Cost distribution of search spaces in query optimization

C.A. Galindo Legaria, J. Pellenkoft, M.L. Kersten

Computer Science/Department of Algorithmics and Architecture

CS-R9432 1994

Cost Distributions of Search Spaces in Query Optimization[*]

César Galindo-Legaria[†] Arjan Pellenkoft Martin Kersten

CWI
P. O. Box 94079, 1090 GB Amsterdam, The Netherlands
{cesar,arjan,mk}@cwi.nl

Abstract

Query optimization algorithms explore a large space of query execution plans looking for an optimal solution. The predominant algorithms move around the search space in either a deterministic or probabilistic way. The performance of probabilistic optimization algorithms is strongly influenced by the cost distribution over the search space, the connectivity of the space and the overhead cost.

The *transformation free* (TF) Query Optimization strategy proposed in this paper is insensitive to the search space connectivity. We show that under weak conditions and the assumption of an accurate cost model the performance and accuracy of the TF strategy in comparable with probabilistic strategies.

CR Subject Classification (1991): [H.2.4] Database Systems, Query Processing; [H.3.3] Information Search and Retrieval; [G.2.2] Graph Theory; [F.2.2] Non-numerical algorithms and problems.
Keywords and Phrases: Randomized query optimization, transformation-based optimization, transformation-free optimization, random generation of query evaluation plans, counting of query evaluation plans.

1 Introduction

Query optimization is a combinatorial search problem, because for a given query one must find the 'best' Query Execution Plan (QEP) out of many semantically equivalent alternatives. For an optimizer the "best" QEP would be the one with the lowest cost, found so far.

For a given query the number of alternative QEPs to consider depends, among other things, on the number of relations and the number of available join algorithms. The number of alternatives grows fast as the number of relations in a query increases.

Due to the large number of alternatives, it becomes impossible to use an exhaustive search strategy when queries are no longer of trivial size. Instead non-exhaustive strategies are often used, which are either deterministic or probabilistic. They aim for a "good" but not necessarily optimal solution. This paper focuses on the probabilistic search strategies.

[*]This work has been supported in part by ESPRIT-III project 7091, " Pythagoras".
[†]C. Galindo-Legaria was supported by an ERCIM fellowship.

Several probabilistic search strategies have been proposed, like : *Simulated Annealing* (SA), *Iterative Improvement* (II) and others [IW87, IK90, IK91, SG88, LVZ93, OL90, INSS92, GD87]. These search strategies are probabilistic in the sense that they start at a randomly selected QEP and/or the generation of the next QEP involves some probability. To generate the next QEP transformation rules are used. These rules are based on properties of the underlying algebra, such commutativity and associativity. The QEPs that can be generated using a single transformation are called the *neighbors* of the current QEP.

The performance of these transformation based optimization algorithms depends on the cost distribution over the search space and its topology [IK90]. Since the set of used transformation rules defines which QEPs are neighbors, a topology is imposed on the search space.

The probabilistic strategy we propose does not move around in the search space following transformations, but randomly chooses a QEP out of all alternatives. This means that the QEP chosen does not depend on the previous QEP. The probability of a QEP being chosen is equal for all QEPs. In this way the *transformation free* (TF) algorithm is not sensitive to the topology of the search space, also See [GLPK94b].

The simple overall architecture of the TF algorithm makes it possible to focus on the cost estimation of a QEP and the random generation of QEPs. In Section 5 the essentials of the algorithm are explained. Also see [GLPK94a].

The results of the experiments presented in this paper provide a quantitative basis for comparison of probabilistic models. We also give a comparison of the performance of II, SA and TF.

In Section 2 we give the definitions we use and describe three optimization algorithms. Section 3 describes details of the experiments, and Section 4 the experiments showing the cost distributions over small search spaces. The counting and random generation of valid trees for acyclic query graphs is described in Section 5 and in Section 6 random sampling techniques are validated. In Section 7 the direct comparison between the tree optimization algorithms is given and our conclusions are given in Section 8.

2 Probabilistic search algorithms

2.1 Definitions

Query graph. We represent a query by means of a *query graph.* Nodes of such 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, 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 an edges

Query evaluation plans. (*QEPs*) are used to evaluate queries, instead of the straight forward definition of product followed by selection given above. A QEP is an operator tree whose inner nodes are labeled by a join operator and whose leaves are labeled by relations. The *result* of a QEP is computed bottom-up in the usual way. QEPs often include annotations on the join-algorithm to use —e. g. nested loops, hash, merge, etc.— when several are available.

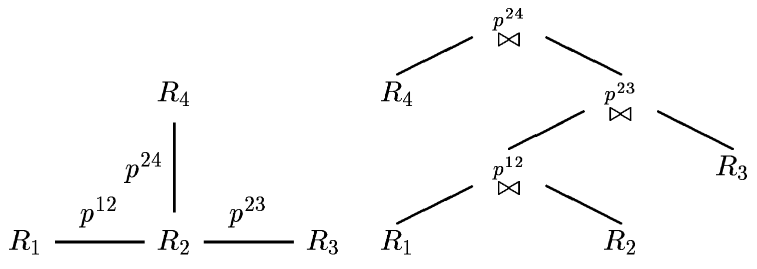


Figure 1: A query graph and one of its valid QEPs.

Note that not every binary tree on the relations of the query is an appropriate QEP, because some may require the use of Cartesian products. Those that do not require products are called *valid* [OL90]. The topology of valid plans is given by *association trees.*

Association tree. Given a connected query graph G, an unordered binary tree T is called an association tree of G, when it satisfies the following recursive definition : The leaves of T correspond one-to-one with the nodes of G, and every subtree of T is an association tree of a connected subgraph of G.

Topology of QEPs. Based on the topology of the QEPs two types can be distinguished; *linear* QEPs and *bushy* QEPs. A linear QEP has at least one base relation as input and a bushy QEP can also have intermediate results as input. Note that a bushy tree can also be linear.

The two representations of queries — QEPs 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 QEP specifies unambiguously the inputs to each operator, and how it can be evaluated. See Figure 1 for a query graph example and one of its valid (linear) QEPs.

Search space. The set of all QEPs from which the optimizer can choose a plan is called the search space. In transformation based optimization algorithms the search space is mostly referred to as a graph G = (V,E) with nodes V and edges E. The nodes represent QEPs and the edges represent transformations between QEPs.

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

The size of the search space can be limited by imposing rules on the type or shape of the QEPs. For instance, by only considering valid QEPs. Such a smaller search space can be beneficial to the optimization process.

A common way of reducing the search space is by considering only the valid QEPs. This search space reduction is based on the assumption that QEPs which incorporate a Cartesian Product are not likely to result in a low cost plan [SAC+79].

Sometimes the search space is reduced even further by only considering linear QEPs, but studies presented by [IK91] suggest that the space of bushy QEPs has a higher percentage of good plans. In our experiments we therefor consider the search space of valid bushy QPEs.

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 we only use hash-join.[1] Only those transformations that lead to a valid QEP can actually be used 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:	$A \bowtie (B \bowtie C) \leftrightarrow B \bowtie (A \bowtie C)$

Lemma 1. *Consider the search space of bushy QEPs with n join operators and the transformation rules above. Then, every QEP has $2 * n - 1$ neighbors.*

> **Proof sketch.** For a given QEP with n join operators we can apply n times the commutativity rule which results into n valid trees. To apply the three other rules we need an internal node which has a parent, so every join operator except the root. In total $n - 1$ potential transformations are possible for each of the three rules. But given a node and its parent only one of the rules will result in a valid plan. Therefor, the number of valid-neighbors for this set of rules is $2 * n - 1$.

2.2 Counting valid QEPs — the easy cases

For a given query graph the number of valid QEPs 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 plans can be computed easily, because they have a "regular" structure. Fortunately the upper and lower bound of number of valid plans for a query graph on n relations is given by query graphs for which the number of valid plans can be computed easily. The lower bound is set by a *string* graph and the upper bound is set by a *completely connected* graph.

In the following description of query graph topologies, n denotes the number of relations participating in the query. The formulas give the number of valid *unordered* trees — trees which do not distinguish the left and right input.

If left and right are distinguished, the total number of valid *ordered* trees can be computed by multiplying the number of valid unordered trees by 2^{n-1}. If there are several possibilities for each join operator -i.e. hash-join, nested-loop or merge-scan-, then the total number of valid trees increases even further.

Suppose we have a query graph in which n relations participate, and there are m implementations for a join available. If for this query graphs there are T valid (un)ordered trees, then the total number of valid trees is : $T * m^{n-1}$.

[1]We did experiments with *nested-loop* as well, and the results are similar to those of hash-join.

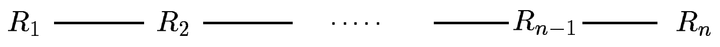


Figure 2: A string query

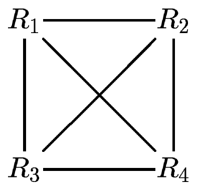


Figure 3: A completely connected query on 4 nodes

String Query. 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. The number of unordered valid QEPs is:

$$\frac{(2n-2)!}{n!(n-1)!}$$

Completely connected Query. The query graph for a completely connected query of n relations, has n nodes of degree *n-1*. See Figure 3 for a completely connected graph on 4 nodes. The number of unordered valid QEPs is:

$$\frac{(2n-2)!}{(n-1)!}$$

The bounds for *acyclic* query graphs are given by the number of possible alternatives of the string query and the *star* query.

Star Query. The query graph of a star query of n relations, has *n-1* nodes with degree 1 and one center node with degree *n-1*. See Figure 4 for a star on 5 relations. The number of unordered valid QEPs is:

$$(n-1)!$$

2.3 Optimization algorithms

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

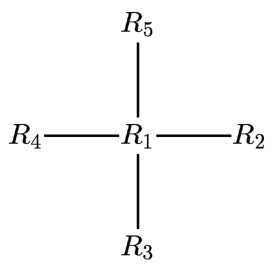


Figure 4: A star query

[NSS86] shows 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 5 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 5: Iterative Improvement

In our experiments we compared the quality of algorithms after exploring a fixed number of plans. For the II algorithm this fixed number of plans was used as stopping condition.

To obtain a random starting point for a local optimization we used our algorithm that generates plans at random in a uniform way, (See section 5.2).

Instead of exhaustively searching all neighbors of a plan to detect whether or not it is a local minimum we used the definition of a *r-local minimum* [Kan91]. This method classifies a plan as local minimum if none of the r randomly selected (with repetition) neighbors has a lower cost. With r being equal to the number of neighbors of the plan. Note that since the plans are selected at random, and repetitions are therefor possible, a r-local minimum is not guaranteed to test all neighbors.

Simulated Annealing (SA) starts at a random QEP and randomly generates moves to next QEPs. If the next QEP is an improvement the move is accepted, if the move leads to a QEP 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 QEP with the lowest cost. It can be shown that the algorithm converges to the global minimum as temperature approaches zero.

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 6 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 6: Simulated Annealing

Like in II the Simulated Annealing algorithm starts at a random state. In our implementation we used our algorithm that generates plans at random in a uniform way, see Section 5.2. For SA specific parameters we used the parameters given by [Kan91].

parameter	value
initial temperature T_0	$2*$ cost of initial plan
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 plans explored. The *equilibrium* condition means that the inner loop finishes if n plans have been explored, with $n = 16 * Number\ of\ joins\ in\ the\ tree$

Transformation Free (TF) generates random QEPs, with replacement, and keeps track of the one with the lowest cost. The algorithm terminates after it has visited n QEPs or when the cost of the best QEP 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 shows the pseudo-code of the TF algorithm.

In our experiments we used the number of plans explored as stopping condition.

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

Figure 7: Transformation Free

2.4 SA and II (motivation)

The probabilistic search algorithms SA, II, and their variations rely heavily on transformation rules to generate candidate execution plans. 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 a starting plan into any other plan in the space— does not guarantee good behavior, 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 behavior of transformation-based algorithms is somewhat limited —rather, they serve to provide qualitative insight [IK91]. A question that motivates the present work is the following:

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

For TF to work we need to know the cost distribution and we must be able to generate QEPs at random in a uniform way. In section 4.1 we show the results of our exploration of the cost distribution over the search space.

Similar experiments were done by [IK91], but they only tested star queries. They showed that, if the variance of the catalog increases (the size difference of the relations increases) the distribution shifts to the left and the range of scaled cost increases. This can be interpreted as that a substantial part of the search space consists of "good" QEPs. In our experiments we extended the range of queries to acyclic query graph.

In Section 5 we discuss several ways to generate plans at random and show how uniformity can be guaranteed.

3 Experimental setup

This chapter discusses the conditions in which the experiments were performed. It describes the cost model, schema, catalogs, queries and our measure for *good* plans.

3.1 Cost model

For all our experiments we used the cost function provided by the Analytical Performance Evaluator which models DBS3 [ACV91]. The initial set of experiments were conducted

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 8: Relations and their attributes

using nested-loop joins. Since this is a conservative implementation of a join operator we changed it to hash-join. The observations in the sequel also hold for nested-loop.

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], and is given below:

$$(\{R\} + \{S\}) * hash + \{R\} * move + \{S\} * comp * F$$

With {R} and {S} denoting the sizes of the two inputs and *hash*, *move*, *comp* and *F* constants. It is assumed that {R} < {S} so the cost of building the hash table depend on the size of the smallest relation.

3.2 Schema, catalogs and queries

Schema The database schema considered is the Portfolio Club Experimental Model (PEM) [ACV91]. It was designed to provide a realistic experimental base for complex query definition, evaluation and benchmarking in the EDS project. The PEM schema contains several relations join-able through foreign keys, See Figure 8 for the database schema.

Catalogs The experiments were done for three different catalogs. They differed in the size of their relations, the sizes are given in Figure 9. The catalogs have been chosen such that we have a catalog with low, middle and high variance in their relation sizes.

	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 9: sizes of the relations

Queries. The experiments have been performed for queries of 4 up to 12 relations on the three catalogs. For the queries of 4 to 8 relations it is possible to enumerate all valid QEPs, which made it possible to compute the exact cost distribution. For the queries of 8 to 12 relations the cost distribution have been computed using sampling.

The cost distribution for the query of 8 relations was completely known, so we could compare this with the cost distribution obtained by sampling to validate the sampling techniques, (See Section 6).

The queries of 8 to 12 relations were also used in comparing the performance of the three optimization algorithms. Figure 10 shows the query graphs for the used queries. The topology of the query graphs is neither a star nor a string, so the uniform random generation of plans had to be closely examined.

3.3 Cost metrics

The purpose of the first experiments are to investigate the distribution of cost over the search space, and especially the ratio of *good* QEPs. Like [Swa89b] we classify queries as *good, acceptable* and *bad* according the following criteria:

good	$cost(p) \leq 2 * cost(cheapestQep)$
acceptable	$2 * cost(cheapestQep) < cost(p) \leq 10 * (cheapestQep)$
bad	$10 * cost(cheapestQep) < cost(Qep)$

The number of plans explored was used for comparison of the optimization algorithms. The advantage of this metric over using time is that the results are less implementation and system dependent. The cost estimation of a plan is assumed to be a major cost factor in an optimization algorithm, so the number of plans for which the cost are computed is a good measure for how much effort an algorithm has put into finding a good plan.

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

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

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

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

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

Figure 10: Queries used in the experiments

4 Cost distribution of small spaces

This section describes our experiments to explore the complete search space of small queries. It describes the algorithm to generate all plans and shows the results obtained.

Exhaustive generation of QEPs. Remember we consider connected queries with acyclic query graphs. Removing an edge form such a graph disconnects it, leaving two connected graphs. This makes it possible to enumerate the set of all valid QEPs efficiently, by recursively splitting a query graph G as follows.

If the graph has one node, then the only QEP is the relation that labels such node; otherwise remove an edge, say labeled p, to disconnect the graph, then find QEPs Q_1, Q_2 for the two connected graphs that remain, and finally return $(Q_1 \overset{p}{\bowtie} Q_2)$ as a QEP for G.

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

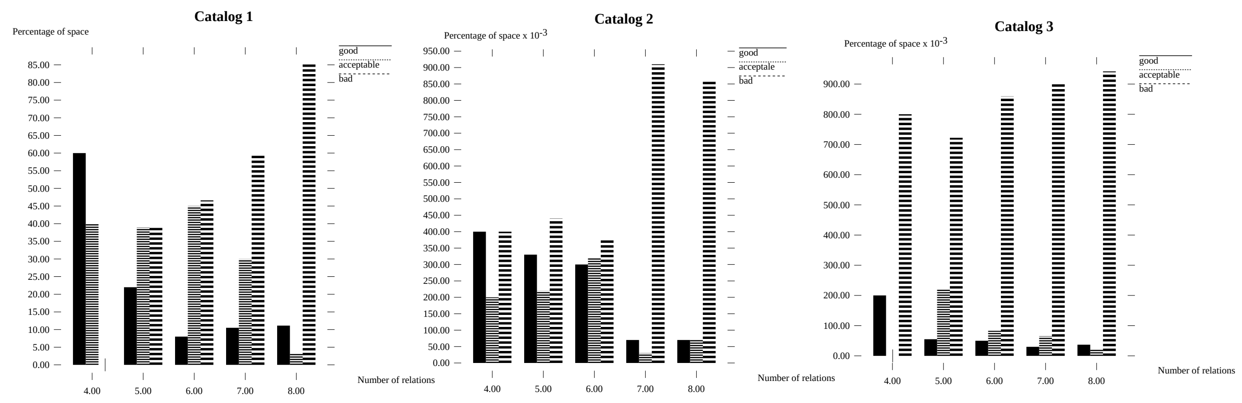


Figure 11: Histograms and probability of finding a good plan in a random sample.

4.1 Results of experiments

Figure 11 shows the cost distribution of valid QEPs for queries having 4 to 8 relations for the three different catalogs. For all spaces explored exhaustively the number of good plans is sufficiently large that a run of several tens of randomly selected plans will hit a good one with high probability.

One can observe that as the number of relations increases the number of good plans decreases. This coincides with observations made by [Swa89a].

The queries of 6 and 7 relation have fewer good plans than query 8 under catalog 1, so the shape of the query seems also to have an impact on the cost. Also the type of catalog has an effect on the cost distribution, catalog 3 (with many big relations) has relatively few good plans compared to catalog 1 and 2.

Using standard statistical techniques the number of random plans needed to hit a good one, with a certain probability, can be computed.

Lemma 2. *Given the ratio of good plans, P_{good}, and the required probability of hitting a good plan ,P_{req}, the number of plans n that must be explored is :*

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

.

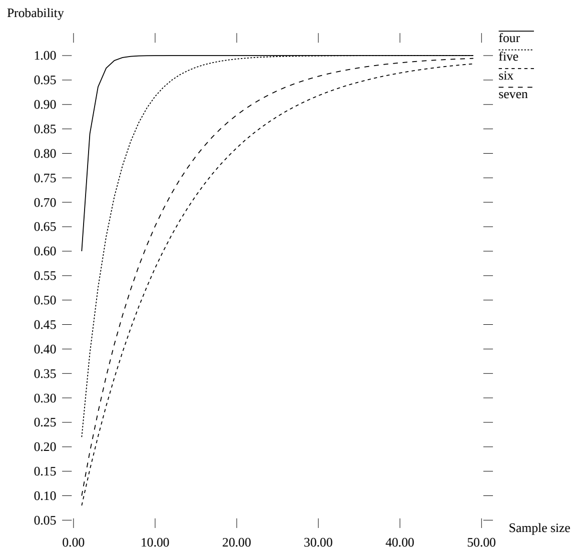


Figure 12: Histograms and probability of finding a good plan in a random sample.

Proof sketch. Given that the ratio of good plans is P_{good}, the chance of selecting a *wrong* plan is $1 - P_{good}$. The chance of selecting a sequence of n *wrong* plans is therefor $(1 - P_{good})^n$.
Since the probability of selecting a *good* plan has to be P_{req}, the probability of selecting a *wrong* plan must be $1 - P_{req}$.
Given the two expressions that tell the probability of selecting a *wrong* plan 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 12 shows the probability of hitting a good plan with 95% probability for several ratios of good plans at increasing sample size.

5 Random generation of QEPs

For small spaces we know the cost distributions, but we are also interested in the cost distributions over bigger spaces. To obtain the cost distributions for bigger spaces we use sampling, which implies the random generation of plans.

We can easily generalize the cost distribution of a sample if the generation of QEPs is done with a uniform distribution. The ratio of good, acceptable and bad plans of the sample then approaches the ratios of the whole space. For sampling search spaces we considered two kinds of sampling techniques, *quasi*-random and *uniform*-random.

5.1 Quasi random generation of QEPs

The following two procedures generate *quasi*-random QEPs. They are easy to implement, but they either do not guarantee uniformity, or take a very long time.

first choice	second choice	plan	probability of generation
(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 13: Generation of quasi random QEPs by edge selection.

Random-edge selection. This procedure is similar to the technique used in Section 4 to generate plans exhaustively. Here, instead of considering all possible graph splittings, we split the graphs by randomly selecting an edge at each step, which results in a random plan.

Random-walk. This procedure is based on transformation rules to move from one valid plan to another. If we start at some plan 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, [Rag90]. In particular, if all nodes in a graph have equal degree, as is the case here [Kan91], then we are equally likely to be at any node of the graph after n steps, for a sufficiently large n, 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 plan efficiently. Instead, we consider all plans visited in a random walk as a random sample of the space.

To see why the random-edge method does not produce equi-probable QEPs, 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 plan is already completely specified —remember 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 13 shows the probability of generating each plan. In principle, it seems that the procedure can be modified to use *weighted* instead of uniform selection at each step, so that the resulting QEPs are all equi-probable. But computation of the appropriate weights is difficult, and we have not found a way to do it efficiently, yet.

5.2 Uniformly distributed random QEPs

We want to be able to generate valid plans for a given query graph, such that all plans have equal probability of being generated. For small queries a simple, and in-efficient, brute-force/filter method is feasible, but for bigger queries a more elaborate method has to be used.

In-efficient generation. 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* plans we need to check if the resulting plan is either valid or

Number of relations	All plans	Valid plans	Fraction
5	40320	576	0.014
10	6.4x10^{15}	1.3x10^{11}	2.1x10^{-5}
15	3.0x10^{29}	7.6x10^{21}	2.5x10^{-8}
20	5.2x10^{44}	1.5x10^{34}	2.8x10^{-11}

Figure 14: Fraction of valid plans

invalid. If it is an invalid plan it must be discarded, and an other plan must be generated until a valid plan is detected.

The efficiency of this method depends on both the fraction of valid plans over the total number of plans and on how fast the type (valid or unvalid) of a plan can be detected. Using the definition of association trees in Section 2 we can come up with an efficient recursive algorithm which labels a tree as valid or invalid.

The fraction of valid plans over the total number of plans decreases fast as the number of relations, n, participating in the query increases. For example, if we compare the total number of valid and invalid plans to the maximum number of valid plans for acyclic queries we can compute the fraction for the *best* case.

The total number of valid and invalid plans is given by the formula of the completely connected query graph : $\frac{(2n-2)!}{(n-1)!}$, and the maximum number of valid plans for acyclic query graphs is given by the formula of the star query : $(n-1)!$. Dividing these results in : $\frac{(n-1)!^2}{(2n-2)!}$. In Table 14 the fraction of valid plans is computed for several n. From the formula and the table we can see that we have to generate many plans before we will generate a valid one. This method is therefore only feasible for very small queries.

Efficient generation. Efficient generation of uniformly distributed plans is not only needed for speeding up the sampling, but it is also essential for the Transformation Free optimization algorithm. The efficient generation of valid plans (association trees) described in the following paragraphs is based on counting the number of association trees for a given *acyclic* query graph, also See [GLPK94a] for details on algorithms.

For computing the number of association trees we consider the level numbers of the relations in a plan, and especially the number of plans in which a relation v is at some level k. This set of trees is denoted as $\mathcal{T}_G^{v(k)}$. The set of all trees for the graph is denoted $\mathcal{T}_G$.

With each node a *level-sequence* is associated. The level-sequence, $a_0, a_1, ..., a_n$ denotes the number of plans in which the node is at a certain level, with a_i denoting the number of trees in which the node is at level i.

For example, if a node has the sequence $0, 2, 2, 1$ this means that there are 0 trees in which the node is at level 0, 2 trees in which the node is at level 1, 2 trees in which the node is at level 2 and one tree in which the node is as level 3, and there are no trees in which the node is at a deeper level.

Some elementary observations are the following:

- If the graph has only one node, then there is exactly one association tree, and v is at level 0; that is, $|\mathcal{T}_G| = \left|\mathcal{T}_G^{v(0)}\right| = 1$, for $n = 1$.
- If the graph has more than one node, then there is no association tree in which v is at level 0; that is, $\left|\mathcal{T}_G^{v(0)}\right| = 0$, for $n > 1$.
- There is no association tree in which v is at level greater than or equal to n; that is, $\left|\mathcal{T}_G^{v(i)}\right| = 0$, for $i \geq n$.
- Since v appears at some unique level in any association tree of G, the total number of association trees is
$$|\mathcal{T}_G| = \sum_i \left|\mathcal{T}_G^{v(i)}\right| \quad .$$

The previous observations provide base cases for the computation of $\left|\mathcal{T}_G^{v(i)}\right|$, and establish their relation with $|\mathcal{T}_G|$.

To compute the number of association trees for a given query graph we use two recurrence equations. The first one *extends* a subgraph by adding one more node, and the second one takes the *union* of two subgraphs whose intersection is a single node. The lemmas and proofs given will be followed by an example.

For convenience, we sometimes represent trees using *lists anchored on a leaf* w, which contain the subtrees found when traversing the tree from the root to w. For example, the lists anchored on w of the trees in Figure 15 are (T_1, T_2), (v, T_1, T_2), (T_1, v, T_2), and (T_1, T_2, v), respectively.

Intuition for extending a subgraph. Assume a query graph G is given and also the set of valid trees $\mathcal{T}_G$ for this graph. If G is extended by a node v which shares an edge with node w, already in the graph, we can construct the set of trees, $\mathcal{T}_{G^{ext}}$, for the extended graph G^{ext}, based on the set of trees $\mathcal{T}_G$ as follows.

In every tree $t \in \mathcal{T}_G$ one leave is labeled w. From this tree we can construct several valid trees for the extended query graph G_{ext}, by "grafting" the new relation v somewhere in the path from the root to leave w. If we write this path as a list anchored on w, i.e. $(T_1, T_2, ..., T_n)$ the valid plans for the extended graph constructed from this list are: $(T_1, T_2, ..., T_n, v)$, $(T_1, T_2, ..., v, T_n)$ ... $(v, T_1, T_2, ..., T_n)$. the level at which v ends up is then $level(w) + 1$, $level(w)$, ... , 1, respectively

From this example we can see that the trees in which v ends up at level l are constructed from trees in which w was at level $l-1, l, l+1, ...$ which leads to the formula of lemma 1.

Lemma 1. *Let $G = (V, E)$ be an acyclic query graph. Let v be a node in V such that $G' = G|_{V-\{v\}}$ is connected, and let $(v, w) \in E$. Then*
$$\left|\mathcal{T}_G^{v(k)}\right| = \sum_{i \geq k-1} \left|\mathcal{T}_{G'}^{w(i)}\right| \quad .$$

Proof sketch. There is a one-to-one mapping between trees in $\mathcal{T}_G$ and pairs of the form (T', k), where $T' = (T_1, \ldots, T_i)$ is the list anchored on w of a tree in $\mathcal{T}_{G'}$, and

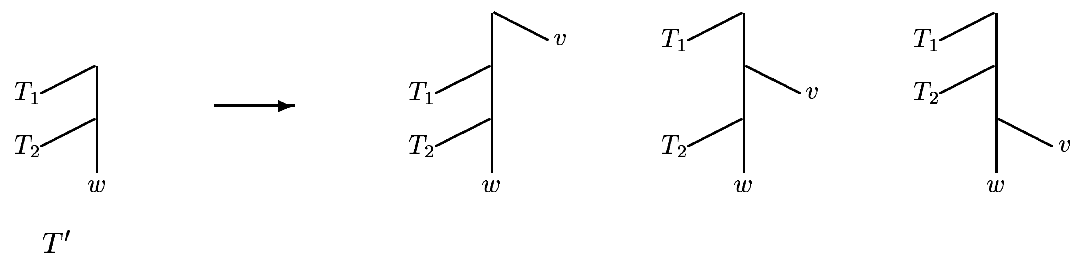


Figure 15: Adding a new leaf to a tree.

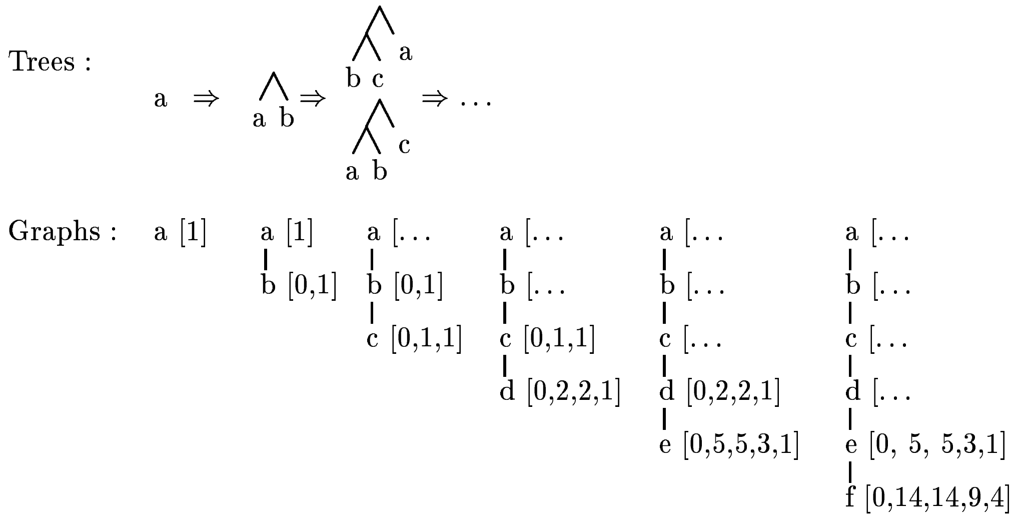

Trees :
a ⇒ a b ⇒ b c a / a b c ⇒ ...
Graphs :
a [1]
a [1] b [0,1]
a [... b [0,1] c [0,1,1]
a [... b [... c [0,1,1] d [0,2,2,1]
a [... b [... c [... d [0,2,2,1] e [0,5,5,3,1]
a [... b [... c [... d [... e [0, 5, 5,3,1] f [0,14,14,9,4]

Figure 16: Extending query graphs by one node

$1 \leq k \leq i+1$. To obtain the tree $T \in \mathcal{T}_G$ corresponding to such a pair, insert a new leaf v in position k of the anchored list of T'. Figure 15 shows an example of T', and the trees obtained from pairs $(T',1)$, $(T',2)$, $(T',3)$. The equation follows from the fact that the tree obtained from a pair (T',k) is in $\mathcal{T}_G^{v(k)}$, and the length of the list of T' is at least $k-1$. ■

Example of extending a subgraph. A base case is a query graph $G_1 = (V_1, E_1)$ with $V_1 = \{a\}$, $E_1 = \{\}$. There is only one plan possible and a is at level 0 in this plan (see definition). Extending Q_1 with node b results in query graph $G_2 = (V_2, E_2)$ with $V_2 = \{a, b\}$ and $E_2 = \{(a, b)\}$. Still there is only one plan possible and b can only be at level 1 in this plan. Extending G_2 with node c results in query graph $G_3 = (V_3, E_3)$ with $V_3 = \{a, b, c\}$ and $E_3 = \{(a, b), (b, c)\}$. Now there are two valid plans, in one plan c is at level 1 and in the other plan c is at level 2. In figure 16 this process is depicted for up to six nodes and for each relevant node the level-sequence is shown.

Intuition for the union of two subgraphs Assume that a query graph G is the result of the union of two disjunct query graphs; G_1 and G_2. These two subgraphs have exactly one node v in common. The union of two trees t_1 and t_2 (a valid plan of G_1 and a valid plan of G_2) results in a valid plan for G. The merging of the two trees is done as follows.

The two paths from the root to the leave v are the anchored lists; $l_1 = (T_1^1, T_2^1, \cdots, T_n^1)$ for t_1 and $l_2 = (T_1^2, T_2^2, \cdots, T_m^2)$ for t_2. If the two paths are merged such that the order two list are maintained the resulting tree is valid for Q. Since the paths have length n and m node v will be at level $n+m$ in the resulting tree, so in the sequence-list of node v, in the resulting query graph, slot a_{m+n} will be increased by the number of possible mixings of these two trees. This slot will also be increased by other combinations of trees whose path lengths add up to $m+n$. So given the two level-sequences of node v for the two subgraphs we can compute the level-sequence for node v for the resulting query graph.

Lemma 2. *Let $G = (V, E)$ be an acyclic query graph. Let V_1, V_2 be sets of nodes such that $V = V_1 \cup V_2$, $V_1 \cap V_2 = \{v\}$, and the graphs $G_1 = G|_{V_1}$ and $G_2 = G|_{V_2}$ are connected. 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 sketch. There is a one-to-one mapping between trees in $\mathcal{T}_G^{v(k)}$ and triplets of the form (T_1, T_2, c), where $T_1 = (T_1^1, \ldots, T_1^i)$ is the list anchored on v of a tree in $\mathcal{T}_{G_1}^{v(i)}$; $T_2 = (T_2^1, \ldots, T_2^{k-i})$ is the list anchored on v of a tree in $\mathcal{T}_{G_2}^{v(k-i)}$; and $c = (\alpha_1, \ldots, \alpha_{k-i+1})$ is a *composition of i in $k-i+1$* [NW78]. To obtain the tree $T \in \mathcal{T}_G^{v(k)}$ corresponding to such a triplet, merge the lists for T_1 and T_2 as specified by c —each α_j indicates how many subtrees of the list of T_1 go between subtrees T_2^{j-1} and T_2^j in the merged list. For example, Figure 17 shows how to find the T of a triplet $((T_1^1, T_1^2), (T_2^1, T_2^2), (1,1,0))$. The equation results from the fact that there are $\binom{k}{i}$ compositions of i in $k-i+1$. ■

Example of the union of two subgraphs Figure 18 shows how the above lemmas are used to compute the number of association trees for a graph $Qg = (V, E)$ of five nodes; $V = \{a, b, c, d, e\}$ and $E = \{(a,b), (b,c), (c,d), (c,e)\}$. The graph is neither a star nor a chain. Each column of the table shows the data for a subgraph of G, and it is computed based on the values of previous columns. The bottom row shows the total number of trees for the subgraph.

For example, since $Qg|_{\{abcde\}}$ extends $Qg|_{\{abcd\}}$ by one node, lemma 1 is used to compute the entries of the last column. The new node e is connected to old node c, so we need to know the number of trees in $Qg|_{\{abcd\}}$ in which c appears at various levels. This information is available in the second-to-last column, whose values were in turn computed using lemma 2 on the result of earlier columns.

The following theorems follow from the application of lemmas 1 and 2. The first theorem refers to the time required to compute a matrix similar to that of Figure 18.

Theorem 1. *The number of association trees for a given acyclic query graph G on n relations can be computed in polynomial time.*[2]

[2]A loose upper bound on theorems 1 and 2, and the corollary of theorem 2 is $O(n^3)$.

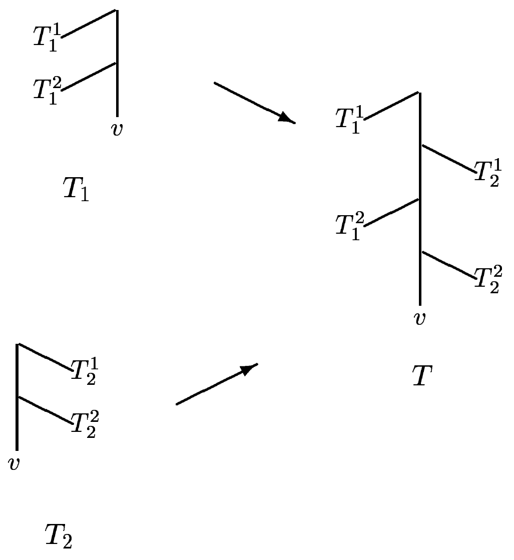


Figure 17: Merging two trees that share a leaf.

$Qg\|_{\{a\}}$	$Qg\|_{\{ab\}}$	$Qg\|_{\{abc\}}$	$Qg\|_{\{d\}}$	$Qg\|_{\{cd\}}$	$Qg\|_{\{abcd\}}$	$Qg\|_{\{abcde\}}$
$\|\mathcal{T}_G^{a(0)}\| = 1$	$\|\mathcal{T}_G^{b(1)}\| = 1$	$\|\mathcal{T}_G^{c(1)}\| = 1$ $\|\mathcal{T}_G^{c(2)}\| = 1$	$\|\mathcal{T}_G^{d(0)}\| = 1$	$\|\mathcal{T}_G^{c(1)}\| = 1$	$\|\mathcal{T}_G^{c(2)}\| = 2$ $\|\mathcal{T}_G^{c(3)}\| = 3$	$\|\mathcal{T}_G^{e(1)}\| = 5$ $\|\mathcal{T}_G^{e(2)}\| = 5$ $\|\mathcal{T}_G^{e(3)}\| = 5$ $\|\mathcal{T}_G^{e(4)}\| = 3$
$\|\mathcal{T}_G\| = 1$	$\|\mathcal{T}_G\| = 1$	$\|\mathcal{T}_G\| = 2$	$\|\mathcal{T}_G\| = 1$	$\|\mathcal{T}_G\| = 1$	$\|\mathcal{T}_G\| = 5$	$\|\mathcal{T}_G\| = 18$

In each column, G is the subgraph of Qg specified in the top row.

Figure 18: Counting the number of association trees.

The second theorem is based on a natural numbering, or *ranking* of all trees of a given graph, extracted from the matrix used in the computation of the number of trees. For example, from the last column of the matrix in Figure 18, we assign the numbers 1 through 5 to the trees of Qg in which e appears at level 1; numbers 6 through 10 to those in which e is at level 2; 11 through 15 to those in which e is at 3; and finally 16 through 18 to those trees in which e is at level 4. Our unranking procedure is based on those presented in [RH77, Li86].

Theorem 2. *Association trees of a given acyclic query graph G on n relations can be unranked in polynomial time.*

Since trees are numbered, and we can reconstruct efficiently any of them given its number, the next corollary follows.

Corollary of theorem 2. *Assuming a source of random numbers is available, association trees of a given acyclic query graph G on n relations can be generated at random with uniform distribution in polynomial time.*

5.3 Unranking

When counting the number of valid trees for a query graph an arbitrary node in the query graph is considered to be the root. For this root the level-sequence is computed based on the level-sequences of its children using the two lemmas of the previous Section. The sum of the sequence levels of the root of the query graph gives the total number of valid trees for the graph, N. Using a "reversed" counting procedure we can map every number form 1 to N to a unique tree. Like counting the number of trees, the unranking is also split into two sections; a node has exactly *one child* (counterpart of extending a query graph by one node) and a node has more than one child (counterpart of merging to subgraphs). In the sequel we give a short description of how the unranking could be performed for the two cases.

One child Given a node r, the root, with level-sequence $a_1^r, a_2^r, a_3^r, ..., a_n^r$ and a number K being the number to compute the corresponding tree for. $K <= N$, with N the total number of valid plans. Node r has exactly one child, node c, with level-sequence $a_1^c, a_2^c, a_3^c, ..., a_m^c$. We can find the *slot* j for which holds: $a_1^r + ... + a_{j-1}^r < K <= a_1^r + ... + a_j^r$, and we can also compute the $offset$ within that slot, $K - a_1^r + ... + a_{j-1}^r$.

Since slot j corresponds to a tree in which node r is at level j we know where to insert node r in the tree that will be computed for the child of r. If node r must be inserted at level j the child of r must be at level $j-1, j, j+1 \cdots$. This means that all trees in which child c is at level $1, \cdots, j-2$ can not be used to create a tree in which node r ends up at the right level.

To unrank the correct tree for c we have to *discard* the trees that are not useful. The rank of the tree for c is computed by adding the offset within slot j to the sum of the level-sequence of c, which provide the non-useful trees, $N_c = \sum(a_1^c + \cdots + a_{j-2}^c) + offset$. Now c is considered to be the root and this process is repeated until the graphs are reduced to base cases.

More children Assume again that there is a root node r with its level-sequence, $a_1^r, \cdots, a_k^r$ and we have to compute the tree that has rank K. But now node r has more than one child.

During the computation of the level-sequence of node r an order was set for the children of node r, say $c_1, c_2, \cdots c_n$. The level-sequence was computed by first computing a temporary level-sequence by merging c_1 and c_2. Then the temporary level-sequence is merged with c_3 to obtain a new temporary level-sequence. This is continued unit there are no more children left.

Unranking a node with more than one child is then solved as repeatedly solving the unranking of a node with two children. The level-sequence of the first child is $a_1^{c1}, \cdots, a_n^{c1}$, and the level-sequence of the second child is $a_1^{c2}, \cdots, a_m^{c2}$.

Again we start by determining in which slot K falls, say slot j. A tree which has node r at level j can be the result of many combinations of two trees, in which r is at level j_1 and j_2 respectively if $j_1 + j_2 = j$. One such combination of two trees can be merged in several ways which are all valid. So these two trees produce $a_{j_1}^{c1} * a_{j_2}^{c2} * mixings(j_1, j_2)$ trees in which node r is at level j. The offset within slot j determines which j_1, j_2 are used and their *mixing*.

Given the two ways of unranking a tree the whole unranking can be solved by recursively unranking nodes. Note that the order in which the level-sequences are computes must also be used to unrank the trees.

6 Sampling spaces

We now compare samples obtained from a known space by our quasi-random and uniformly-random procedures. 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 plans having a cost less or equal to c. Formally, in a space S the *accumulated cost distribution* is

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

6.1 Accuracy of sampling methods

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 4, 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 19 shows the correlation coefficients found for a query of eight relations, for increasing sample sizes.

The random-edge sampling method does not approach the exact cost distribution, because it favors certain plans. Since we do not know the quality of the plans 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 uniformly-random method converges faster to the known cost distribution.

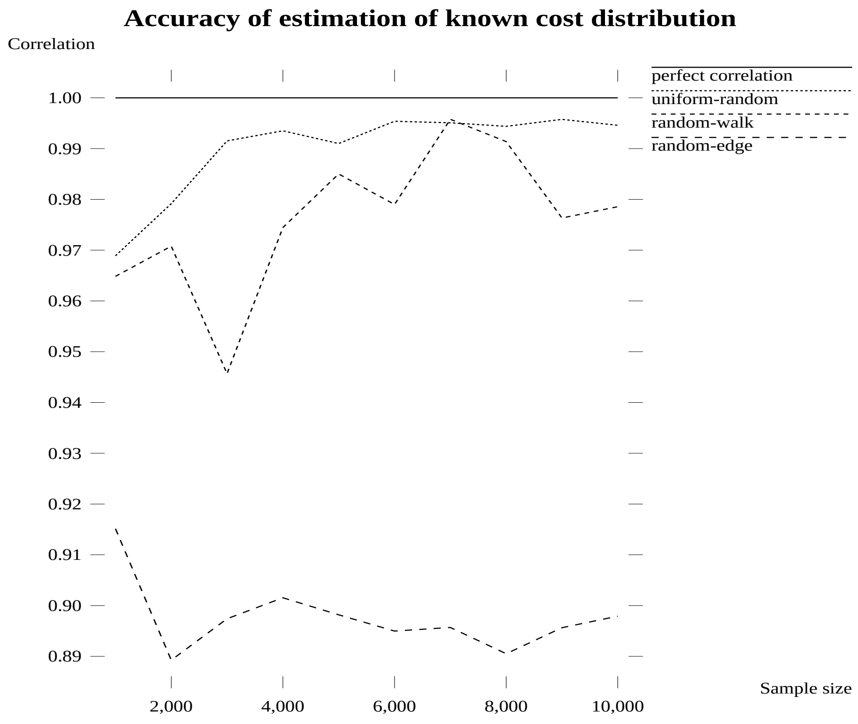


Figure 19: Correlation of approximations to cost distribution using random sampling.

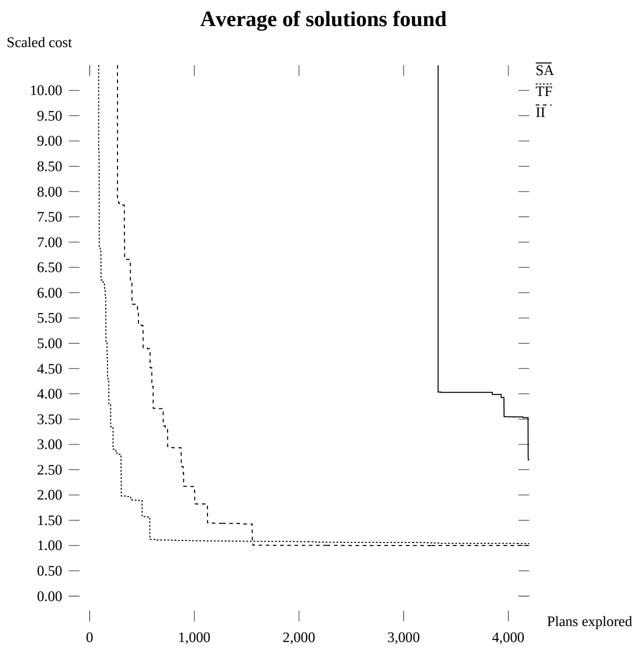
With the uniformly-random method we also sampled spaces that could not be explored exhaustively. These big spaces still showed some “good” plans but the percentage of “good” plans decreased. This is in line with the observations of [Swa89a] for the spaces of linear execution plans and with our results of Section 4. The decreasing number of good plans implies an increase in the number of plans considered by a transformation-free optimization strategy. But also other optimization strategies need to explore more plans as the search space increases. In the next Section we present a direct comparison of optimization algorithms.

7 Comparison of optimization algorithms

The two aspects which are important in comparing the performance of optimization algorithms are, the quality of the output and the time it takes to compute it. Instead of ‘time‘ we use the number of QEPs explored, because the processing time depends too much on the implementation and would make it harder to compare algorithms.

In an actual comparison of algorithms more weight could be given to either of these aspects. For example, when comparing algorithms for ad hoc queries the total time (optimization time + execution time of the query) would be a good measure.

We now compare the performance of an optimization method that relies completely on uniform random generation of candidates with two transformation-based optimization algorithms, *Simulated-annealing* and *Iterative-improvement*. Detailed descriptions of these algorithms can be found in [IK90, SG88]. Our implementation of SA and II is based on



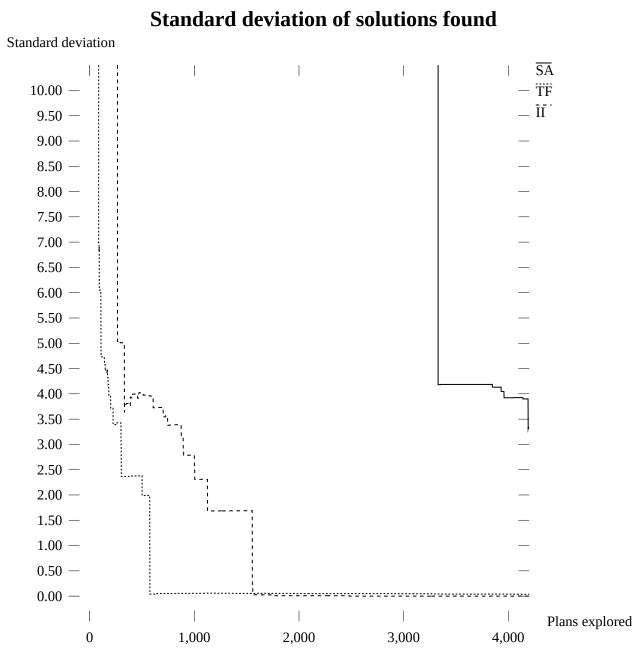


Figure 20: Average and standard deviation of cost of solutions found.

the pseudo-code presented in [IK90]. We also set the parameters of the algorithms to the values suggested in that paper.

The behaviour of an optimization strategy can be represented by a function mapping the number n of plans explored, to the estimated cost of the best plan 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 plans 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. Although the number of plans explored does not account for all the resources required by an algorithm, we follow [LVZ93] in using it as an approximation of an *implementation-independent* measure for optimization cost.

7.1 Results

In our 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 10,000 plans. The

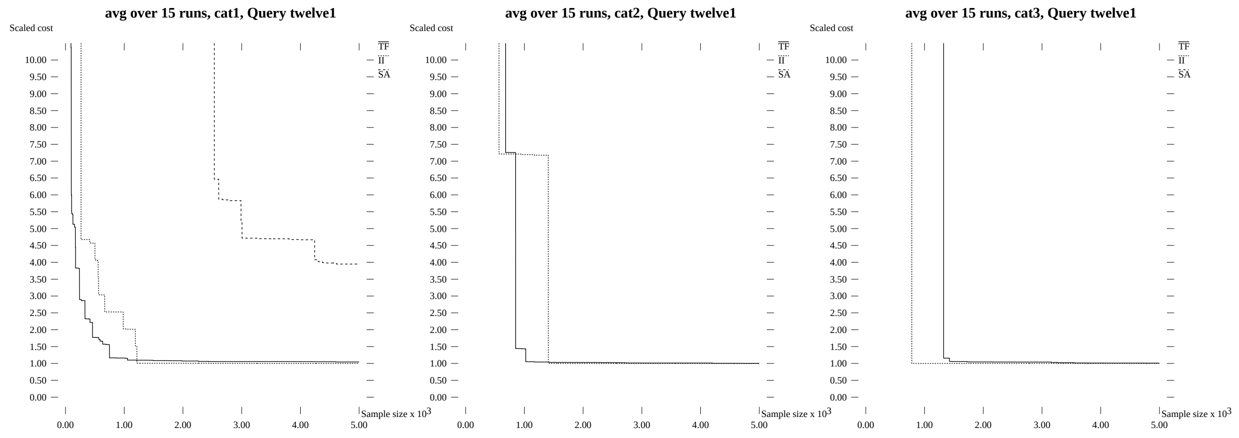


Figure 21: Average query on twelve relations

number of repetitions for each experiment was 20, 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 plan that is twice as expensive as that found by any method.

To analyze the results we computed the average and standard deviation of the solutions S_n^{II}, S_n^{SA} and S_n^{TF}, for $1 \leq n \leq 10,000$. The result of this analysis, for a query of 12 relations, is shown in Figure 20. To get a more readable graph, Figure 20 is limited to 4,000 plans. The graph is typical of the results we obtained on all queries and catalogs examined.

The average of the solutions found after 10,000 plans by each algorithm were avg($S_{10,000}^{SA}$) = 1.045, avg($S_{10,000}^{II}$) = 1.000, and avg($S_{10,000}^{TF}$) = 1.055. The standard deviations were std($S_{10,000}^{SA}$) = 0.055, std($S_{10,000}^{II}$) = 0.000, std($S_{10,000}^{TF}$) = 0.015.

On the average, SA is not able to find a good plan within 4000 plans; it finds these only after exploring 2000 more plans (not visible in the graph). On the average, TF finds good plans faster than II —e. g. TF finds plans with a scaled cost of 2 after exploring about 300 plans while II needs about a 1000. When II and TF keep exploring more plans II will find slightly better plans than TF. The maximum difference occurs after exploring 1800 plans but is very small (1.01 v.s. 1.08).

The other graph shows that TF not only finds good plans fast, on average, but that the quality of the plans found in different runs are also close together. After 600 plans the standard deviation of TF is already 0.04 while that of II is only about 3.90. This leads to consider the *time of convergence*, defined as the number of plans required to reach a given maximum standard deviation. Setting the threshold to 0.1, SA converges after exploring 8985 plans, finding a solution of average cost 1.11 at that point; II converges after 1559 plans, finding a solution of cost 1.05; and TF converges after 575 plans, finding a solution of cost 1.12.

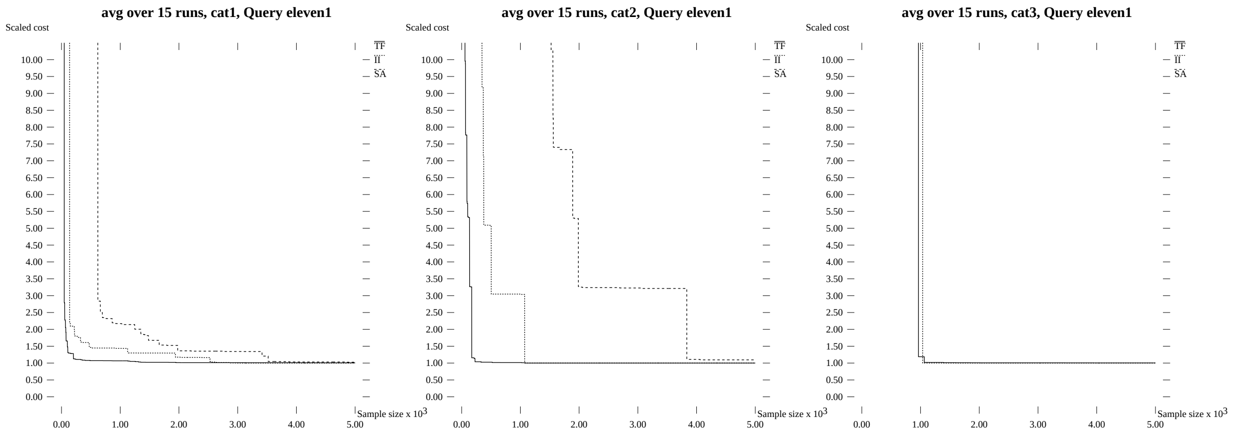


Figure 22: Average. Query on eleven relations

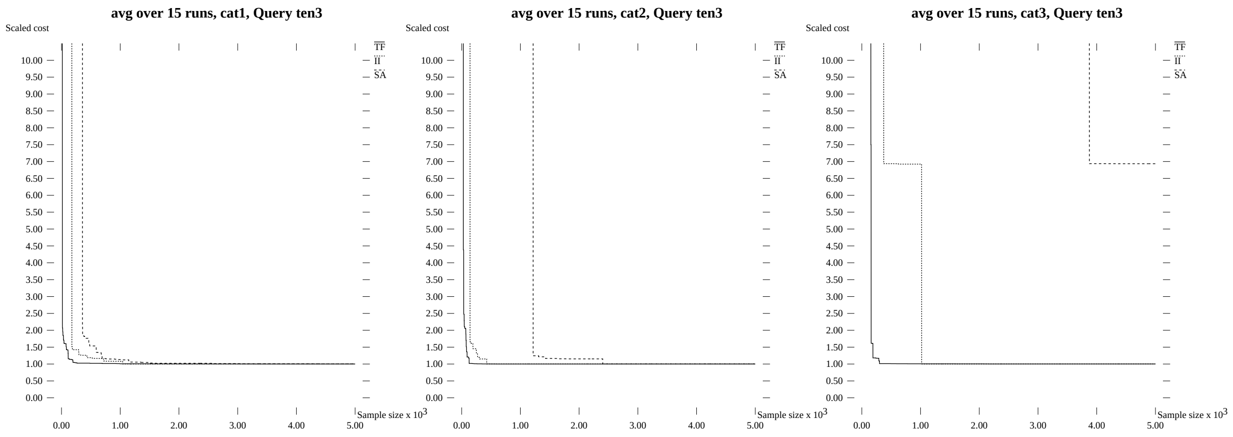


Figure 23: Average. Query on ten relations

8 Conclusions

Why use transformations? 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 plans to reach a local minimum, on average —yet we point out that the expected length of a sequence of random plans 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.*

We are currently investigating multi-phase, hybrid algorithms for fast-convergence/high-quality, based on our results. Nevertheless, 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.

Contributions. The prime novelty of this paper is its 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 plans uniformly distributed over the search space. This leads to a strategy where a random sequence of valid plans is generated and analyzed on their perceived cost. The best plan 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 plan. Our results then provide a natural baseline against which the added value of applying transformations and heuristics can be quantified.

To realize our proposed optimization scheme, we solve the problem of efficiently generating uniformly-distributed random plans, for queries with acyclic graphs. In the process, we also count the exact number of existing plans. Generation of a uniformly-distributed random plan is a basic primitive that other randomized algorithms can now use —due to the complexity of previously known methods, only *quasi*-random selection had been used for non-star graphs.

Related work. Transformation-based optimization is a general and powerful techniques with applications beyond join-order selection; see, for example, [FMV94]. More related to our present work, research on randomized optimization of join queries has been performed by Swami and Gupta [SG88, Swa89b, Swa89a, SI92], 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 QEPs. Finally, the cost model of Lanzelotte, Valduriez, and Zaït is that of a parallel database.

Future research. Our agenda for future research includes, first, the extension of our experiments to larger queries. Then, to remove our current restriction on the query graph topology, we need an efficient procedure to generate uniformly-distributed plans on cyclic graphs.

We are currently studying hybrid, multi-phase algorithms based on a "mix" of randomized choices, transformations, and heuristics. Depending on the ratio of optimization cost over query evaluation cost, and on the number of times the optimized query will be executed, some applications do need such a "mix," despite the likely necessary tuning.

Finally, we are also interested in the accuracy of the estimation of query evaluation cost —e. g. cost model calibration and size estimation of intermediate results— and how it affects the quality of the optimization output.

Acknowledgements. We are grateful to Johan van den Akker, Carel van den Berg, Josef Kolař, Fred Kwakkel, Zandra Navarro-Villicaña, Arno Siebes, and Chris Thieme for their suggestions and comments on preliminary drafts of this paper.

References

[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.

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

[GLPK94a] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Uniformly-distributed random generation of join orders. Technical report, CWI, 1994. Technical Report CS-R9431.

[GLPK94b] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten. Fast, randomized join-order selection —Why use transformations? In *Proceedings of the Twentieth*

International Conference on Very Large Databases, Santiago, 1994. Also CWI Technical Report CS-R9416.

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

[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.

[INSS92] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. *Proc. of the 18th VLDB Conference, Vancouver, British Columbia, Canada*, pages 103–114, 1992.

[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.

[Li86] L. Li. Ranking and unranking of AVL-trees. *SIAM Journal of Computation*, 15(4):1025–1035, November 1986.

[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.

[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.

[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.

[SI92] 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.