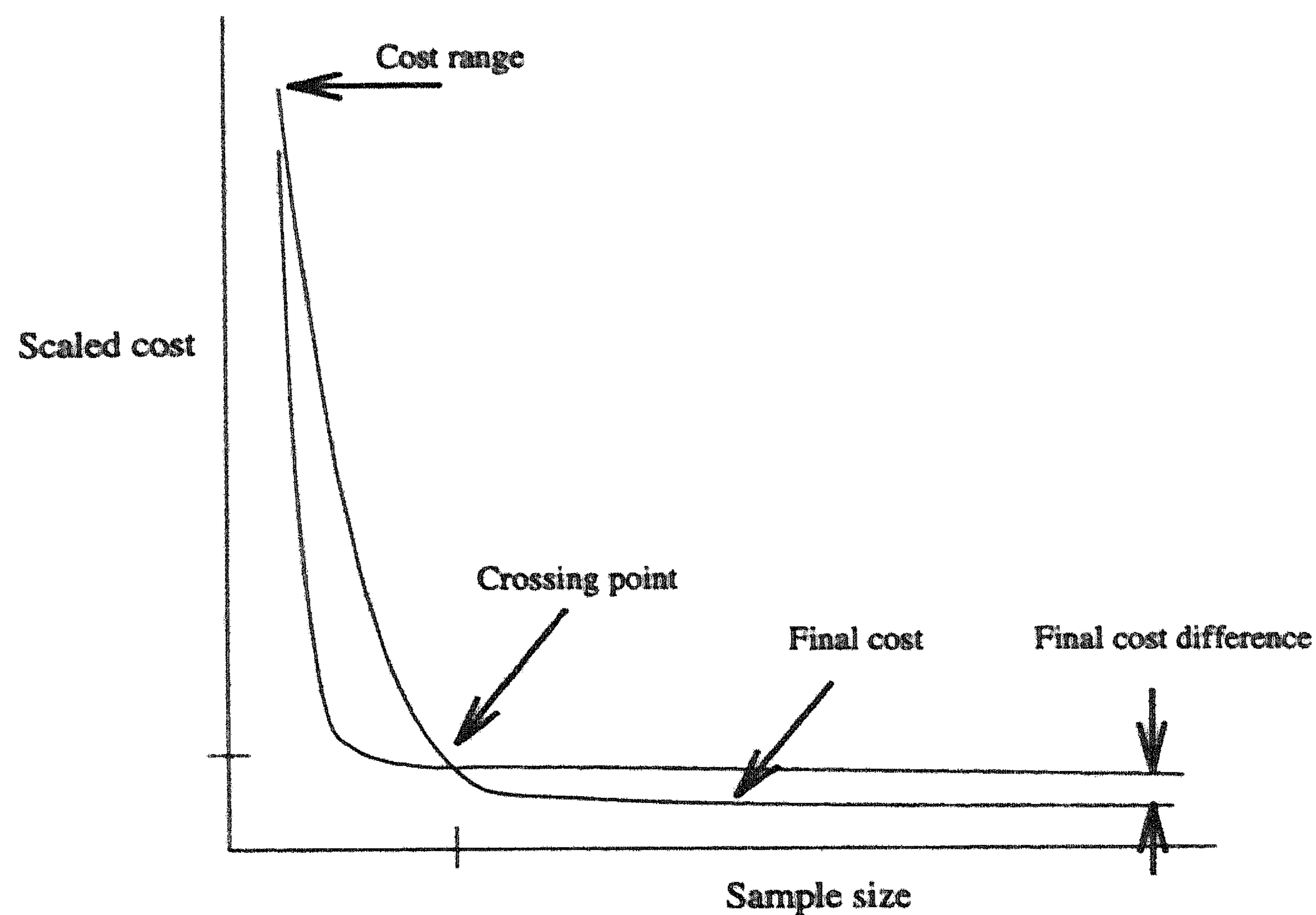


Figure 8.2: Skeleton performance graph

After exploring many join trees, the cost of solutions found by probabilistic algorithms improves very slowly. We could say that at some point the optimizer becomes ***stable*** and call the quality of the join tree at that point the ***final cost***. The difference in final cost is used to compare algorithms.

Another important characteristic of the graph is the *cost range*. If the difference between the best solution and the worst solution in the search space is small, the optimization has a relatively smaller impact on the execution time of the query. If, on the other hand, the cost range is large, the optimizer can produce a dramatic improvement on query performance.

These three aspects —*crossover point, final cost (difference)* and *cost range*— of a performance graph are helpful in analyzing the performance of the search strategies.

8.2 Results

This section discusses the experiments done to verify the performance of the TF algorithm. In particular, we present the behaviour of TF, II and SA for various catalogs and join methods. All graphs shown are a representative sample of a much larger set explored.

8.2.1 Original Catalogs

The original catalogs are used for our first experiment. As mentioned in Section 8.1 the optimizers only consider join trees in which all join algorithms are either nested-loop, merge-scan or hash-join.

We observed that as the catalogs changed, from low variance to high variance, the *cost range* of the graphs increased and the *crossover point* shifts to the right with the variance. The *final cost* of the II and TF algorithm are similar for catalogs 2 and 3. Only for the low-variance catalog 1 the II algorithm is consistently better.

For the high-variance catalog the II algorithm needs to explore many more join trees to find a join tree that outperforms the join trees found by the Transformation Free algorithm. Figure 8.3 is prototypical for the results obtained. It shows the results for a query of 20 joins when only hash-joins are considered (the results for nested-loops and merge-join are similar).

The behaviour of the TF algorithm for the three different catalogs is explained by the increasing ratio of good join trees, but the behaviour of the transformation based algorithms is harder to grasp. For catalog 1 the imposed topology helps the II algorithm finding good join trees, although the ratio of good join trees is small. This topology has changed for the worse in catalog 2 and also for catalog 3 the II algorithm converges slowly.

8.2.2 Enlarged Catalogs

To examine the impact of the large buffer on the performance of the search algorithms, we enlarged the relation sizes of the original catalogs. For these big relations, the join trees with only hash-joins were consistently cheaper than join trees with only merge scan or nested loop. This search space of join trees, with only hash joins, also showed the biggest change in performance as the catalogs change. Since all relations of catalog 1 have the same size, the performance graphs of the original and enlarged catalogs are very similar. For catalog 2 the TF algorithm finds join trees much faster than II and also the distance between the graphs has grown in comparison to the original catalog 2. For catalog 3 the TF algorithm improves faster before the *crossover point*, but this *crossover point* has a high cost.

With the enlarged catalogs 2 and 3 both the cost range and the difference between final costs has grown. In Figure 8.4 the performance graphs of the search algorithms are given for the tree catalogs when only hash-joins are used. To make the performance graphs of the search algorithms visible, the scale of the y-axis have been enlarged.

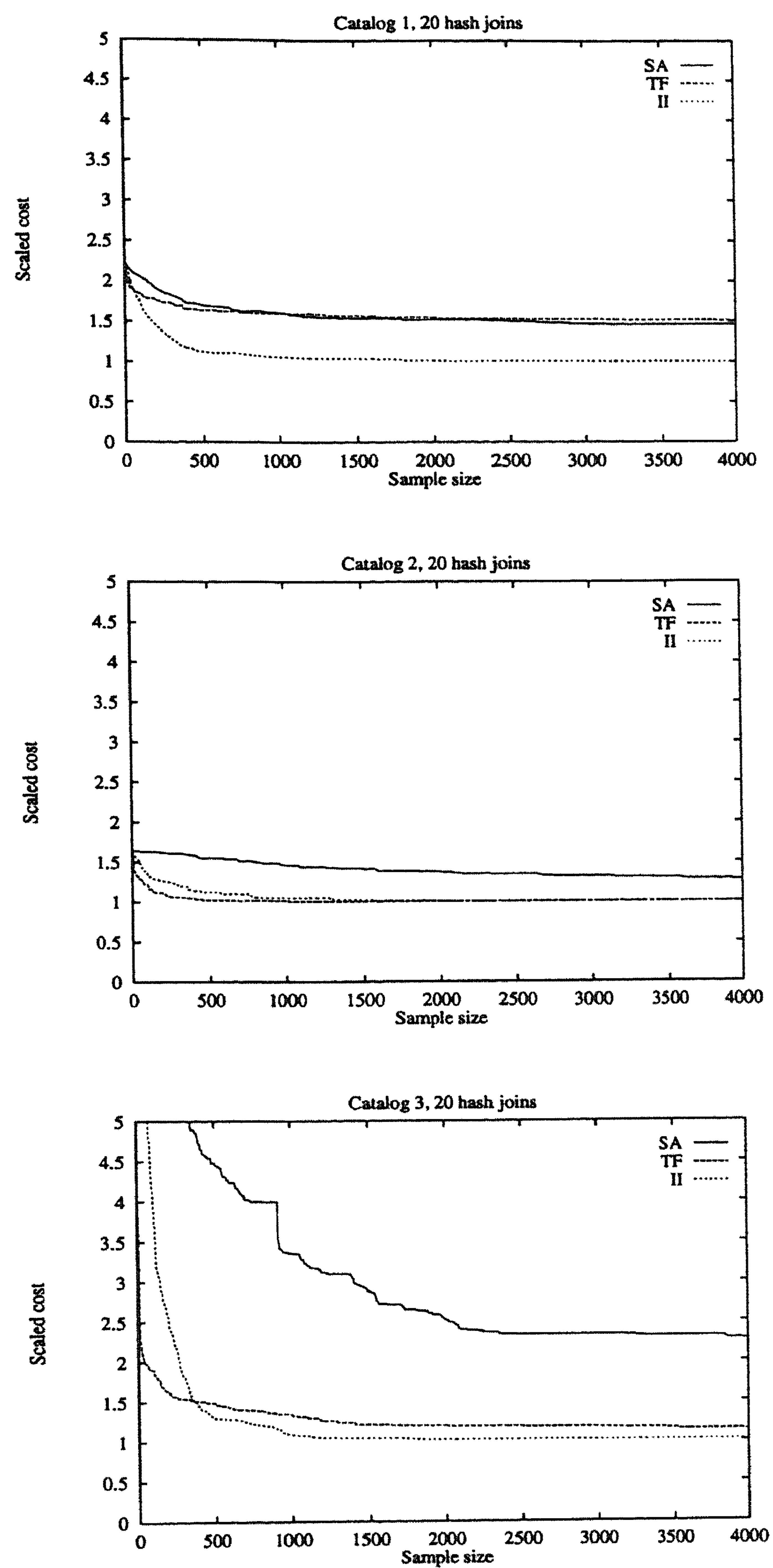


Figure 8.3: Space of hash join trees for the original catalogs

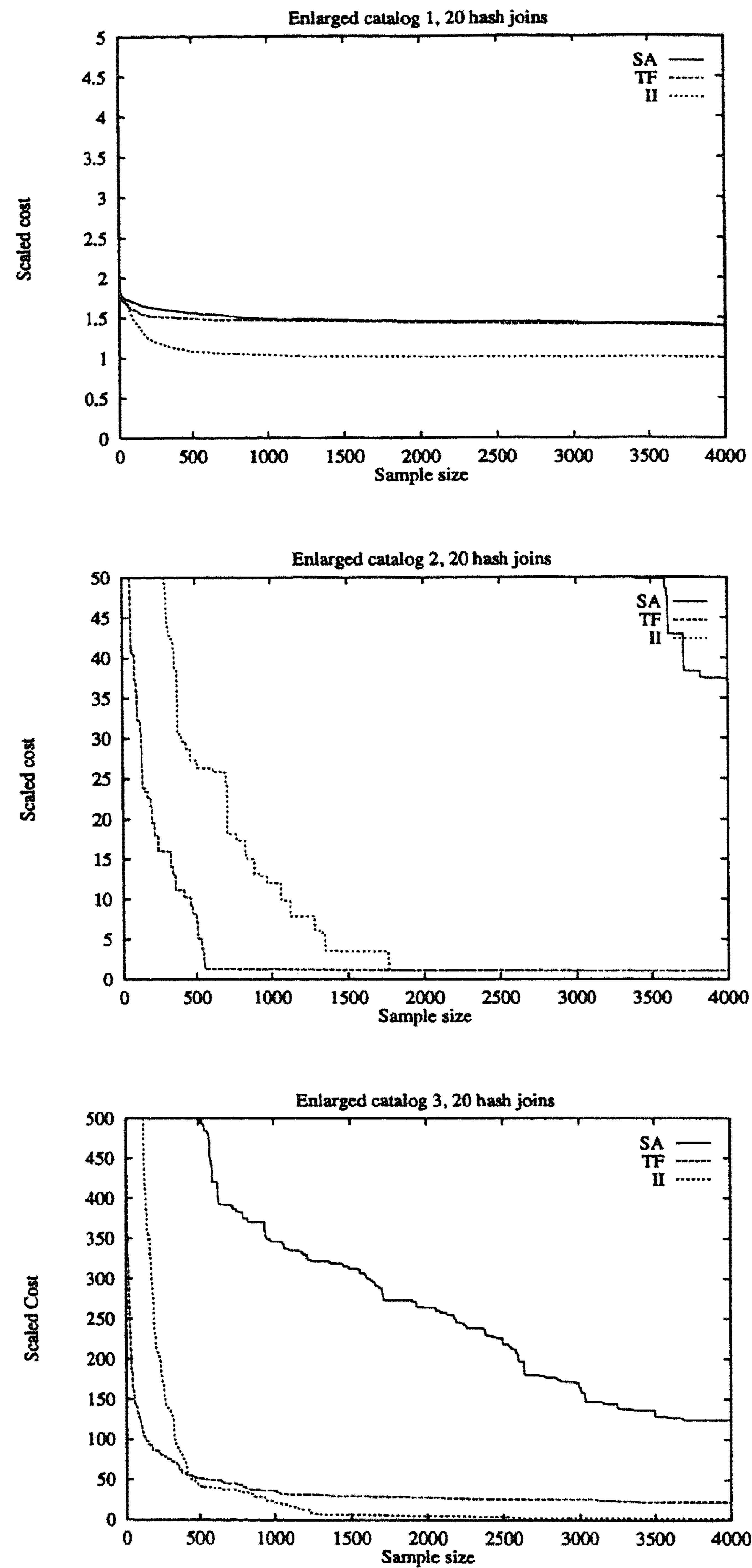


Figure 8.4: Space of hash join trees for the enlarged catalogs

8.2.3 Multiple Join Algorithms

We now consider the use of multiple join algorithms in join trees. To deal with this case in transformation based strategies, a rule is added that changes the algorithm at a specific join operator. Such addition leads to a dramatic growth of the search space. If m join algorithms are considered and the join trees joins n relations, each join tree in the original search space is mapped to m^{n-1} join trees with join selection. This big search space seems to contain cheaper join trees —e. g. a hash-join whose inputs are sorted can be replaced by a merge-scan— but it also introduces many join trees with higher cost. Important for the performance of all three search algorithms is how the cost distribution changes, and for transformation based optimizers also the modified connectivity of the search space.

Uniformly random generation of elements from the product space is easy. Simply generate a join tree at random and select independently and uniformly a join algorithm for each join in the join tree.

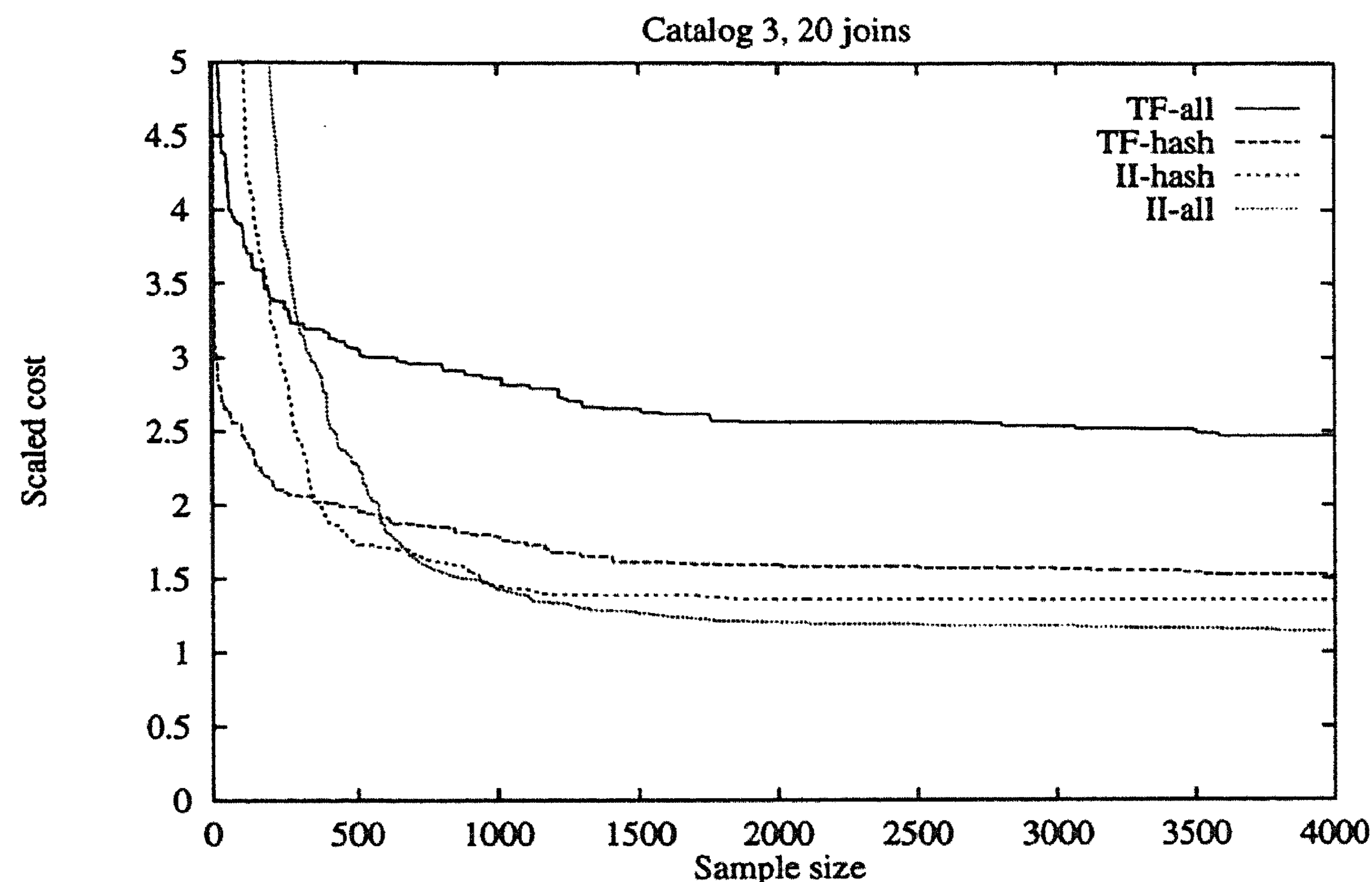


Figure 8.5: Multiple join methods

Figure 8.5 shows the performance graphs for II and TF using all join algorithms. As a reference, also the result of II and TF on the restricted space of join trees that use hash-joins only are shown. The effect of the product space is clear from this graph. Initially, both TF and II progress

about as quickly in the space restricted to hash-joins as in the more general space, but then TF becomes stable in more costly solutions.

We can conclude that the reduced percentage of good join trees in the bigger space has a negative effect on the performance of the TF algorithm. However, the topology imposed by the "change-join-algorithm" transformation rule seems particularly appropriate for a transformation-based search.

In Section 8.4, we show experiments in which random generation and the use of transformation rules are mixed. Ideally these methods should incorporate the good behaviour of both the TF and II algorithm, fast convergence and good final join trees.

8.3 Arbitrary set of Transformation rules

The set of transformation rules used by II and SA are based on algebraic properties of the join evaluation order, like *commutativity* and *associativity*, see Section 7.1. The search spaces that II and SA search can be viewed as graphs in which the nodes represent the join trees and the transformations are represented by the edges. These search graphs are regular graphs — each node has the same number of neighbors. A partial characterization of these regular graphs is given by the number of neighbors of each node and the *inter node* distance — the maximum length of a path that connects any two nodes.

In [Kan91] it is shown that for the set of transformation rules we use for II and SA, the number of neighbors is $2J - 1$ — commutativity provides J transformations and associativity $J - 1$. The inter node distance was proven to be no more than $\frac{J(J+1)}{2}$, with J the number of joins of the query.

To study the effect of transformation rules on the performance of the II algorithm we, used rules that impose an *arbitrary* topology on the search space. This new set of rules is chosen such that the imposed topology is similar to that of the original search space — same number of neighbors and the same inter node distance.

Our transformation rules are not based on the algebraic properties of the join-operator, but use the *rank* of a tree to compute its neighbors. Ranking is an arbitrary mapping of a set of n elements (trees) to the integers 1 through n, and is not related to the estimated cost of the elements. An unranking function determines the element T that corresponds to a given rank r, see Chapter 6.

The transformation function used computes rank k for the i-th neighbor of a tree with rank r. To select a random neighbor a number between 1 and the total number of neighbors is generated at random. This allows

the transformation function to compute the rank of the neighbor and the unrank function determines the corresponding tree. The transformation function we used is:

> $f(i,r) = (r + m^i)\ MOD\ N$, with base $m = \sqrt[d]{N}$, d the total number of neighbors and N the size of the search space.

The unranking function generates unordered trees, so to obtain a search space that has the same partial characterization as the original search space we use the transformation function as follows. The total number of neighbors is set to $2J - 1$, of which $J - 1$ are generated by the transformation function. The other J neighbors are obtained by commuting the j-th join operator.

An upper bound for the size of the search space is given by $J!$, in which case $m = \sqrt[J-1]{J!} = O(J)$. It can be shown that the inter node distance of this new search space is $J * (m - 1) = O(J^2)$, as in the original search space that is based on the commutativity and associativity properties of the join operator.

8.3.1 Experimental results

The experiments of Section 8.2.1, 8.2.2 and 8.2.3 show that transformations are effective towards the end of the optimization process. An experiment has been setup to determine if this behaviour is specific for the chosen set of transformation rules, or if it is a more general property of transformation based search.

For the II algorithm an alternative set of transformations was used. This alternative set of rules was chosen arbitrarily as described in the previous section. The algorithm explored up to 4000 join trees and each experiment has been repeated 20 times. A representative performance graph is given in Figure 8.6. It shows the graphs of TF, II with original transformations and II with the arbitrary transformations (ARB) on a high variance catalog.

The search space with an arbitrary topology imposed upon it still aids the II algorithm in finding good join trees. Towards the end of the optimization these arbitrary rules perform better than the TF algorithm, but the original set of rules still finds the best join trees.

8.4 Hybrid search algorithms

Considering all experiments performed, an improvement of transformation based optimizers seems feasible by balancing the generation of random join

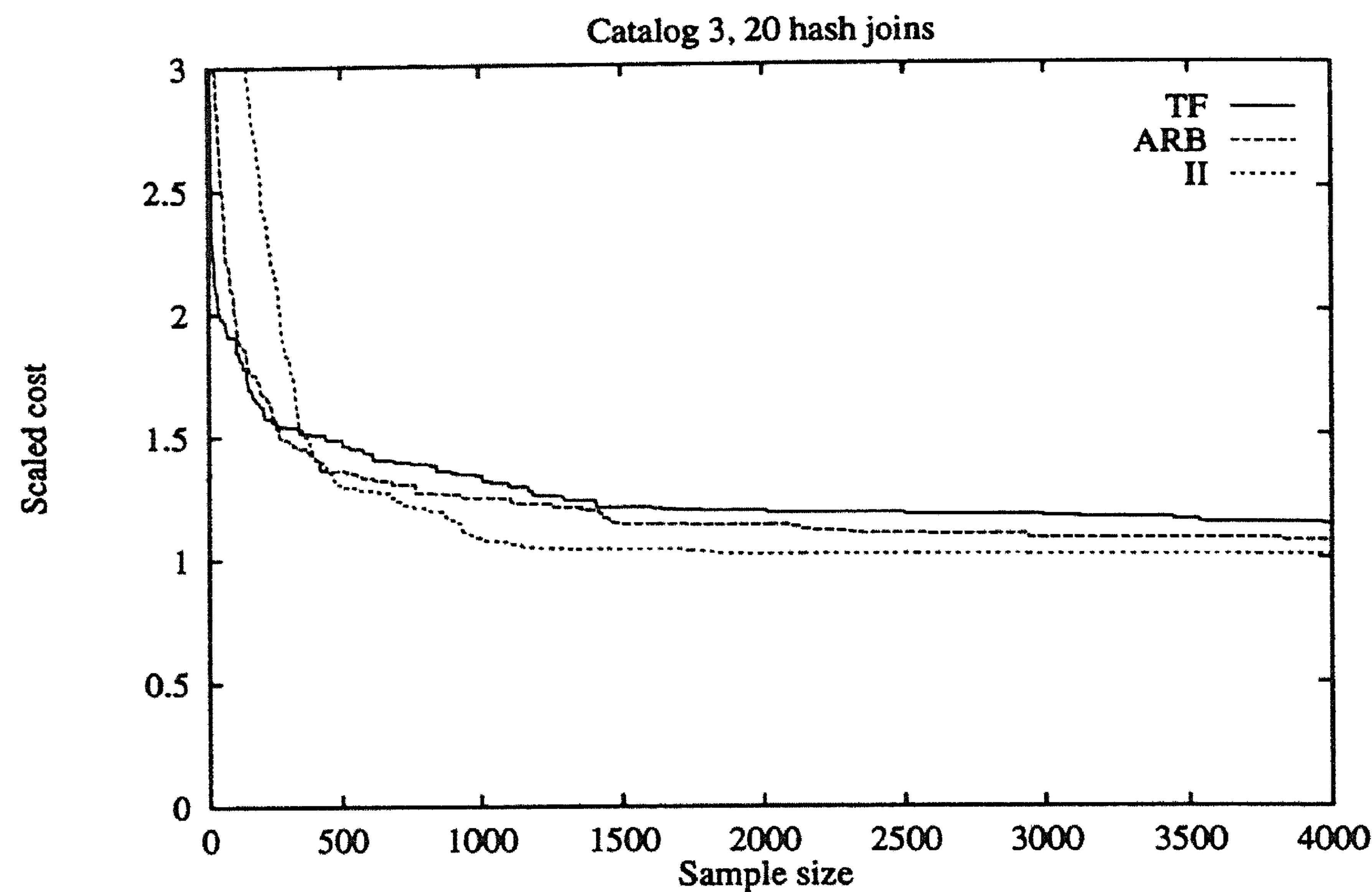


Figure 8.6: Space of hash join trees for the original high variance catalog

trees with the application of transformations. Other multi-phase optimization schemes have been proposed in [Kan91, LVZ93], but they still rely mainly on transformations to generate alternatives.

8.4.1 Set Based Iterative Improvement

It is reasonable to consider starting the search by generating a predefined number of join trees (TF-phase), followed by one transformation-based local optimization. During this local optimization phase no new random starting points are generated. A generalization of this idea is what we call the *Set-based Iterative Improvement* (SII_n) algorithm. This hybrid algorithm is an II algorithm that uses the best join tree of a randomly generated set as starting state for a local optimization. The n represents the size of the randomly generated start set. Figure 8.7 shows the pseudo-code of the algorithm.

8.4.2 Experimental results

Figure 8.8 shows the performance of SII_{100}, as well as TF and II for the space of join trees, when using the enlarged catalog 3, the original set of

```
PROCEDURE SII(n) {
  minS = infinite; // with cost(infinite) = infinite
  WHILE not (stopping_condition) DO {
      S = random state;
      FOR i = 1 TO n - 1 DO {
          S' = random state;
          IF cost(S') < cost(S) THEN S = S';}
      WHILE not (local_minima(S)) DO {
          S' = random state in neighbors(S);
          IF cost(S') < cost(S) THEN S = S';}
      IF cost(S) < cost(minS) THEN minS = S;}
  return(minS);}
```

Figure 8.7: Set-Based Iterative Improvement

transformation rules and only hash-joins were allowed. The graph of the SII_{100} algorithm reflects the behaviour of both the TF and II algorithm. It converges as fast as the TF graph in the first part of the graph and then picks up the behaviour of the II algorithm, resulting in very good quality join trees. Figure 8.8 is typical for the behaviour of the SII algorithm.

Figure 8.9 shows the performance of SII_{100} on the space of join trees plus join-algorithm selection, also in combination with enlarged catalog 3. Although the TF algorithm has a weak performance for this search space, the SII_{100} algorithm maintains its good behaviour.

Kang showed in [Kan91] that the local optimization runs of the II algorithm can be quite long. From our experiments with the hybrid algorithm it shows that if the local optimization runs are started with "better" join trees the runs are shorter, so more runs can be done. The SII_n algorithm uses this observation to find good join trees faster.

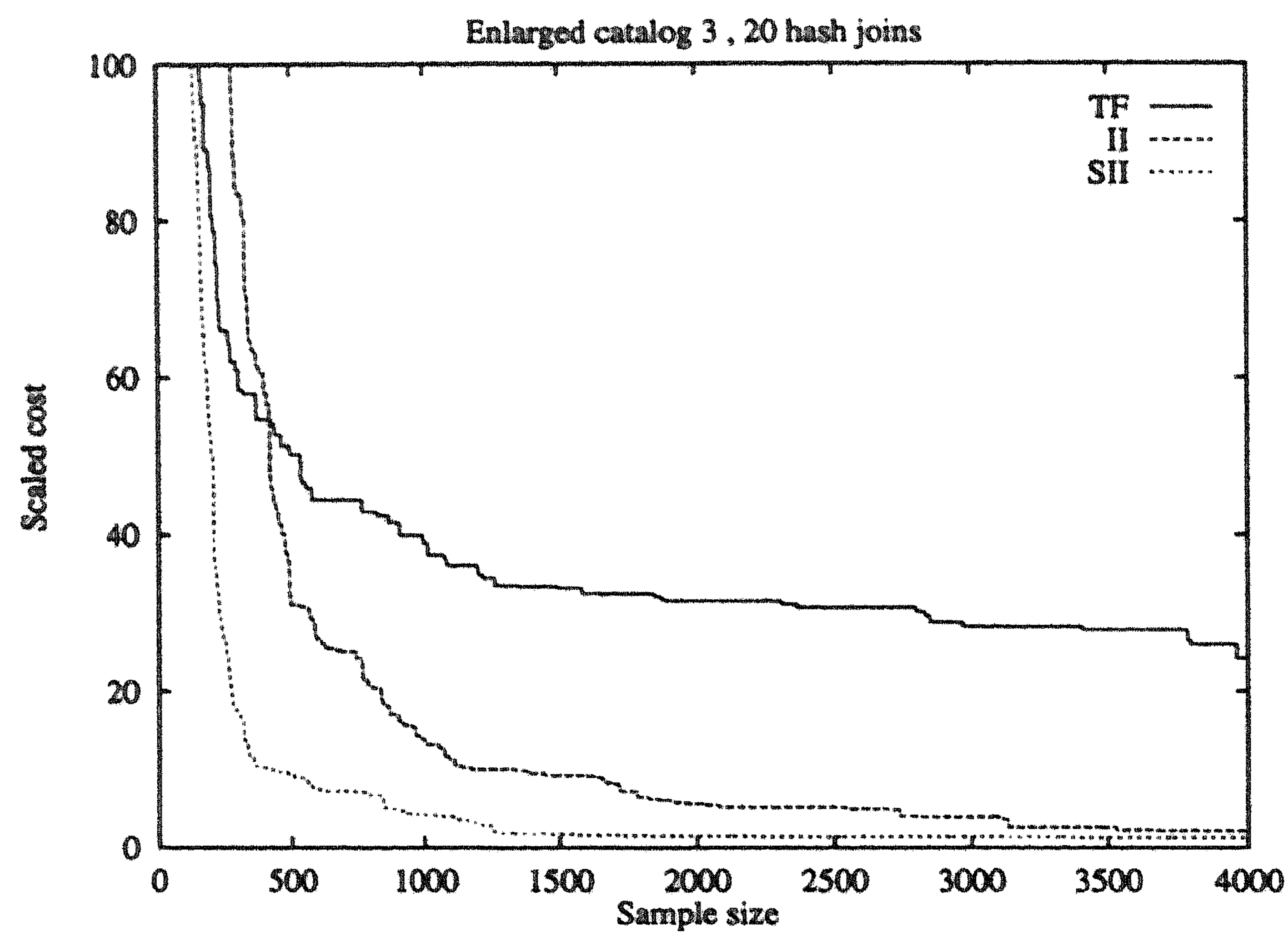


Figure 8.8: Hybrid search on the restricted space of hash-joins

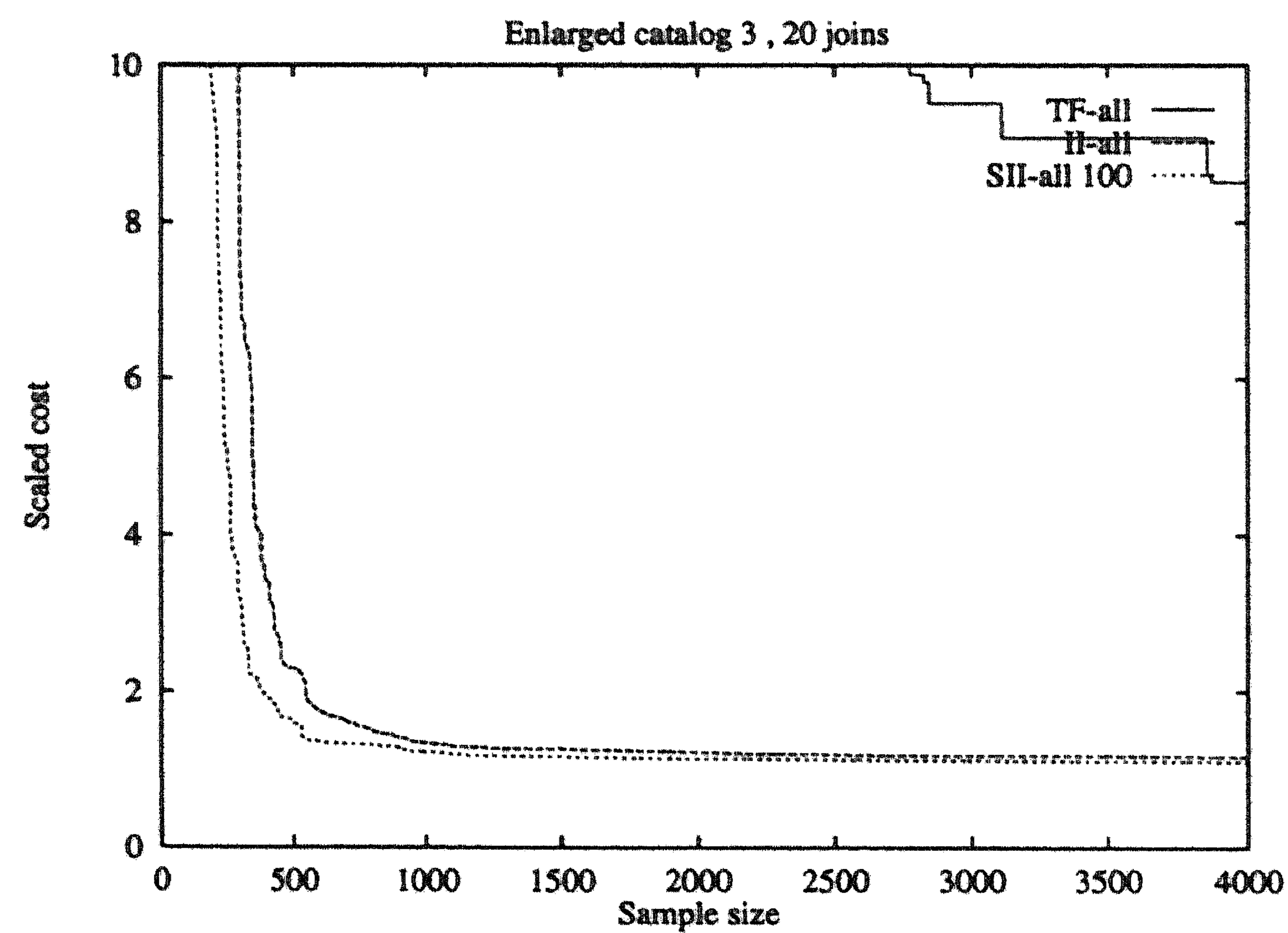


Figure 8.9: Hybrid search using all available join algorithms

8.5 Summary

In this chapter we examined the impact of several factors on the performance of probabilistic query optimization algorithms, in particular the relative behaviour of random picking of solutions with respect to transformation-based search. The results of random picking give a direct indication of the proportion of good solution in the search space, while the transformation-based search also depends on the topology imposed by the specific set of transformations used.

The experiments show that the results obtained in Chapter 7 for a main-memory database remain valid, for the most part, when the I/O-based cost model of [IK90, Kan91] is used instead. A transformation-free algorithm finds good join trees faster than a transformation-based approach, but the transformation-based search finds the best join trees in the end. This happens because the ratio of good join trees is substantial and the topology imposed by associativity/commutativity/exchange transformations does not seem to aid the search significantly at the beginning of the process. But as the search process progresses the imposed topology starts to aid the search process. Experiments showed that this also holds for a set of arbitrary transformations.

We then studied the effect of selecting a join algorithm, in addition to a join evaluation order. In this case the search space becomes the product of two exponentially large spaces, and its properties turn out to be qualitatively different from those of selection of a join tree alone. The proportion of good join trees decreases in this combined space, and at the same time the topology induced by the change-algorithm rule seems to favour the transformation-based search.

When the cost of successive candidates is highly correlated with the former ones, the search process improves slowly, but can find good solutions at the end. The transformation-based local search is therefore effective towards the end of the search, to refine solutions obtained quickly by less local methods. To verify this we described and tested a two-phase optimization approach that starts with random picking to generate good join trees quickly, and then applies transformations for further refinement. The result is a combination of the best of both search strategies: fast convergence to solutions of very high quality. We believe this hybrid approach is basically the best alternative in a "blind" probabilistic search — i. e. domain-independent and without using heuristics— probably with an additional Simulated Annealing phase at the end as suggested in [IK90].

Chapter 9

Conclusions

This final chapter summarizes the research described in this thesis and discuss its contributions to the area of query optimization. Also, directions for future research are given.

9.1 Summary

For database systems we have addressed the problem of finding an optimal, or near optimal, join tree. In Chapter 1 the research problems and objectives were stated after a brief introduction of databases and query optimization problems. The general definitions used throughout the thesis were given in Chapter 2 as well as a discussion of existing join tree selection algorithms.

In Part I, the complexity issues related to the repeated generation of elements and the number of join trees for acyclic queries is addressed, as well as the random generation of join trees.

In Chapter 3 we have analysed the problem of duplicate generation for transformation based optimizers that explore a space exhaustively. For several query graph topologies the space of either linear or bushy join trees was considered. For transformation based optimizers the generation of duplicates is a serious problem. Even for small queries the number of duplicates exceeds the number of new operators, and it increases dramatically with the size of the query. In particular, for the Volcano-type optimizers the ratio of duplicates over new operators can be up-to $O(2^{n \log(4/3)})$. The detailed complexity analysis developed is the first that we are aware of for this type of optimizers.

In Chapter 4 we described in detail efficient sets of transformation rules, for several classes of query graph topologies for both bushy and linear join trees. Our approach to an efficient transformation-based generation algorithm is to keep track of the transformation rules that can still be applied without generating duplicates at any point in the search space. The overhead of the method consists of a few bits per operator. The conditioned application of rules can be incorporated easily in the existing framework of modern query optimizers, and tests corroborate that considerable performance improvements result from the large reduction of generated operators. The performance improvement gained by avoiding the generation of duplicates is significant in practice.

Subsequently, in Chapter 5 we described techniques and procedures for counting the number of bushy or linear join trees to evaluate an acyclic query. The difficulty of the counting problem results from the fact that there is no one-on-one mapping between join trees and a simple combinatorial structure. Our concept of a standard decomposition graph provides a supporting structure for counting, because it defines a canonical construction for each tree. In addition, computing an array of values that characterizes the number of canonical constructions can be computed bottom up in an efficient way.

In Chapter 6 we have described how the one-on-one mapping between integers and join trees can be done using the RANK and UNRANK procedures. These algorithms are built on the join tree counting theory and techniques developed in Chapter 5. The UNRANK procedure makes it possible to construct an efficient algorithm which generates join trees at random with a uniform distribution. Also an improved algorithm for generating join trees at random has been described which reduced the time complexity from $O(n^2 log n)$ to $O(n^2)$.

The theory of Part I is supported by empirical evaluation in Part II. Experiments with optimization algorithms were described, which heavily depend on the random generation of join trees.

In Chapter 7 the transformation-free (TF) optimization algorithm was introduced. This algorithm selects join trees at random from the space of alternatives while keeping track of the tree with the lowest estimated cost. The performance of TF has been compared to Simulated Annealing and Iterative Improvement for various database catalogs and queries. The result is that ***transformation-based optimizers find very good solutions, but take a long time, compared with the transformation-free algorithm.***

Exhaustive exploration or sampling of the search space of a class of queries provides a precise measure of the run length required to hit a good join tree. Our results then provide a natural baseline against which the

added value of applying transformations and heuristics can be quantified.

In Chapter 8 we examined the impact of several factors on the performance of probabilistic query optimization algorithms. In particular the relative behaviour of random picking of solutions with respect to transformation-based search. The experiments show that the results obtained in Chapter 7 for a main-memory database remain valid, for the most part, when the I/O-based cost model of [IK90, Kan91] is used instead. A transformation-free algorithm finds good join trees faster than a transformation-based approach, but the transformation-based search finds the best join trees in the end. This happens because the ratio of good join trees is substantial and the topology imposed by associativity / commutativity / exchange transformations does not seem to aid the search significantly at the beginning of the process. But as the search process progresses, the imposed topology starts to aid the search process. Experiments showed that this also holds for a set of arbitrary transformation rules.

We then studied the effect of selecting a join algorithm, in addition to a join evaluation order. In this case the search space becomes the product of two exponentially large spaces, and its properties turn out to be qualitatively different from those of selection a join tree alone. The proportion of good join trees decreases in this combined space, and at the same time the topology induced by the change-algorithm rule seems to favour the transformation-based search.

The incremental transformation-based, local search is effective towards the end of the search, while random selection is more effective at the beginning of the optimization process. To study the combination of both effects, we described and tested a two-phase optimization approach that starts with random picking to generate good join trees quickly, and then applies transformations for further refinement. The result is a combination of the best of both search strategies: fast convergence to solutions of very high quality. We believe this hybrid approach is basically the best alternative in a "blind" probabilistic search — i. e. domain-independent and without using heuristics— probably with an additional Simulated Annealing phase at the end as suggested in [IK90].

9.2 Future research

Design of an appropriate set of transformation rules is an important task seldom emphasized in the rule-based optimization literature. For several classes of query graph topologies duplicate-free sets of transformation rules have been shown. However, no methodology exists for creating such duplicate-free sets. Moreover, it is not likely that for every set of trans-

formation rules a duplicate-free alternative exists.

Future research could be directed at determining for which conditions a set of transformation rules can be converted into a duplicate free one. Also the interaction between duplicate-free rules and other rules of a transformation based system is an open area of research.

For acyclic queries both the counting problem has been solved and the random generation of join trees with a uniform distribution. Although acyclic queries cover perhaps most of the queries posed in practice, cyclic queries are frequent enough to deserve attention. Therefore, a second track for future research could be focused on studying the class of cyclic queries, but the problem is more difficult. Many database problems become significantly more complex when cyclic structures are allowed (see for example [BFMY83]), and so-far the techniques we use for the acyclic case do not seem to extend easily to cyclic queries.

For the experimental part there are several open issues to be addressed. First, as soon as the random generation of join trees for cyclic queries is solved the experiments should be extended accordingly. Second, by allowing operator trees to contain both join operators and Cartesian products, cheaper solutions can be found than allowing only join trees. However, these operator trees also introduce solutions which are much more expensive. It is unclear yet how the cost distribution over this space of operator trees is and how the Transformation Free optimization algorithm would perform. Finally, the incorporation of heuristics in probabilistic query optimizers in a robust manner should be studied. This should be done by "rigging the odds" in favour of the better join trees — e.g. if, in general, bushy join trees are known to be cheaper than linear trees they should get a higher chance of being generated.

Also exploring the possibilities of 2-phase algorithms is a direction for future research. A first step is to obtain indicators which are easy to compute and give a good characterization of the search space. This would make it possible to use specific optimizers for specific types of search spaces.

Appendix A

DBS3 Measurements

The graphs contained in this appendix show the results of the experiments as described in chapter 7. For each combination of query and catalog a graph with the "average" performance over 20 runs of the three optimization algorithms is given.

For the graph in figure A.8 a graph at a larger scale is given in A.9. At a larger scale the general behaviour of the TF algorithm, when compared to II, still holds. However the II finds acceptable and good plans after exploring only 783 alternatives while TF needs to explore another 546 plans.

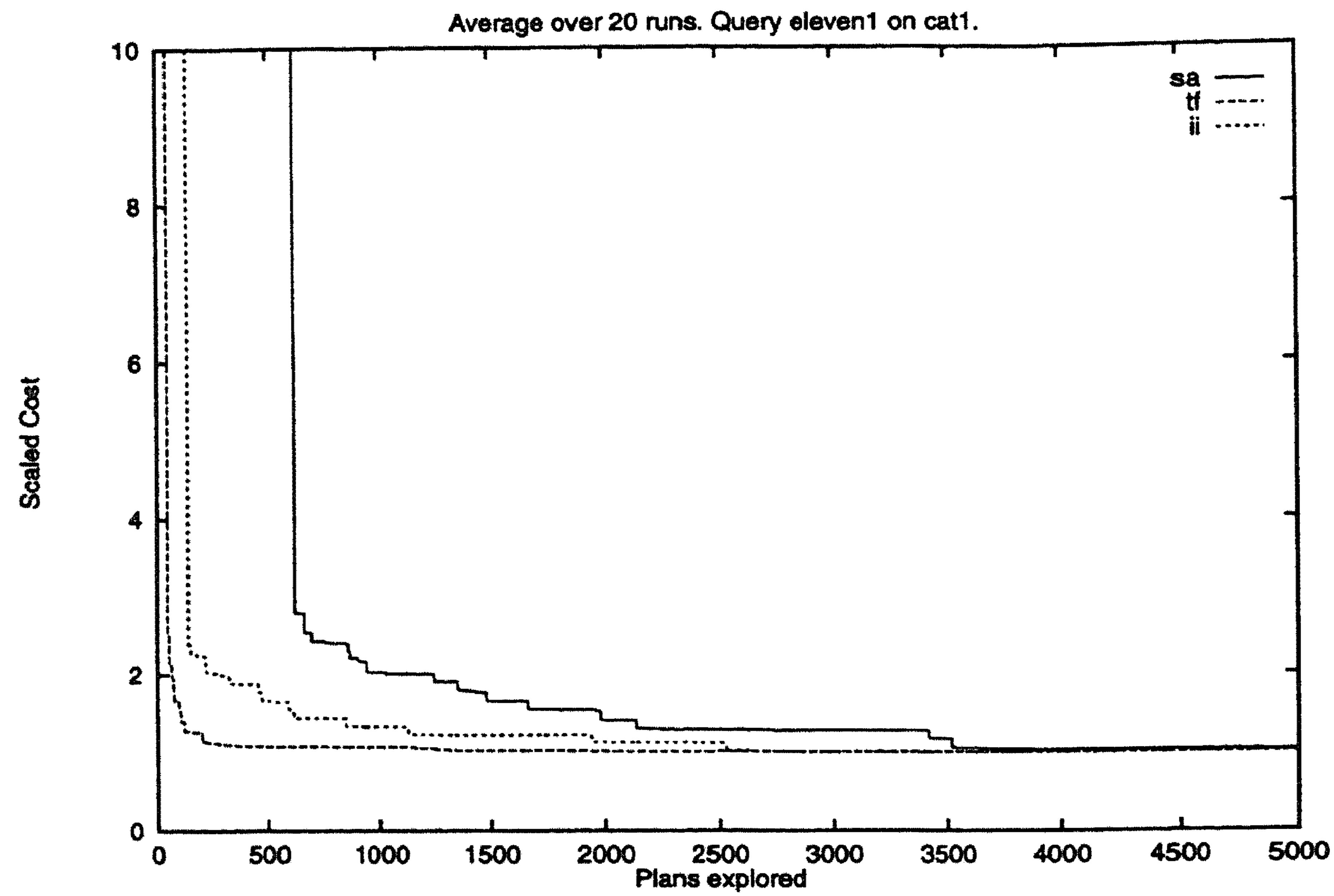


Figure A.1: Average of cost of solution found.

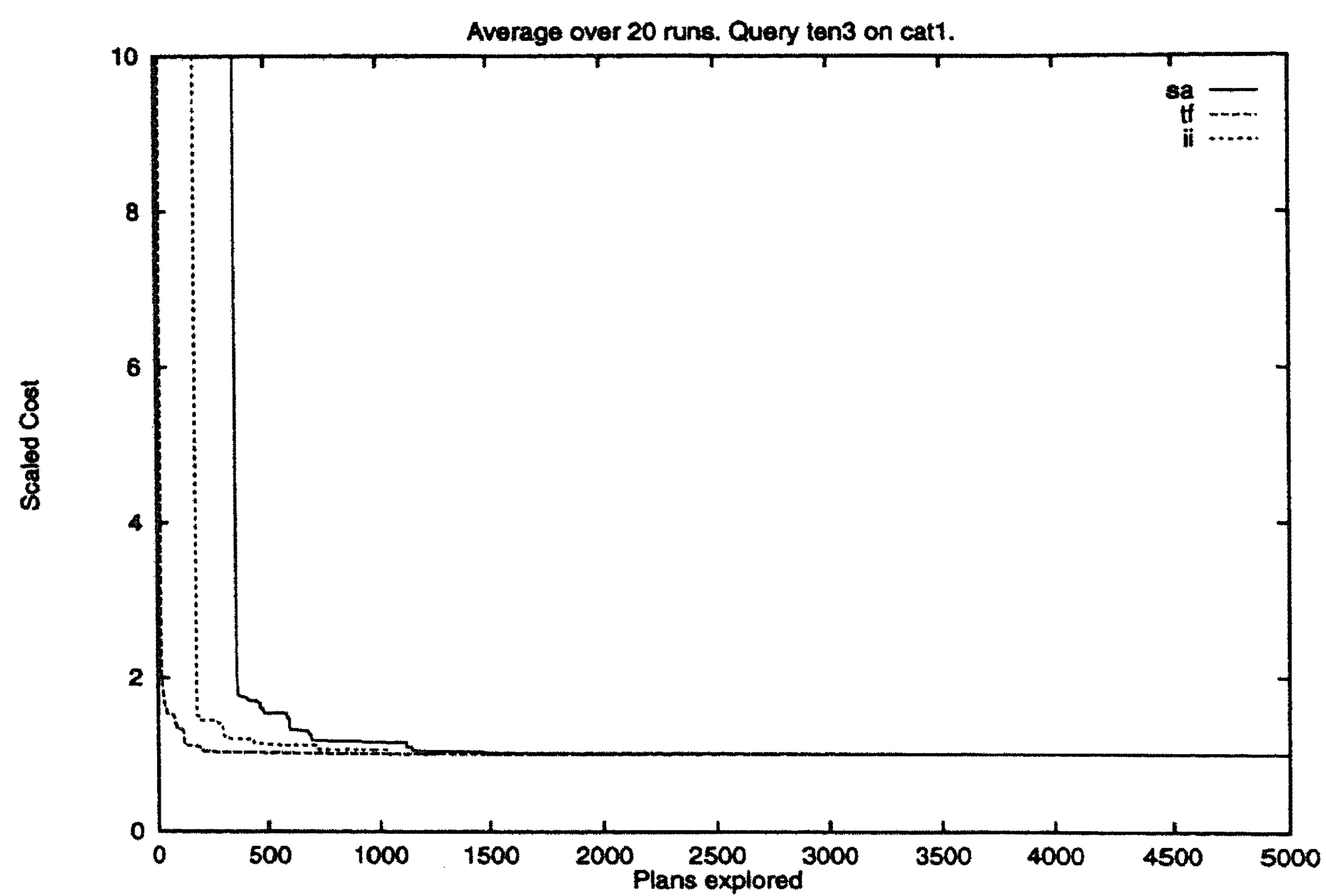


Figure A.2: Average of cost of solution found.

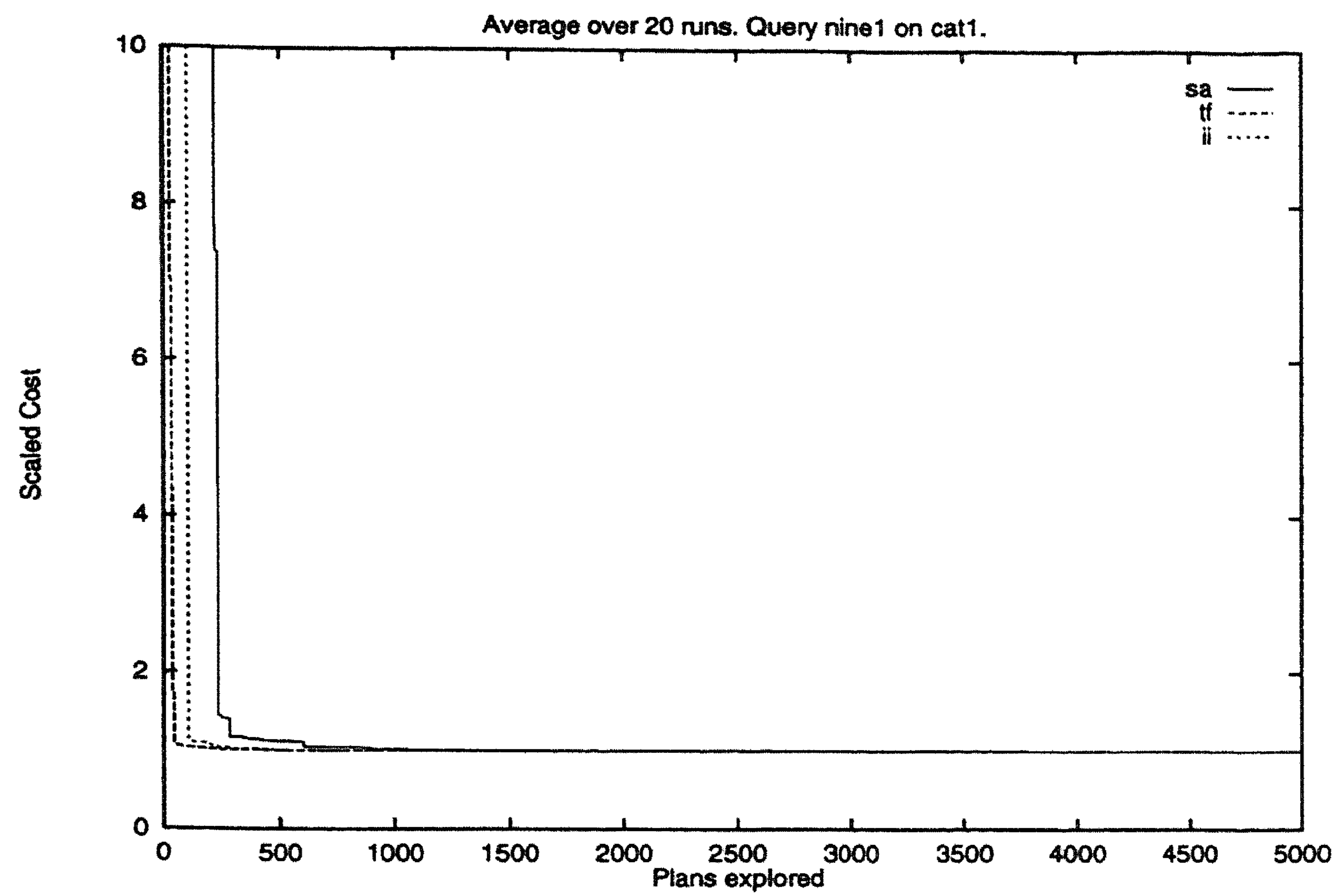


Figure A.3: Average of cost of solution found.

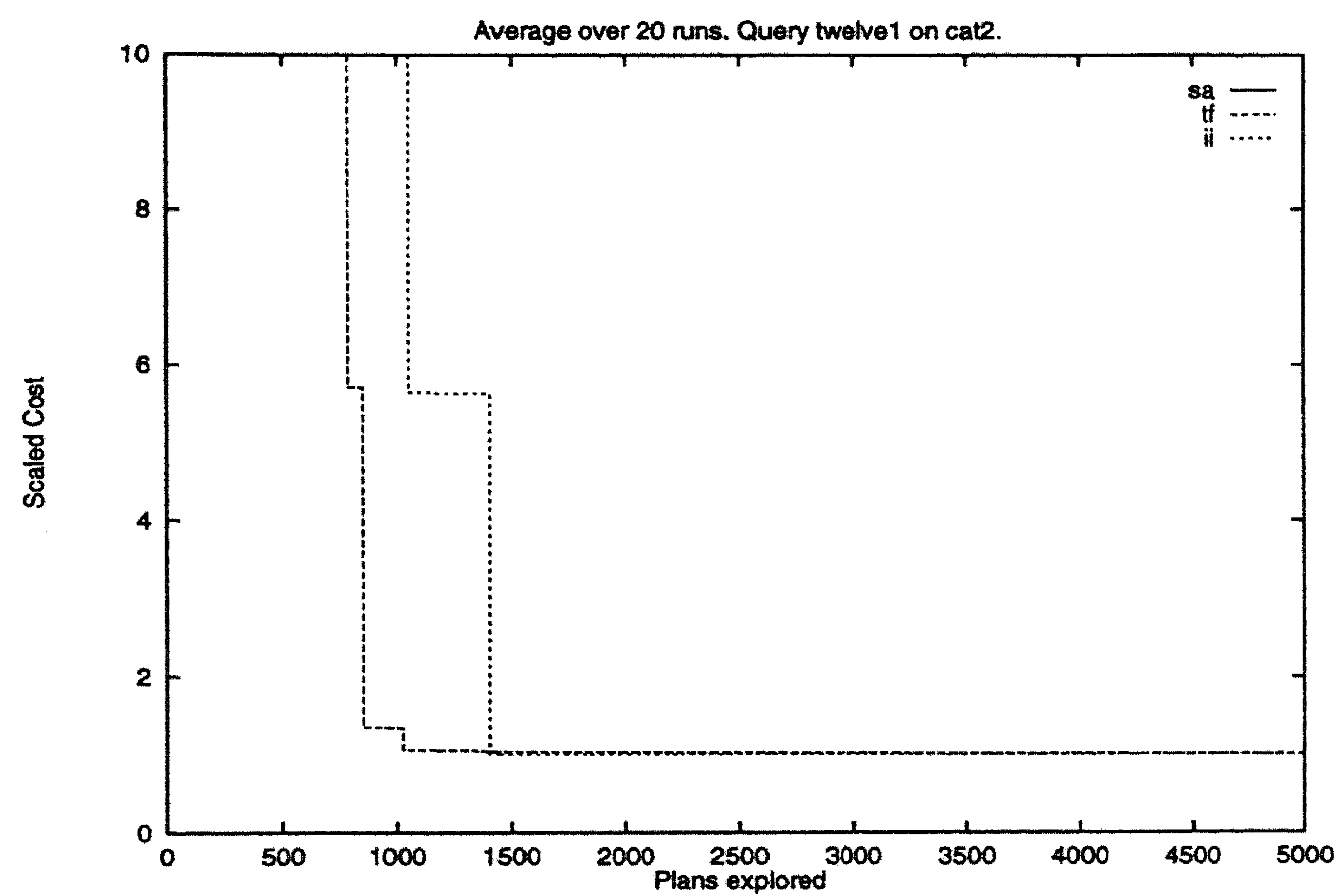


Figure A.4: Average of cost of solution found.

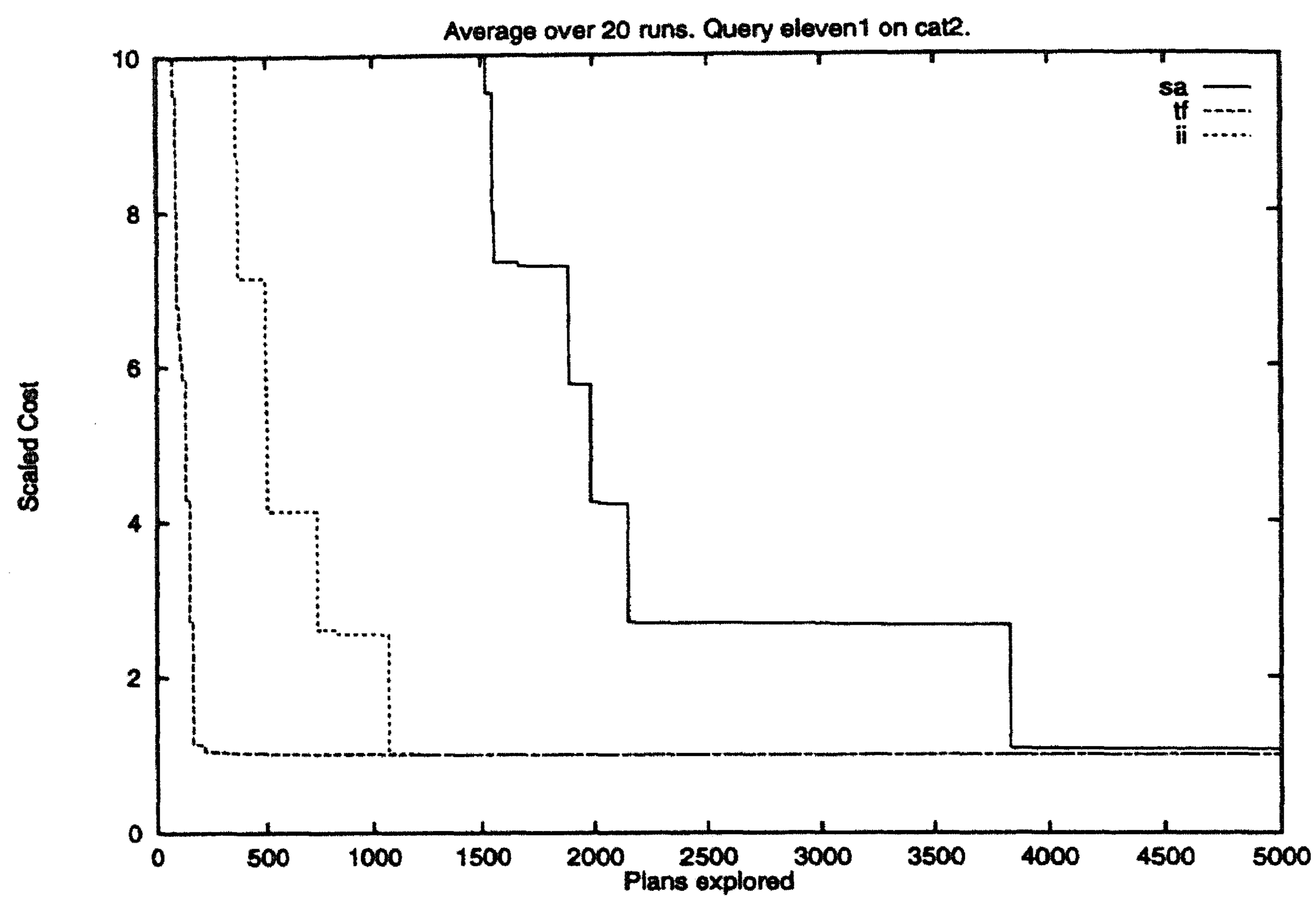


Figure A.5: Average of cost of solution found.

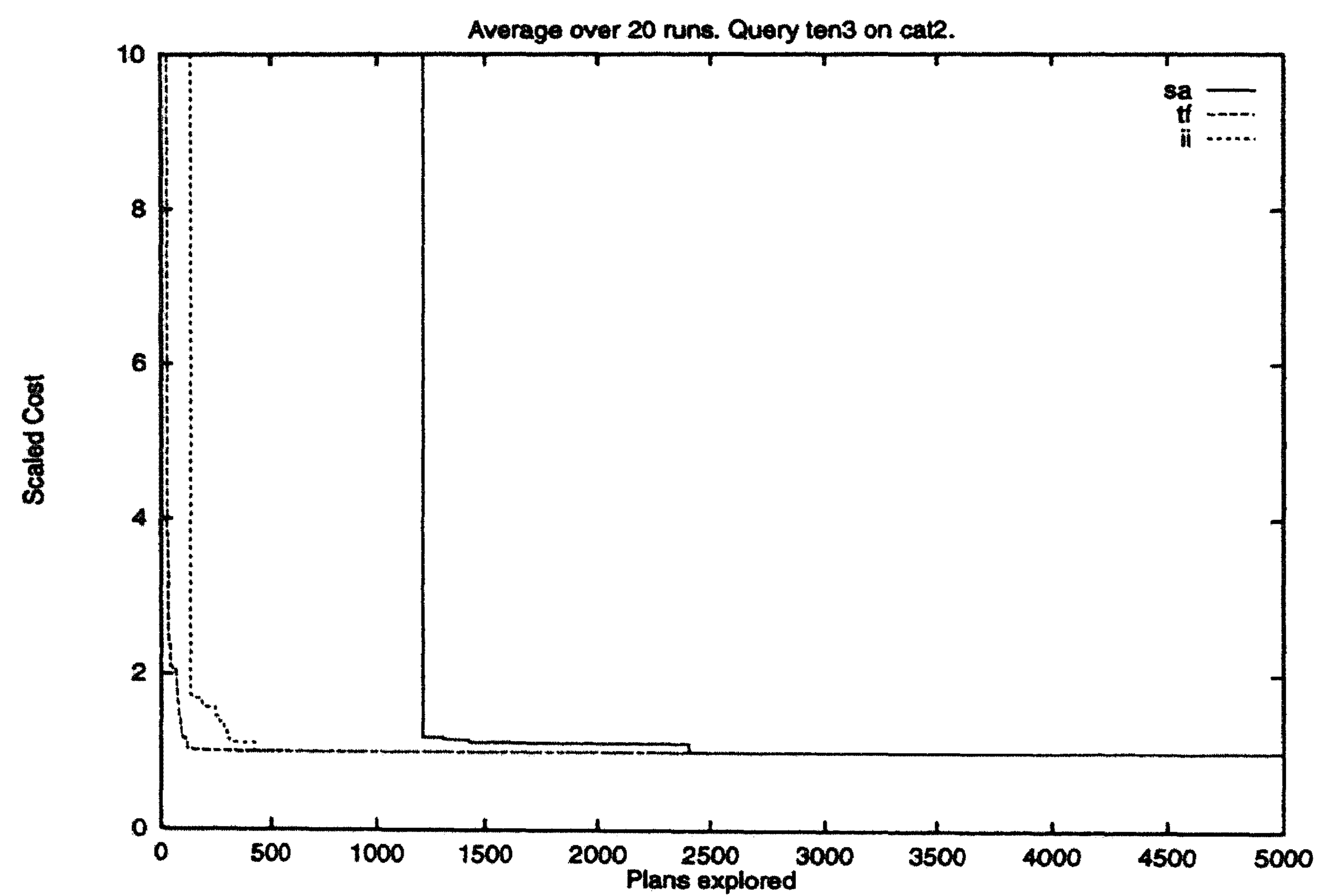


Figure A.6: Average of cost of solution found.

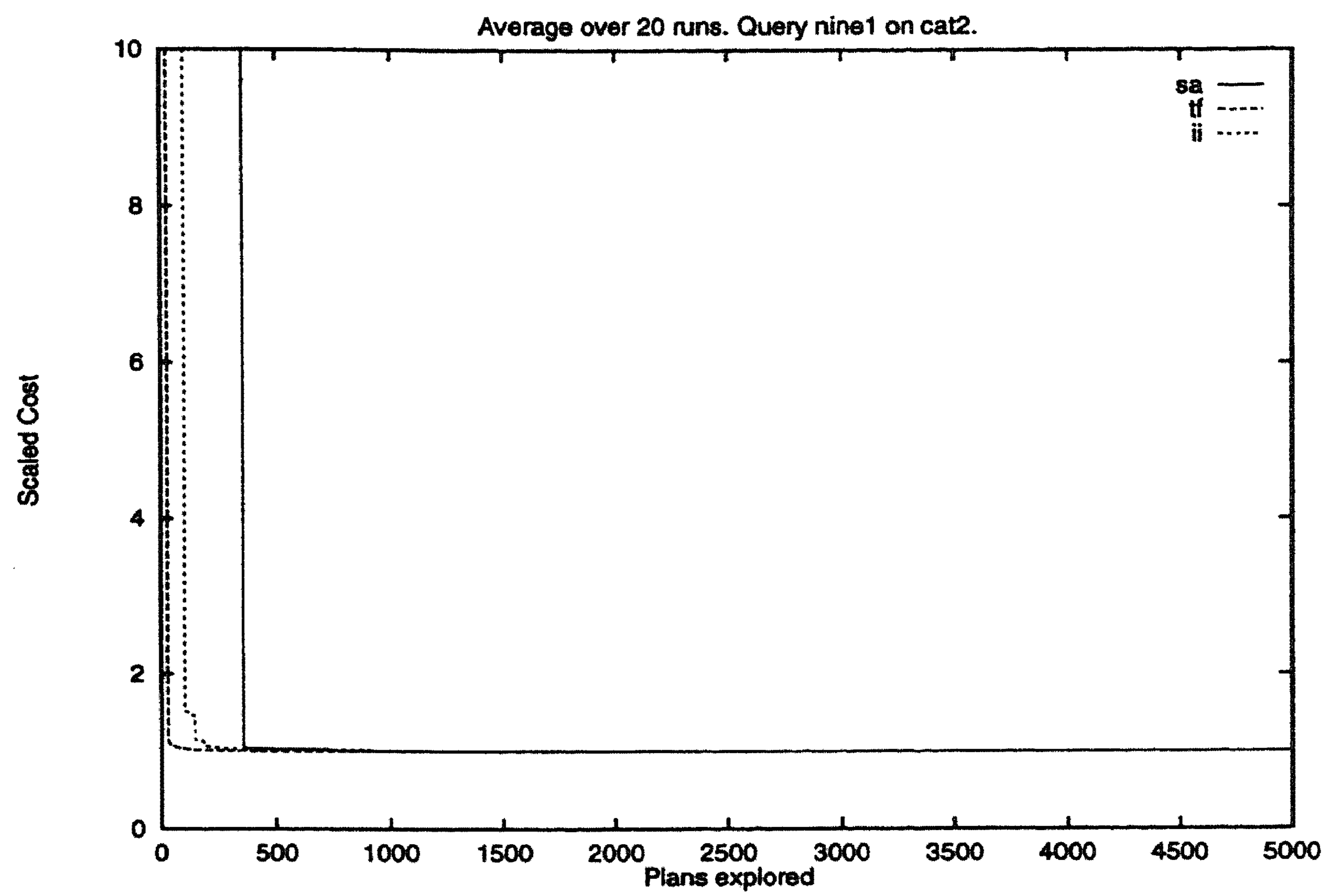


Figure A.7: Average of cost of solution found.

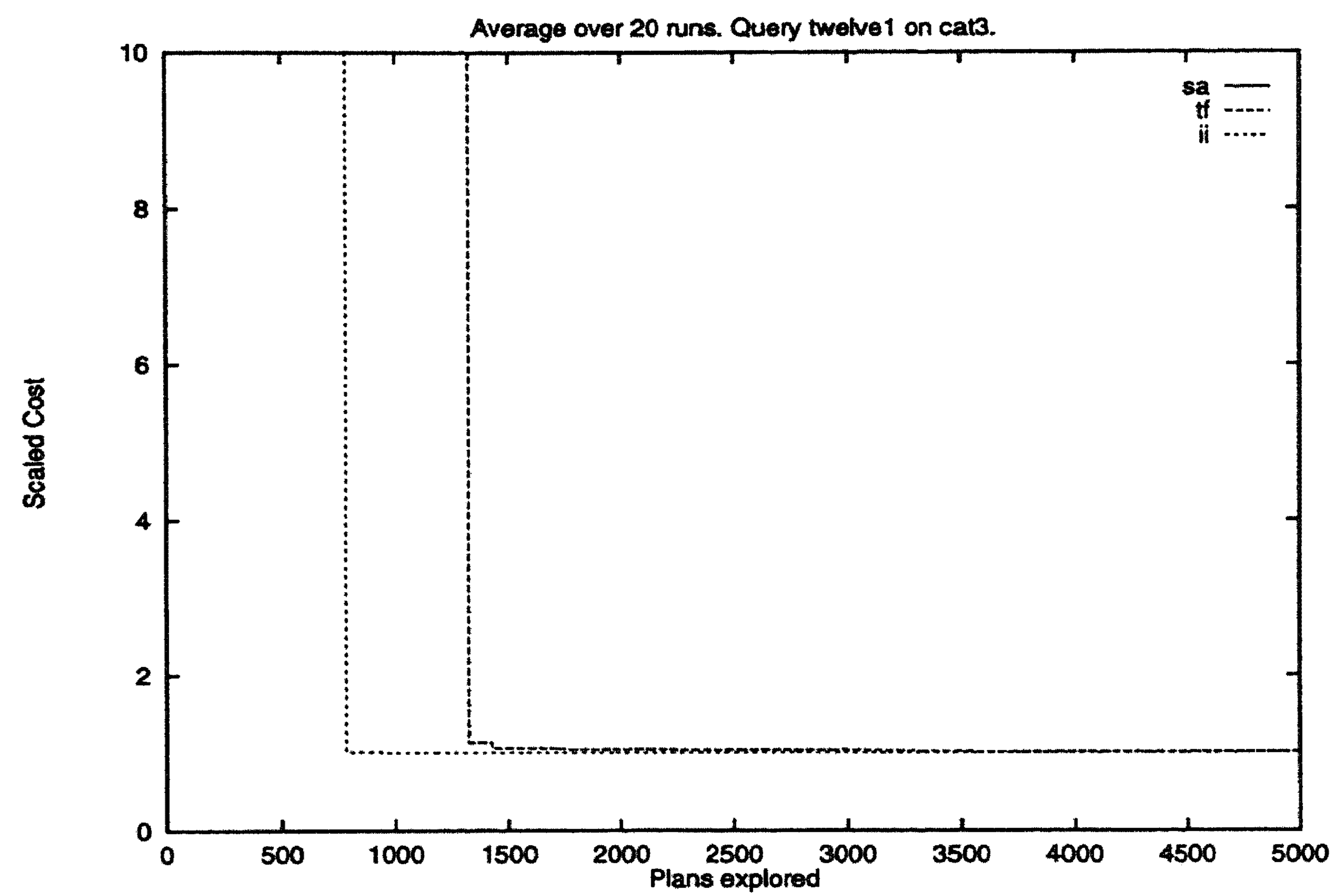


Figure A.8: Average of cost of solution found.

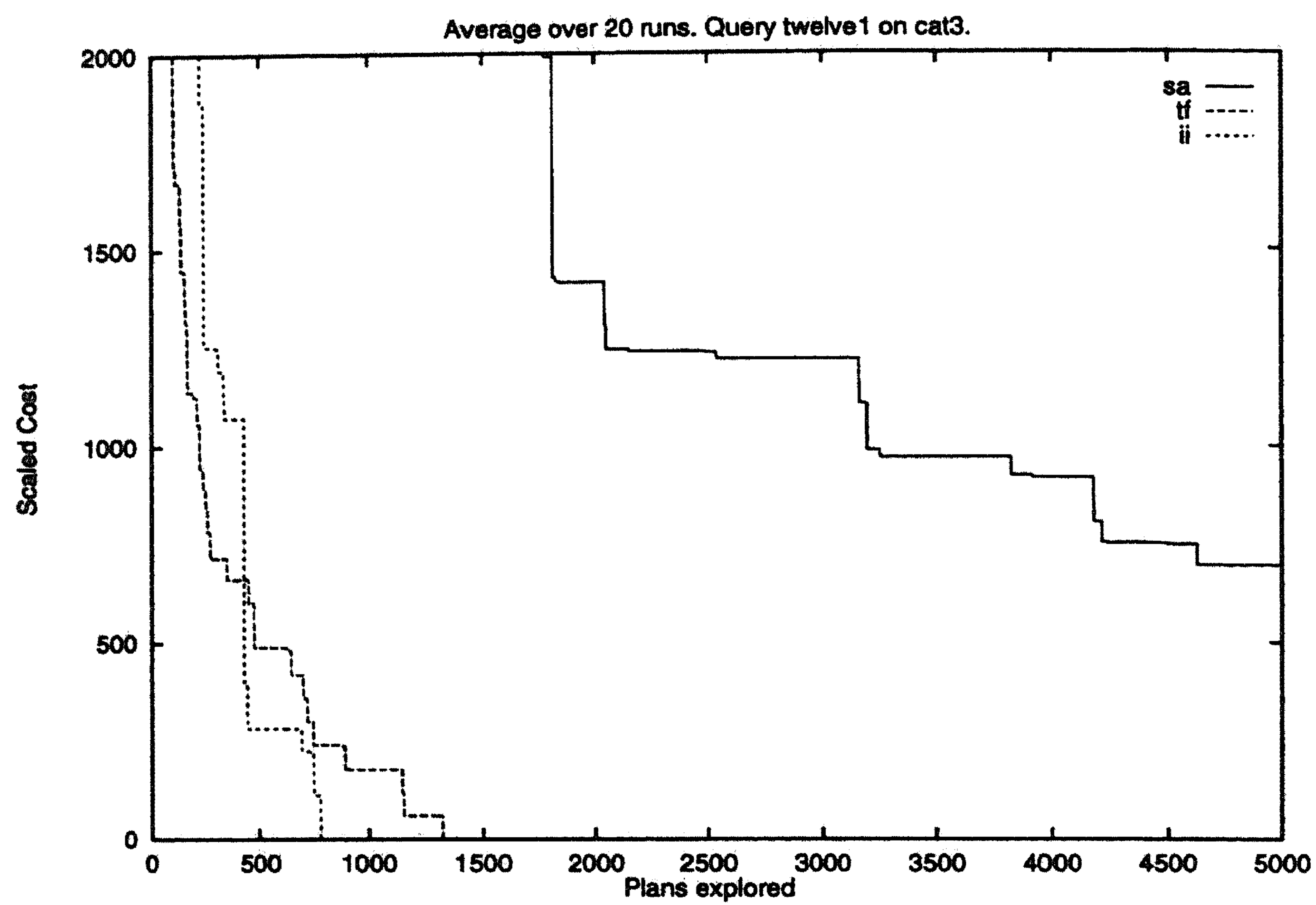


Figure A.9: Average of cost of solution found.

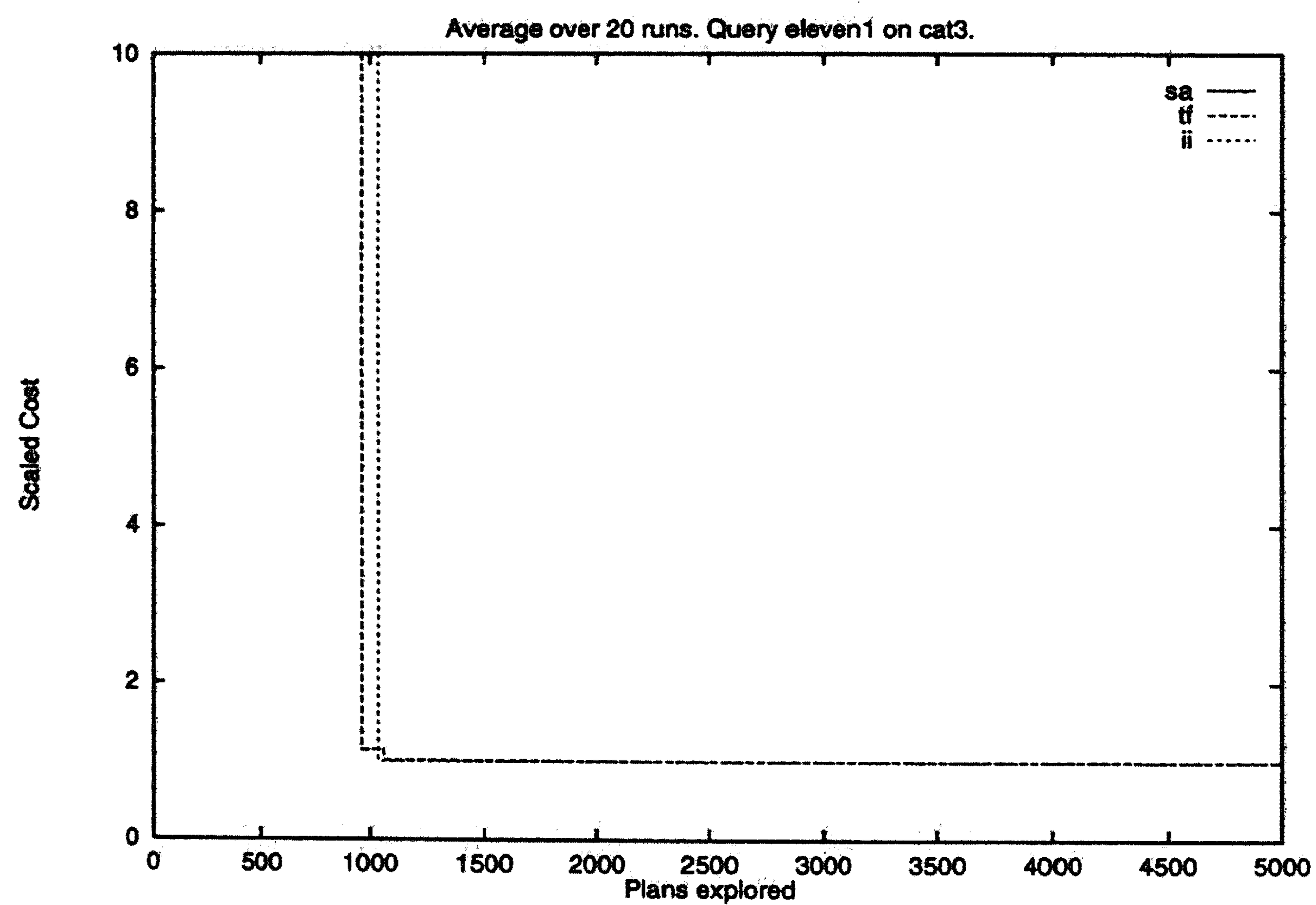


Figure A.10: Average of cost of solution found.

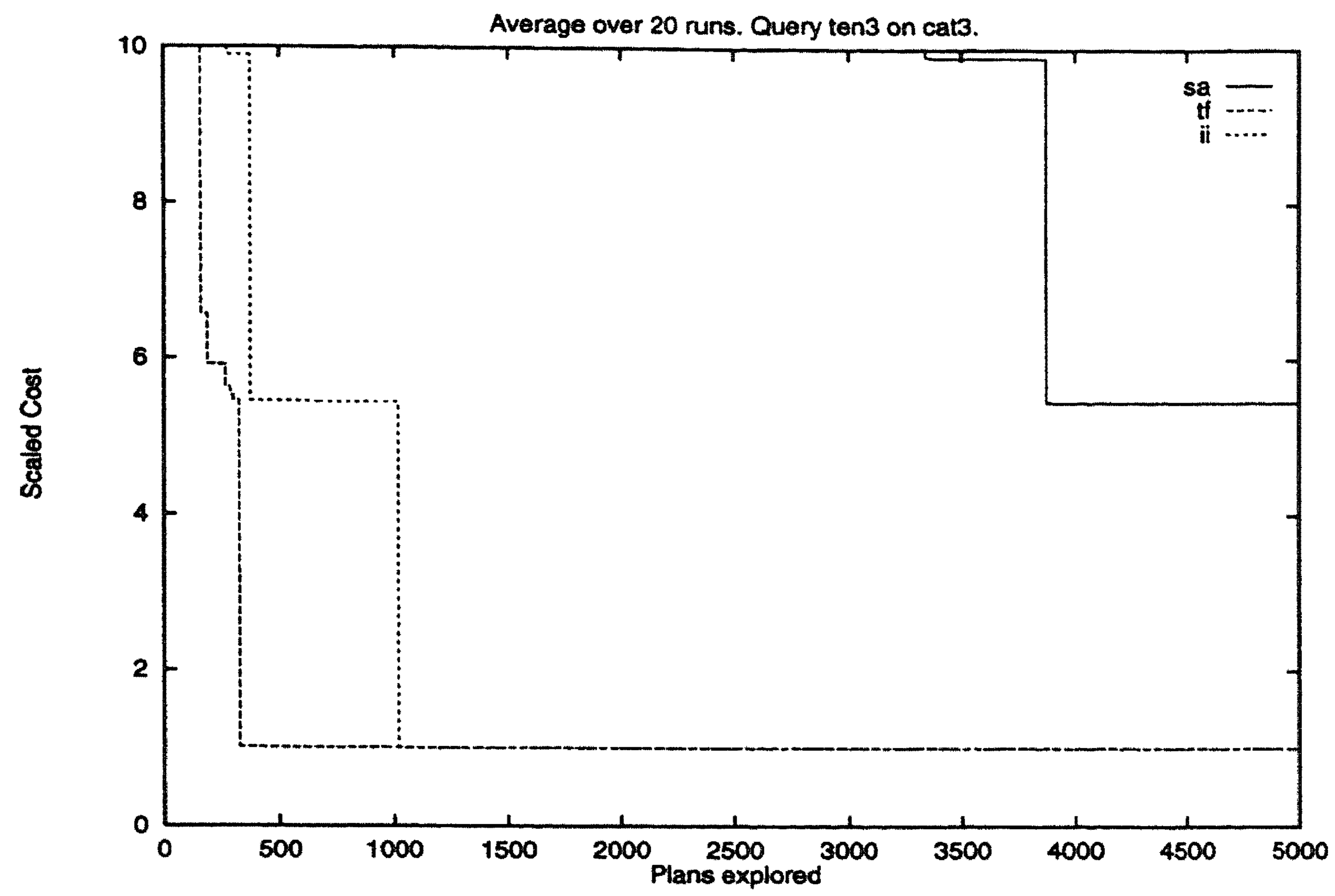


Figure A.11: Average of cost of solution found.

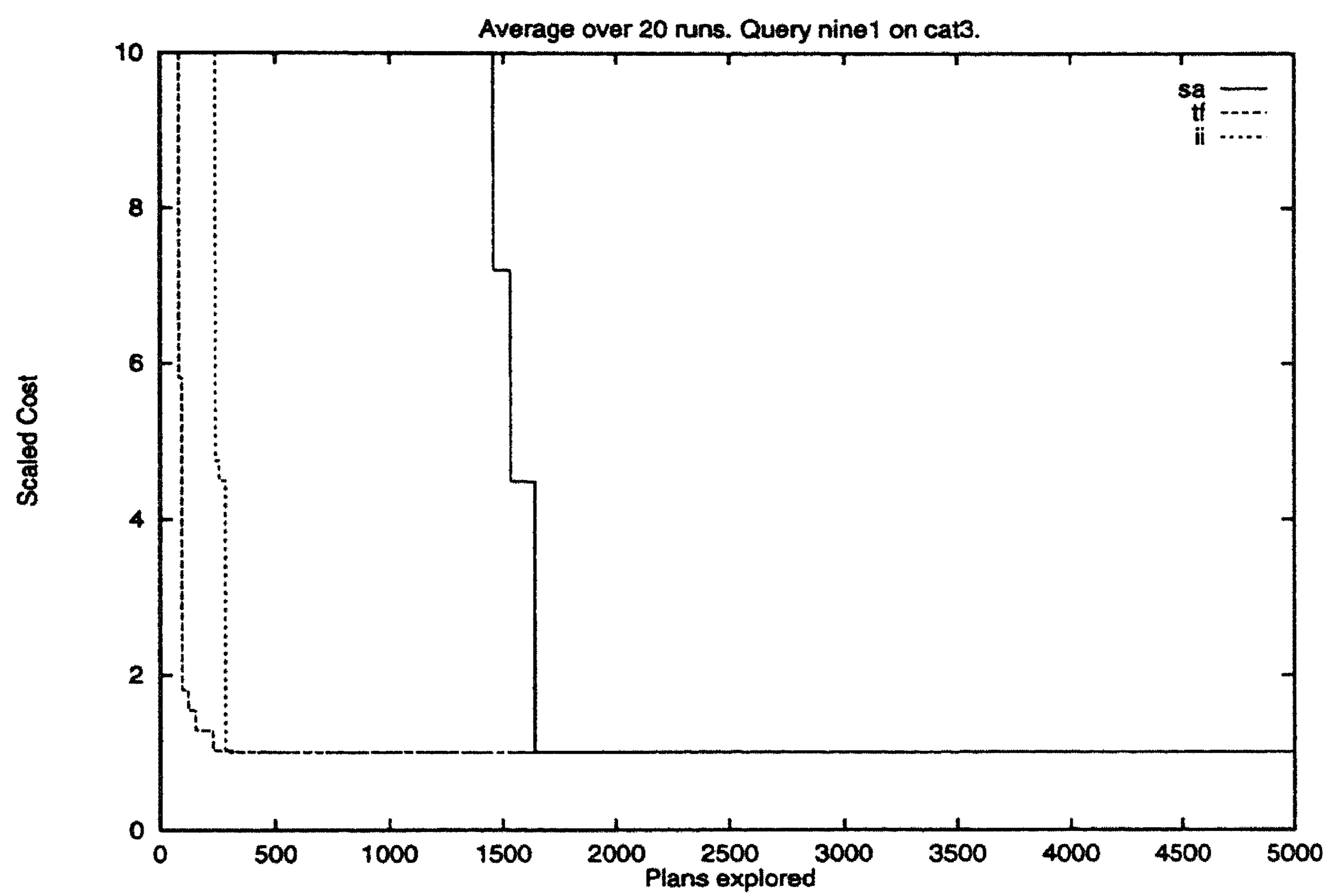


Figure A.12: Average of cost of solution found.

Appendix B

Query graphs

The query graphs shown in this appendix were used in the experiments of Chapter 8. The queries were generated at random for a database schema that consists of 110 relations. With each relation 4 attributes.

4:
```
107 — 36 - 81 ——— 32 ——— 63
```

5:
```
107 — 36 - 81 ——— 32 ——— 63
            |
            64
```

6:
```
107 — 36 - 81 ——— 32 ——— 63 - 55
            |
            64
```

7:
```
107 — 36 - 81 ——— 32 ——— 63 - 55
            |     |
            64    31
```

8:
```
                  8
                  |
107 — 36 - 81 ——— 32 ——— 63 - 55
            |     |
            64    31
```

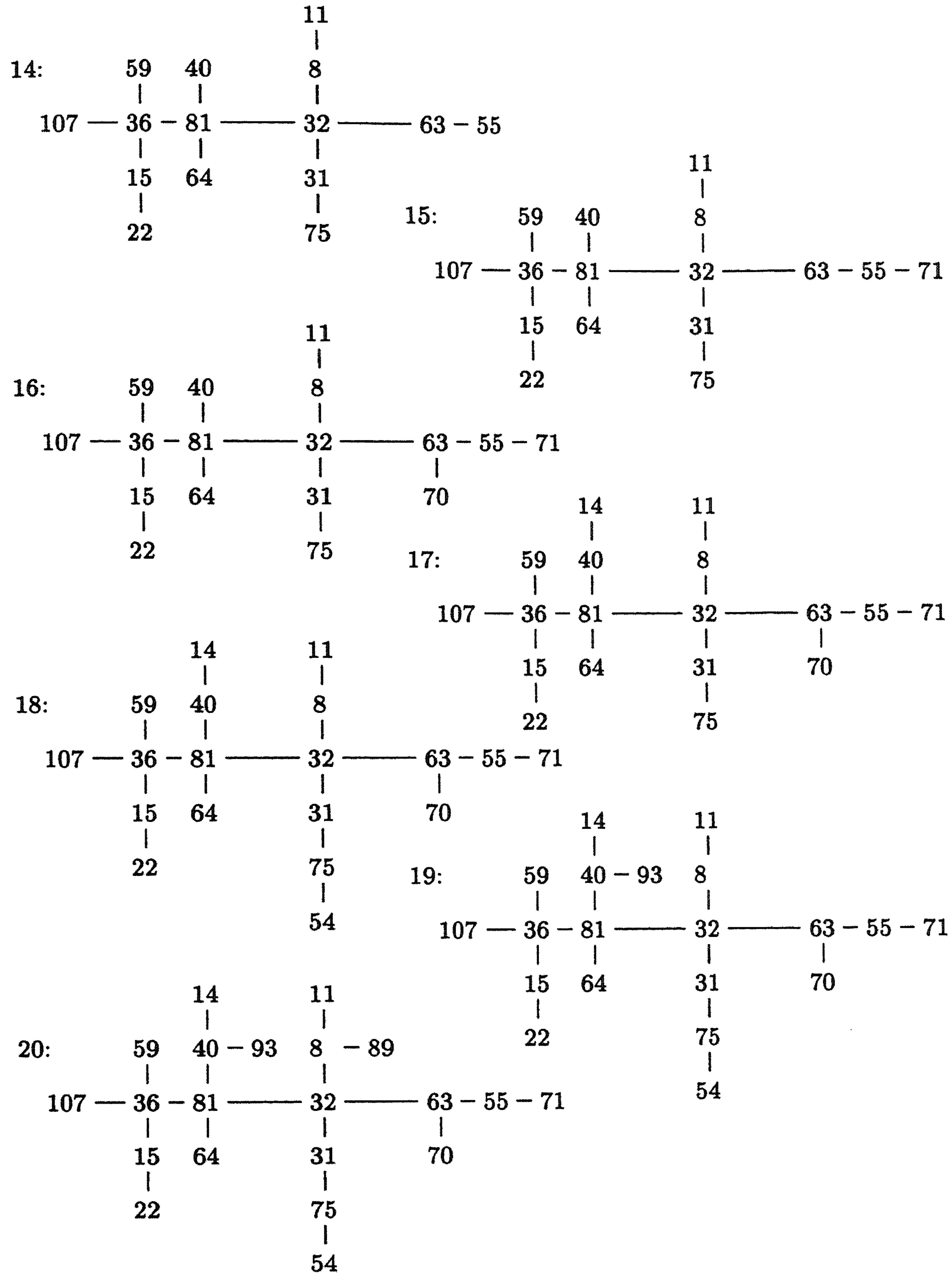
9:
```
                  8
                  |
107 — 36 - 81 ——— 32 ——— 63 - 55
      |     |     |
      15    64    31
```

10:
```
                  8
                  |
107 — 36 - 81 ——— 32 ——— 63 - 55
      |     |     |
      15    64    31
      |
      22
```

11:
```
      59          8
      |           |
107 — 36 - 81 ——— 32 ——— 63 - 55
      |     |     |
      15    64    31
      |
      22
```

12:
```
      59          8
      |           |
107 — 36 - 81 ——— 32 ——— 63 - 55
      |     |     |
      15    64    31
      |           |
      22          75
```

13:
```
      59    40    8
      |     |     |
107 — 36 - 81 ——— 32 ——— 63 - 55
      |     |     |
      15    64    31
      |           |
      22          75
```


14:
11
59 40 8
107 — 36 – 81 —— 32 —— 63 – 55
15 64 31
22 75
15:
11
59 40 8
107 — 36 – 81 —— 32 —— 63 – 55 – 71
15 64 31
22 75
16:
11
59 40 8
107 — 36 – 81 —— 32 —— 63 – 55 – 71
15 64 31 70
22 75
17:
14 11
59 40 8
107 — 36 – 81 —— 32 —— 63 – 55 – 71
15 64 31 70
22 75
18:
14 11
59 40 8
107 — 36 – 81 —— 32 —— 63 – 55 – 71
15 64 31 70
22 75
54
19:
14 11
59 40 – 93 8
107 — 36 – 81 —— 32 —— 63 – 55 – 71
15 64 31 70
22 75
54
20:
14 11
59 40 – 93 8 – 89
107 — 36 – 81 —— 32 —— 63 – 55 – 71
15 64 31 70
22 75
54

Bibliography

[ACV91] F. Andrès, M. Couprie, and Y. Viémont. A multi-environment cost evaluator for parallel database systems. *Procedings of the 2nd Int. DASFAA Japan*, 1991.

[Ald89] D. Aldous. An introduction to covering problems for random walks on graphs. *Journal of Theoretical Probability*, 2(1):87–89, 1989.

[BFMY83] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, July 1983.

[BMG93] J.A. Blakeley, W. J. McKenna, and G. Graefe. Experiences bulding the open oodb query optimizer. *Proceedings of the ACM SIGMOD Conf on Management of Data, Washington DC*, 1993.

[CM95] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cartesian products. In *Proceedings of the International Conference on Database Theory, Prague*, 1995.

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

[CP85] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill, New York, 1985.

[Dat90] C.J. Date. *An Introduction to Database Systems*, volume 1. Addison Wesley Publishing Company, 5th edition, 1990.

[FMV94] J. C. Freytag, D. Maier, and G. Vossen, editors. *Query Processing for Advanced Database Systems.* Morgan Kaufmann, San Mateo, California, 1994.

[GCD+94] G. Graefe, R. L. Cole, D . L. Davison, W. J. McKenna, and R. H. Wolniewicz. *Query Processing for Advanced Database Systems*. Morgan Kaufmann, 1994.

[GD87] G. Graefe and D. J. DeWitt. The exodus optimizer generator. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 160–172, 1987.

[GJ74] F. Göbel and A.A. Jagers. Random walks on graphs. *Stochastic Processes and their Applications*, 2(1):311–336, 1974.

[GLPK94] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten. Fast, randomized join-order selection —Why use transformations? In *Proceedings of the 20th International Conference on Very Large Databases, Santiago*, 1994. Also CWI Technical Report CS-R9416.

[GLPK95] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten. Uniformly-distributed random generation of join orders. In *Proceedings of the International Conference on Database Theory, Prague*, pages 280–293, 1995. Also CWI Technical Report CS-R9431.

[GLW82] U. Gupta, D. T. Lee, and C. K. Wong. Ranking and unranking of 2-3 trees. *SIAM Journal of Computation*, pages 582–590, August 1982.

[GM93] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. *Procedings of the 9th International Conference on Data Engineering, Vienna, Austria*, pages 209–218, 1993.

[Gra89] G. Graefe. Volcano: An extensible and parallel dataflow query processing system. Technical report, Oregon Graduate Center, 1989.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Gra95] G. Graefe. The cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3):19 – 29, September 1995.

[GV89] G. Gardarin and P. Valduriez. *Relational Databases and Knowledge Bases*. Addison-Wesley, 1989.

[HCL+90] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. Technical Report RJ 7278, IBM Research Division, Almaden Research Center, 1990.

[HP73] F. Harary and E. M. Palmer. *Graphical Enumeration.* Academic Press, 1973.

[HP88] W. Hasan and H. Pirahesh. Query rewrite optimization in starburst. Technical Report RJ 6367, IBM Research Division, Almaden Research Center, 1988.

[IK84] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, September 1984.

[IK90] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 312–321, 1990.

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

[IW87] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 9–22, 1987.

[Kan91] Y. C. Kang. *Randomized Algorithms for Query Optimization.* PhD thesis, University of Wisconsin-Madison, 1991. Technical report #1053.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. *Proceedings of the 12th International Conference on Very Large Databases, Kyoto*, pages 128–137, 1986.

[Knu68] D. E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, 1968. Second edition, 1973.

[KRB85] W. Kim, D. S. Reiner, and D. S. Batory, editors. *Query processing in database systems.* Springer, Berlin, 1985.

[LVZ93] R. S. G. Lanzelotte, P. Valduriez, and M. Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. *Proc. of the 19th VLDB Conference, Dublin, Ireland*, pages 493–504, 1993.

[McK93] W. J. McKenna. *Efficient Search in Extensible Database Query Optimization: The Volcano Optimizer Generator.* PhD thesis, University of Colorado, Boulder, 1993.

[Mor92] S. Morishita. Avoiding cartesian products in programs for multiple joins. *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 368–379, 1992.

[NSS86] S. Nahar, S. Sahni, and E. Shragowitz. Simulated annealing and combinatorial optimization. *23rd Design Automation Conference*, pages 293–299, 1986.

[NW78] A. Nijenhuis and H. S. Wilf. *Combinatorial algorithms.* Academic Press, New York, 2nd edition, 1978.

[OL90] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. *Proc. of the 16th VLDB Conference, Brisbane, Australia*, pages 314–325, 1990.

[Ozk86] E. Ozkarahan. *Database Machines and Database Management.* Prentice-Hall, 1986.

[PGLK97a] A. Pellenkoft, G.A. Galindo-Legaria, and M.L. Kersten. The complexity of transformation-based join enumeration. *Proceedings of the 23st International Conference on Very Large Databases, Athens*, pages 306–315, August 1997.

[PGLK97b] A. Pellenkoft, G.A. Galindo-Legaria, and M.L. Kersten. Duplicate-free generation of alternatives in transformation-based optimizers. *Proceedings of the fifth international conference on database systems for advanced applications, Melbourne, Australia*, pages 117 –123, April 1997.

[Rag90] P. Raghavan. Lecture notes on randomized algorithms. Technical Report RC 15340, IBM Research Division, T. J. Watson, 1990.

[RH77] F. Ruskey and T. C. Hu. Generating binary trees lexicographically. *SIAM journal of Computation*, 6(4):745–758, December 1977.

[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. *Proc. of the 1979 ACM-SIGMOD Conference on the Management of Data*, pages 23–34, 1979.

[SG88] A. N. Swami and A. Gupta. Optimization of large join queries. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 8–17, 1988.

[Sha86] L. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239–264, September 1986.

[SI92a] A. N. Swami and B. R. Iyer. A polynomial time algorithm for optimizing join queries. *Proc. of the 9th International Conference on Data Engineering, Vienna, Austria*, pages 345–354, 1992.

[SI92b] A. N. Swami and B. R. Iyer. A polynomial time algorithm for optimizing join queries. Technical Report RJ 8812, IBM Research Division, Almaden, 1992.

[Swa89a] A. N. Swami. *Optimization of Large Join Queries.* PhD thesis, Stanford University, 1989. Technical report STAN-CS-89-1262.

[Swa89b] A. N. Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 367–376, 1989.

[Swa91] A. N. Swami. Distribution of query plan costs for large join queries. Technical Report RJ 7908, IBM Research Division, Almaden, 1991.

[Tay90] Y. C. Tay. On the optimality of strategies for multiple joins. *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 124–131, 1990.

[Ull89a] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, New York, USA, 1989.

[Ull89b] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, New York, USA, 1989.

[Val92] P. Valduriez, editor. *Parallel Processing and Data Management.* Chapman and Hall, London, 1992.

[VF90] J. S. Vitter and Ph. Flajolet. Analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 9, pages 431–524. North Holland, 1990.

[vL90] J. van Leeuwen. Graph algorithms. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity. North Holland, 1990.

[VM96] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proceedings of the ACM SIGMOD Conf on Management of Data, Montreal*, pages 35–46, 1996.

[WM97] W.Scheufele and G. Moerkotte. On the complexity of generating optimal plans with cross products. In *Proceedings of the 23st International Conference on Very Large Databases, Athens*, 1997.