Optimizing Join Orders

Michael Steinbrunn[*] *Guido Moerkotte*[+] *Alfons Kemper*[*]

[*]Universität Passau
Fakultät für Mathematik
und Informatik
94030 Passau, Germany
kemper, *steinbrunn* @*db.fmi.uni-passau.de*

[+]Universität Karlsruhe
Fakultät für Informatik
76128 Karlsruhe, Germany
moer@ira.uka.de

Abstract

Recent developments in database technology, such as deductive database systems, have given rise to the demand for new, cost-effective optimization techniques for join expressions. In this paper many different algorithms that compute approximative solutions for optimizing join orders are studied since traditional dynamic programming techniques are not appropriate for complex problems. First, two possible solution spaces, the space of left-deep and bushy processing trees, respectively, are evaluated from a statistical point of view. The result is that the common limitation to left-deep processing trees is, from a purely statistical point of view, only advisable for certain cost models. Basically, optimizers from three classes are analysed: heuristic, randomized and genetic algorithms. Each one is extensively scrutinized with respect to its working principle and its fitness for the desired application. It turns out that randomized and genetic algorithms are well suited for optimizing join expressions. They generate solutions of high quality within a reasonable running time. The benefits of heuristic optimizers, namely the short running time, are often outweighed by merely moderate optimization performance.

1 Introduction

In recent years, relational database systems have become the standard in a variety of commercial and scientific applications. Because queries are stated in a non-procedural manner, the need for optimizers arises that transform the straightforward translation of a query into a cost-effective evaluation plan. Due to their high evaluation costs, joins are a primary target of query optimizers. If queries are stated interactively or embedded in a general purpose programming language, there are generally only few relations involved. The optimization of theses expressions can be carried out by exhaustive search, possibly enhanced by pruning techniques that exclude unlikely candidates for good solutions. For instance, in System R [SAC+79], a dynamic programming algorithm is employed for the optimization of joins. This approach works well as long as only few relations are to be joined, but if the join expression consists of more than about five or six relations, dynamic programming techniques become quickly prohibitively expensive. Queries of this kind are encountered in recent developments such as deductive database systems, where join expressions may consist of a large number of relations. Another source for such queries are long path expressions ($\text{obj.attr}_1.\text{attr}_2.\cdots.\text{attr}_n$) in object-oriented database systems that are to be traversed in reverse direction. Hence, there is a demand for optimization techniques that can cope with this kind of query in a cost-effective manner. In this paper, we shall examine approaches for the solution of this problem and assess their advantages and disadvantages. The rest of the article is organized as follows: In Section 2 we shall give an exact definition of the problem and of the terms, and we present several cost models we shall be using later on in our analysis. Section 3 deals with the problem of different solution spaces for evaluation strategies. In Section 4 we describe common optimization strategies with varying working principle, which are subject to a quantitative analysis in Section 5. Section 6 concludes the paper.

2 Problem Description

The problem of determining good evalution strategies for join expressions has been addressed as early as 1979 with the development of System R [SAC+79]. The work in this area can be divided into two major streams: First, the development of efficient algorithms for performing the join itself, and second, algorithms that determine the nesting order in which the joins are to be performed. In this article, we shall be concentrating on the generation of low-cost join nesting orders while disregarding the specifics of join computing—[ME92] provides a good overview on this subject.

In usual relational database systems, join expressions that involve more than about five or six relations are rarely encountered. Therefore, the computation of an optimal join order with lowest evaluation cost by exhaustive search is perfectly feasible—it takes but a few seconds CPU time. But if more than about eight relations are to be joined, the—in general—NP-hard problem of determining the optimal order [IK84] cannot be solved exactly anymore. We have to rely on algorithms that compute (hopefully) good approximative solutions. Those algorithms fall into two classes: first, augmentation heuristics that build an evaluation plan step by step according to certain criteria, and second, randomized algorithms that perform some kind of “random walk” through the space of all possible solutions seeking a solution with minimal evaluation cost.

2.1 Definition of Terms

The input of the optimization problem is given as the *query graph* (or, *join graph*), consisting of all relations that are to be joined as its nodes and all joins specified as its edges. The edges are labelled with the *join predicate* and the *join selectivity.* The join predicate maps tuples from the cartesian product of the adjacent nodes to $\{\text{false}, \text{true}\}$, depending whether the tuple is to be included in the result or not. The join selectivity is the ratio included/total number of tuples. As a special case, the cartesian product can be considered a join operation with join predicate $\equiv$ true and a join selectivity of 1.

The *search space* (or, *solution space*) is the set of all evaluation plans that compute the same result. A *point* in the solution space is one particular plan, i.e., solution for the problem. A solution is described by the *processing tree* for evaluating the join expression. Every point of the solution space has a *cost* associated with it; a *cost function* maps processing trees to their respective costs.

The processing tree itself is a binary tree that consists of base relations as its leaves and join operations as its inner nodes; edges denote the flow of data that takes place from the leaves of the tree to the root.

The goal of the optimization is to find the point in the solution space with lowest possible cost (global minimum). As the combinatorial explosion makes exhaustive enumeration of all possible solutions infeasible (cf. Table 1) and the NP-hard characteristic of the problem implies that there (presumably) cannot exist a faster algorithm, we have to rely on heuristics that compute approximative results.

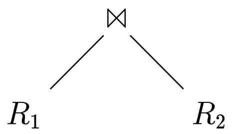
2.2 Cost Models

Our investigations are based on the cost models discussed in this subsection. Each of these cost models measures cost as the number of tuples that have to be processed. In spite of being rather simplish, this approach concentrates on the important characteristics of different join algorithms and their associated cost. The absolute accuracy of the cost estimates may not be satisfactory, but it turns out that these cost models are very well suited for comparative purposes such as ours.

The cost functions map processing trees, specifically root nodes of processing trees, to costs. For instance, the join $R_1 \bowtie R_2$ corresponds to the processing tree

#Relations (n)	#Processing Trees	#Solutions (#Trees $\cdot\, n!$)
1	1	1
2	1	2
3	2	12
4	5	120
5	14	1,680
6	42	30,240
7	132	665,280
8	429	17,297,280
9	1,430	518,918,400
10	4,862	17,643,225,600
11	16,796	670,442,572,800
12	58,786	28,158,588,057,600
$\vdots$	$\vdots$	$\vdots$

Table 1: Possible Nesting Orders for Join Evaluation



A cost function C assigns the cost for performing the join to the root node. It is computed recursively as the sum of the cost for obtaining the two son nodes and the cost for joining them in order to get the final result, i.e.,

$$C_{total} = C(R_1) + C(R_2) + C(R_1 \bowtie R_2) \tag{1}$$

where $C(R_k)$, the cost for obtaining the son nodes, is defined as

$$C(R_k) := \begin{cases} 0 & \text{if } R_k \text{ is a base relation} \\ C(R_i \bowtie R_j) & \text{if } R_k = R_i \bowtie R_j \text{ is an intermediate result} \end{cases}$$

As a special case, if both R_1 and R_2 are base relations, the total cost C_{total} equals $C(R_1 \bowtie R_2)$, as expected.

2.2.1 Result Cardinality

The most simple cost model takes just the cardinality of the join result into account, independently of the join method. For two relations, $C_{out}(R_1 \bowtie R_2) = |R_1 \bowtie R_2|$; for a general processing tree, the recursive definition (1) applies ($C_{out} = C_{out}(R_1) + C_{out}(R_2) + |R_1 \bowtie R_2|$).

2.2.2 Nested Loop

The nested-loop join is the standard strategy in case there are no indexes available that may support the join operation. Every tuple of the outer relation must be checked against every tuple of the inner relation, so we get $C_{nl}(R_1 \bowtie R_2) = |R_1| \cdot |R_2|$. The cost for a general processing tree is computed as defined in (1).

2.2.3 Sort-Merge

The sort-merge join performs the operation in two steps: first, the operand relations are sorted by the join attributes, if necessary. The cost for sorting one relation R is estimated as $|R| \log |R|$. Second, the operand relations are merged, which requires one pass through each one. Thus, the cost can be computed as $C_{sm}(R_1 \bowtie R_2) = |R_1| + |R_2| + |R_1| \log |R_1| + |R_2| \log |R_2| = |R_1| \cdot (1 + \log |R_1|) + |R_2| \cdot (1 + \log |R_2|)$. Again, the cost for a general processing tree is computed as defined in (1).

2.2.4 Hash Loop

This join method depends heavily on the order of the operand relations: only the outer relation must be scanned in its entirety, whereas access to the inner relation's tuples is carried out by means of a hash table on the join attribute. Thus, the cardinality of the inner relation does not influence the cost of the join, provided the hash table is properly designed (i.e., short overflow lists). Assuming the average length of the hash table's overflow lists as $h = 1.2$ yields a cost estimate of $C_{hl}(R_1 \bowtie R_2) = |R_1| \cdot h$; for a general processing tree, see (1).

2.2.5 Extended Nested Loop

In order to verify the hypothesis that the simple cost models presented above are sufficient for comparing different join optimization strategies, we also implemented a "real world" nested loop cost model derived from [Ull88] that yields an estimate for the number of page accesses from secondary memory that are necessary for performing the join operation:

$$C_{dnl}(R_1 \bowtie R_2) = B_{R_1} \cdot (\sigma_{12} \cdot |R_2| + \frac{B_{R_2}}{M-1}) + B_{R_2}(\sigma_{12} \cdot |R_1| + 1)$$

M is the number of pages available in main memory, B_R is the number of pages on secondary memory that hold tuples of the relation R, and σ_{12} is the selectivity of the join.

The smaller relation (assumed to be R_2) is divided into fragments of size $M - 1$. For each such fragment, a page from R_1 is fetched and the (partial) join performed. The cost C_{join} for this computation is

$$C_{join} = B_{R_1} \cdot \frac{B_{R_2}}{M-1} + B_{R_2}.$$

The cost C_{store} for storing the result is

$$C_{store} = \frac{\sigma_{12} \cdot |R_1| \cdot |R_2| \cdot (l_{R_1} + l_{R_2})}{b},$$

where l_R is the size of a tuple in relation R and b is the page size. $|R| \cdot l_R/b$ fits on B_R pages, thus

$$C_{store} = B_{R_1} \cdot \sigma_{12} \cdot |R_2| + B_{R_2} \cdot \sigma_{12} \cdot |R_1|$$

The complete cost C_{dnl} is $C_{join} + C_{store}$. For the computation of a general processing tree's cost, (1) applies here as well.

3 Solution Space for the Join Ordering Problem

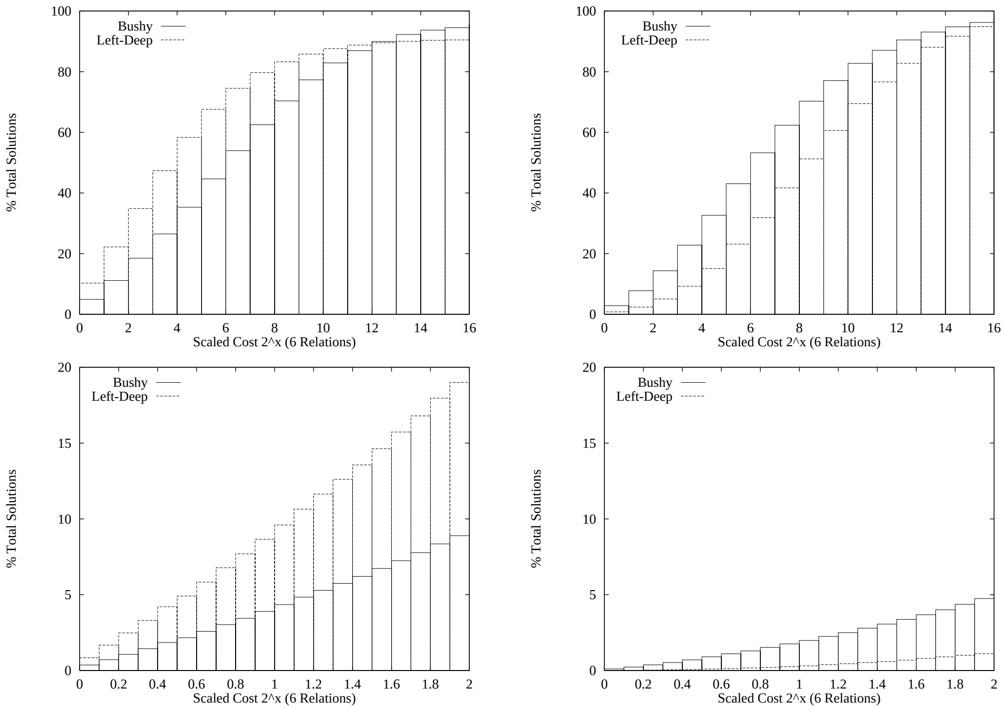
Generally, the solution space is defined as the set of all processing trees that compute the result of the join expression and that contain each base relation exactly once. The leaves of the processing trees consist of the base relations, whereas the inner nodes correspond to join results of the appropriate sons. As the join operation is commutative and associative, the number of possible processing trees increases quickly with increasing number of relations involved in the join expression in question. Traditionally, a subset of the complete space, the set of so-called *left-deep processing trees*, has been of special interest to researchers [SAC+79, SG88, Swa89]. We shall now study the characteristics of both the complete solution space and the subset of left-deep trees as the most interesting special cases, although other tree shapes might be contemplated, e.g., right-deep trees or zig-zag trees).

3.1 Left-Deep Trees

This subset consists of all processing trees where the inner relation of each join is a base relation. For a fixed number of base relations, the specification "left-deep" does not leave any degrees of freedom concerning the shape of the tree, but there are $n!$ ways to allocate n base relations to the tree's leaves. It has been argued that good solutions are likely to exist among these trees.

3.2 General (Bushy) Trees

The solutions in this space are in no way restricted. Consequently, the solution space with left-deep processing trees is a (true) subset of this solution space. Because the shape of possible processing trees can be arbitrary, the cardinality of this set is much higher: For n base relations, there are $\binom{2(n-1)}{n-1}(n-1)!$ different solutions.



(a) Nested Loop Cost Model

(b) Hash Loop Cost Model

Figure 1: Distribution of Costs

3.3 Cost Distribution

Especially randomized algorithms depend on the solution space: if the ratio good/bad solutions ("quality ratio") is too low, it may take a long time for these algorithms to find acceptable results. On the other hand, those algorithms that perform "random walks" are also influenced by the shape of the solution space [IK91]; however, we neglect this aspect in the following discussion and concentrate solely on the abovementioned "quality ratio."

Figures 1a and 1b show the quality of the solutions on the x-axis as scaled costs and their cumulative fraction of all possible solutions on the y-axis for six relations. The figures are averages of 200 randomly generated queries. The lower two pictures are magnifications of the most interesting interval 2^0–2^2.

For the nested loop cost model, we observe that about ten percent of the left-deep solutions are no worse than twice (2^1) the minimum cost, and a total of forty-five percent are better or equal to sixteen (2^4) times the optimum. On the other hand, the ratio of good solutions with bushy processing trees is much lower—only about five percent of all solutions incur costs with less than two times the minimum, and twenty-five percent are better or equal than sixteen times the minimum.

Changing the cost model makes a big difference: in contrast to the nested loop model, the hash loop cost model shows advantages for the bushy tree solution space, although the absolute percentages are lower than for the nested loop cost model. The reason for this surprising fact is the highly asymmetric cost model, where a general, bushy tree is (in principle) much better capable of exploiting the inherent optimization potential.

However, the optimization potential is not necessarily exploited equally well by the algorithms that construct processing trees of one of these two classes. If a higher potential is exploited poorly, the net result may well be just the contrary to the expectations suggested by the Figures 1a and 1b.

4 Join Ordering Strategies

The problem of finding a good nesting order for n-relational joins can be tackled in several different ways:

1. Deterministic Algorithms
   Every algorithm in this class constructs a solution step by step in a deterministic manner, either by applying a heuristic or by exhaustive search.

2. Randomized Algorithms
   Algorithms in this class pursue a completely different approach: first, a set of *moves* is defined. These moves constitute *edges* between the different solutions of the solution space; two solutions are connected by an edge if (and only if) they can be transformed into one another by exactly one move. Each of the algorithms performs a random walk along the edges according to certain rules, terminating as soon as no more applicable moves exist or a time limit is exceeded. The best solution encountered so far is the result.

3. Genetic Algorithms
   Genetic algorithms make use of a randomized search strategy very similar to biological evolution in their search for good problem solutions. Although in this aspect genetic algorithms resemble randomized algorithms as discussed above, the approach shows enough differences to warrant a consideration of its own. The basic idea is to start with a random population and generate offspring by random crossover and mutation. The "fittest" members of the population (according to the cost function) survive the subsequent selection; the next generation is based on these. The algorithm terminates as soon as there is no further improvement or after a predetermined number of generations. The fittest member of the last population is the solution.

4. Hybrid algorithms
   Hybrid algorithms combine the strategies of pure deterministic and pure

Customer

CId	CName	CFirst
2531	Jones	Bernhard
2532	Smith	Andrew
2533	Baker	Edward
2534	Bowler	Jim
2535	Miller	Humphrey

Order

CId	**TId**	**PId**	Qty.
2532	0-22233	GRA	50
2532	0-22233	DOV	20
2532	0-22691	GRA	10
2533	0-04434	FPR	10
2533	0-20691	GRA	10

Author

AId	AName	AFirst
ALHU	Huxley	Aldous
GEOW	Orwell	George
LECA	Carrol	Lewis
CHDI	Dickens	Charles

Book

TId	**PId**	Stock
0-04434	DOV	150
0-20691	FPR	100
0-21704	DOV	300
0-22233	DOV	50
0-22691	FPR	200
0-25183	DOV	200
0-25183	GRA	150

Publisher

PId	PName
DOV	Dover Publications
GRA	Grafton Books
FPR	Fiction Press

Work

TId	Title	AId
0-04434	Brave New World	ALHU
0-21704	Brave New World Revisited	ALHU
0-20691	Nineteen Eighty-Four	GEOW
0-22691	Animal Farm	GEOW
0-22233	Alice's Adventures in Wonderland	LECA
0-25183	Through the Looking-Glass	LECA

Figure 2: Sample Relations

randomized algorithms: solutions obtained by deterministic algorithms are used as starting points for randomized algorithms or as initial population members for genetic algorithms.

4.1 Deterministic Algorithms

In order to illustrate the working principle of each algorithm, we make use of an example database that consists of the six relations Customer, Order, Author, Work, Book and Publisher of a bookshop database (cf. Figure 2); the corresponding join graph is depicted in Figure 3. The key attributes are typeset in a boldface font. The work of an author may be published by several publishers

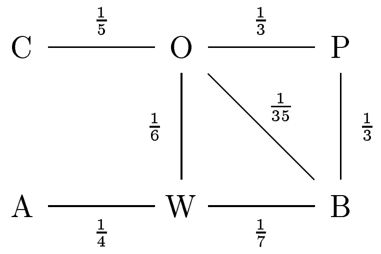


Figure 3: Join graph and selectivities for the relations in Figure 2

(e.g., hardcover and paperback). Thus, a book is defined by its title and its publisher (similar to the ISBN).

We shall now take a closer look at four different heuristic algorithms of this class . The asymptotic complexity of a best-first search is equivalent to exhaustive search, so the practical value is limited to a very small number of joining relations (about 8–10). We use this algorithm solely as the basis for comparison with algorithms that only compute approximative solutions in Section 5.

4.1.1 Minimum Selectivity

Good solutions are generally characterized by intermediate results with small cardinality. The *minimum selectivity heuristic* builds a left-deep processing tree step by step while trying to keep intermediate relations as small as possible. The *selectivity factor* s of the join $R_1 \bowtie R_2$, that is defined as

$$s := \frac{|R_1 \bowtie R_2|}{|R_1| \cdot |R_2|},$$

can be used to achieve this goal. First, the set of relations to be joined is divided into two subsets: the set of relations already incorporated into the intermediate result, denoted as $\mathcal{R}_{used}$ (which is initially empty), and the set of relations still to be joined with the intermediate result, denoted as $\mathcal{R}_{remaining}$ (which initially consists of the set of all relations). Then, in each step of the algorithm, the relation $R_i \in \mathcal{R}_{remaining}$ with the lowest selectivity factor

$$s_i := \frac{\left| R_i \bowtie \left(\bigotimes_{R_u \in \mathcal{R}_{used}} R_u \right) \right|}{|R_i| \cdot \left| \bigotimes_{R_u \in \mathcal{R}_{used}} R_u \right|}$$

is joined with the (so far) intermediate result and moved from $\mathcal{R}_{remaining}$ to $\mathcal{R}_{used}$. Figure 4 shows the complete algorithm for left-deep processing trees.

function MinSel

inputs rels "*List of relations to be joined*"
outputs pt "*Processing Tree*"

  pt := NIL

  do

    if pt = NIL **then**
      R_i := "*Relation with smallest cardinality*"
      pt := R_i
    else
      R_i := "*Relation from* rels *with smallest selectivity factor for the join with* pt"

$$\text{pt} := \begin{array}{c} \bowtie \\ / \qquad \backslash \\ \boxed{\text{pt}} \qquad R_i \end{array}$$

    end

    rels := rels − [R_i]

  while rels $\neq$ [ ]

return pt

Figure 4: Algorithm "Minimum Selectivity"

Example: The join order for the six relations "Customer" (C), "Order" (O), "Author" (A), "Publisher" (P), "Work" (W) and "Book" (B) is determined by MinSel as follows: First, the relation with the smallest cardinality, i.e., "Publisher," is selected. Then the selectivities for the remaining relations for the join with "Publisher" is computed. The selectivity for the join with "Book" and "Order" is equal (1/3, and the only ones that are less than 1, cf. Figure 3), so we choose arbitrarily "Book" as the second relation in the order. For the next relation, "Order" and "Work" are candidates. The selectivity for $PB \bowtie O$ is 1/35, whereas the selectivity for $PB \bowtie W$ is 1/7. Thus, the third relation is "Order." For the fourth relation, we determine the selectivity $PBO \bowtie W$ as 1/6 and $PBO \bowtie C$ as 1/5, so the we get $PBOW$ as the new partial order. The final order is $PBOWCA$, because the selectivity $PBOW \bowtie C$ is 1/5, whereas the selectivity for $PBOW \bowtie A$ is 1/4. The cost according to the hash loop cost model (Section 2.2.4) is $(|P|+|P \bowtie B|+|P \bowtie B \bowtie O|+|P \bowtie B \bowtie O \bowtie W|+|P \bowtie B \bowtie O \bowtie W \bowtie C|) \cdot h = 13h = 15.6$.

function TopDown

inputs rels "*List of relations to be joined*"
outputs pt "*processing tree*"

  if rels $\neq$ [ ] **then**
    R_i := "*Relation from* rels *that incurs the lowest cost when joined with all other relations from* rels"
    rels := rels \ [R_i]
  end

return

$$\begin{array}{ccc} & \bowtie & \\ / & & \backslash \\ \boxed{\text{TopDown(rels)}} & & R_i \end{array}$$

Figure 5: Algorithm "Top-Down"

4.1.2 Top-Down Heuristic

A further simple heuristic is the Top-Down heuristic. Its working principle is based on the observation that the last joins are the most important joins in the expression in terms of cost, simply because the intermediate results tend to be rather large towards the end of the evaluation. Therefore, the Top-Down heuristic selects the relation from the set that can be joined with lowest cost with the remaining relations. This strategy is applied repeatedly until no relations are left (Figure 5).

Example: The first relation in the order is the one with lowest cost if joined with the rest. The costs for each of the six possibilities is (based on the sample relations and the hash loop cost model):

$$\begin{array}{rcrcr}
C_{hl}(OBAPW \bowtie C) & = & h & = & 1.2 \\
C_{hl}(CBAPW \bowtie O) & = & 35h & = & 42.0 \\
C_{hl}(COAPW \bowtie B) & = & 5h & = & 6.0 \\
C_{hl}(COBPW \bowtie A) & = & h & = & 1.2 \\
C_{hl}(COBAW \bowtie P) & = & h & = & 1.2 \\
C_{hl}(COBAP \bowtie W) & = & h & = & 1.2
\end{array}$$

The cost for C, A, P and W is the same, so we choose one arbitrarily, C. The remaining five choices in the second step are:

$$\begin{array}{rclcl} C_{hl}(BAPW \bowtie O) & = & 7h & = & 8.4 \\ C_{hl}(OAPW \bowtie B) & = & 5h & = & 6.0 \\ C_{hl}(OBPW \bowtie A) & = & h & = & 1.2 \\ C_{hl}(OBAW \bowtie P) & = & h & = & 1.2 \\ C_{hl}(OBAP \bowtie W) & = & h & = & 1.2 \end{array}$$

Choosing A as the next relation, the third step is:

$$\begin{array}{rclcl} C_{hl}(BPW \bowtie O) & = & 7h & = & 8.4 \\ C_{hl}(OPW \bowtie B) & = & 5h & = & 6.0 \\ C_{hl}(OBW \bowtie P) & = & h & = & 1.2 \\ C_{hl}(OBP \bowtie W) & = & h & = & 1.2 \end{array}$$

The fourth step after choosing P:

$$\begin{array}{rclcl} C_{hl}(BW \bowtie O) & = & 7h & = & 8.4 \\ C_{hl}(OW \bowtie B) & = & 5h & = & 6.0 \\ C_{hl}(OB \bowtie W) & = & h & = & 1.2 \end{array}$$

After choosing W as the fourth relation in the order, the remaining choices are:

$$\begin{array}{rclcl} C_{hl}(B \bowtie O) & = & 7h & = & 8.4 \\ C_{hl}(O \bowtie B) & = & 5h & = & 6.0 \end{array}$$

Thus, we get B and O as the fifth and sixth relation, respectively, and the final order is $((((O \bowtie B) \bowtie W) \bowtie P) \bowtie A) \bowtie C$ with total cost $C_{hl} = (5+1+1+1+1) \cdot h = 9h = 10.8$.

4.1.3 Krishnamurthy-Boral-Zaniolo Algorithm

In [IK84], Ibaraki and Kameda showed that it is possible to compute the optimal nesting order in polynomial time, provided the query graph forms a tree (i.e., no cycles) and the cost function is a member of a certain class. On the basis of this result, Krishnamurthy, Boral and Zaniolo developed in [KBZ86] an algorithm (from now on called KBZ algorithm) that computes the optimal solution for a tree-query in $O(n^2)$ time, where n is the number of joins.

In the first step, every relation plays, in turn, the rôle of the root of the query tree. For all roots, the tree is linearized by means of a *ranking function* that establishes the optimal evaluation order for that particular root. The linearized tree obeys the tree's order, in other words, a parent node is always placed before the son nodes. The evaluation order with lowest cost is the result of the algorithm.

By transforming the query tree into a rooted tree, a parent node for every node can be uniquely identified. Thus, the selectivity of a join, basically an edge attribute of the query graph, can be assigned to the nodes as well. If the cost function C can be expressed as $C(R_i \bowtie R_j) = |R_i| \cdot g(|R_j|)$ where g is an arbitrary function, the join cost can be assigned to a particular node, too. This is true for the simple cost functions discussed in Section 2.2, with the exception of the cost function for the sort-merge join. The cost can be computed recursively as follows (Λ denotes the empty sequence, and l_1 and l_2 partial sequences):

$$
\begin{aligned}
C(\Lambda) &= 0 \\
C(R_i) &= \begin{cases} |R_i| & \text{if } R_i \text{ is the root node} \\ g(|R_i|) & \text{else} \end{cases} \\
C(l_1 l_2) &= C(l_1) + T(l_1)C(l_2)
\end{aligned}
$$

The auxiliary function $T(l)$ is defined as:

$$
T(l) = \begin{cases} 1 & \text{if } l = \Lambda \text{ (empty sequence)} \\ \prod_{R_k \in l} \sigma_k |R_k| & \text{else} \end{cases}
$$

σ_k denotes the selectivity of the join of R_k with its parent node.

The algorithm is based on the so-called "Adjacent Sequence Interchange Property" [IK84] for cost functions that can be expressed as $C(R_i \bowtie R_j) = |R_i| \cdot g(|R_j|)$. If the join graph J is a rooted tree and A, B, U and V are subsequences of J (U and V non-null), such that the partial order defined by J is not violated by $AVUB$ and $AUVB$, then

$$
C(AVUB) \leq C(AUVB) \Leftrightarrow \text{rank}(U) \leq \text{rank}(V)
$$

The *rank* of a non-null sequence S is defined as

$$
\text{rank}(S) := \frac{T(S) - 1}{C(S)}
$$

Thus, the cost can be minimized by sorting according to the ranking function rank(S), provided the partial order defined by the tree is preserved.

The algorithm for computing the minimum cost processing tree consists of the auxiliary function "linearize" and the main function "KBZ." First, the join tree is linearized according to the function *linearize* in Figure 6, where a bottom-up merging of sequences according to the ranking function is performed. In the last step, the root node becomes the head of the sequence thus derived. However, it is possible that the root node has a higher rank than its sons, therefore a normalization of the sequence has to be carried out. That means that the first relation in the sequence (the root node) is joined with its successor; if R_i is the root node and R_j its successor, the rank of the new relation R_{ij} is computed as

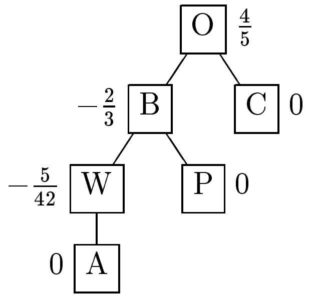
$$
\text{rank}(R_{ij}) = \frac{T(R_i) \cdot T(R_j) - 1}{C(R_i) + T(R_i) \cdot C(R_j)} \tag{2}
$$

```
function Linearize

inputs r "Root of a (partial) tree"
outputs chain "Optimal join order for the tree-shaped join graph with
               root r"
  chain := [ ]
  for all s in Sons(r)
    l := Linearize(s)
    "Merge l into chain according to ranks"
  end

  chain := r + chain
  "Normalize the root node r (cf. text)"

return chain
```

Figure 6: Auxiliary Function "linearize"

```
function KBZ

inputs joingraph
outputs minorder "join order"

  tree := "Minimum spanning tree of joingraph"
  mincost := ∞

  for all node in tree
    l := Linearize(node)
    "Undo normalization"

    cost := Cost(l)
    if cost < mincost then
      minorder := l
      mincost := cost
    end
  end

return minorder
```

Figure 7: KBZ-Algorithm



(a)
Initial Tree

	$\|R\|$	σ_R	$T(R)$	C(R)	rank(R)
O	5	1	5	5	4/5
C	5	1/5	1	h	0
B	7	1/35	1/5	h	−2/3
P	3	1/3	1	h	0
W	6	1/7	6/7	h	−5/42
A	4	1/4	1	h	0

(b)
Computation of Ranks

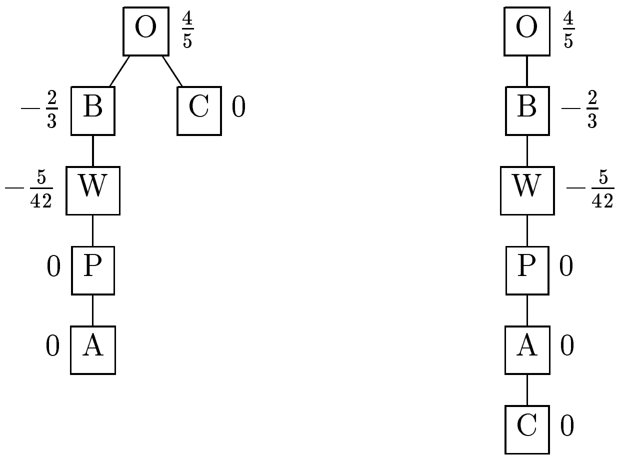
Figure 8: Initial Tree and Computation of Ranks

If necessary, this step has to be repeated until the order of the sequence is correct. The cost of the sequence is computed with the recursive cost function C.

In the main function *KBZ* (Figure 7), this procedure is carried out for each relation of the join graph acting as the root node. The sequence with lowest total cost is the result of the optimization.

The algorithm can be extended to general (cyclic) join graphs in a straightforward way, namely by reducing the query graph to its minimal spanning tree using Kruskal's algorithm [Kru56]. The weight of the join graph's edges is determined by the selectivity of the appropriate join, and the minimal spanning tree is determined as the tree with the lowest product of edge weights, rather than the sum of the edges' weights, as usual in other applications of Kruskal's algorithm. This extension has been suggested in [KBZ86]. However, if the join graph is cyclic, the result is no longer guaranteed to be optimal—it is but a heuristic approximation. When we speak of the "KBZ algorithm" in later sections, we refer to this extension with the computation of the minimal spanning tree of the join graph.

Example: Again we use the sample relations introduced at the beginning of this section. As the join graph is cyclic, we must compute the minimum spanning tree first. We remove the edges OW and OP from the join graph and get the tree in Figure 8a as the spanning tree. In the first step of the KBZ algorithm, the ranks of all relations must be computed. Starting with initial tree in Figure 8a, the result is depicted in the table in Figure 8b. h is, as in the previous examples, the average chain length of the hash loop cost model that serves as the function g in the cost function $C(R_1 \bowtie R_2) = |R_1| \cdot g(|R_2|)$ required for the KBZ algorithm. In order to obtain a chain of relations, the sons of the root node are linearized recursively. We start with the left subtree rooted at B. The sons of B are the leaf



(a)
First Linearization Step

(b)
Second Linearization Step

Figure 9: Illustration of the KBZ Algorithm

node P and the (linear) tree WA, so the linearization is carried out by merging P and WA into a chain according to their ranks. The ranks of P and A are equal, so we choose an arbitrary order (cf. Figure 9a). Because the root of this new, linear subtree (B) is already of lower rank than its son, no normalization is necessary, and we can proceed with the second merge step. The subtree rooted at B and the leaf node C are merged yielding the linearized tree in Figure 9b, which is the optimal order for this particular spanning join tree rooted at O; the associated cost is $C_{hl} = 10.8$.

This algorithm must be applied to all possible roots of the join tree; the lowest cost encountered is the final result. However, it is not necessary to perform the complete computation as shown above for each root node; moving to another root results in only minor changes to the ranks. For instance, in Figure 10 the change of the root from O to C is shown: only the ranks of O and C themselves must be recomputed, because only for these two nodes the parent changes.

If we carry out the algorithm for the remaining relations C, B, P, W, and A acting as root node, we obtain the following orders:

1. Relation C as root node:
   Join order: $((((C \bowtie O) \bowtie B) \bowtie P) \bowtie W) \bowtie A$, Cost: $C_{hl} = 15.6$

2. Relation B as root node:
   Join order: $((((B \bowtie O) \bowtie W) \bowtie P) \bowtie A) \bowtie C$, Cost: $C_{hl} = 13.2$

3. Relation P as root node:

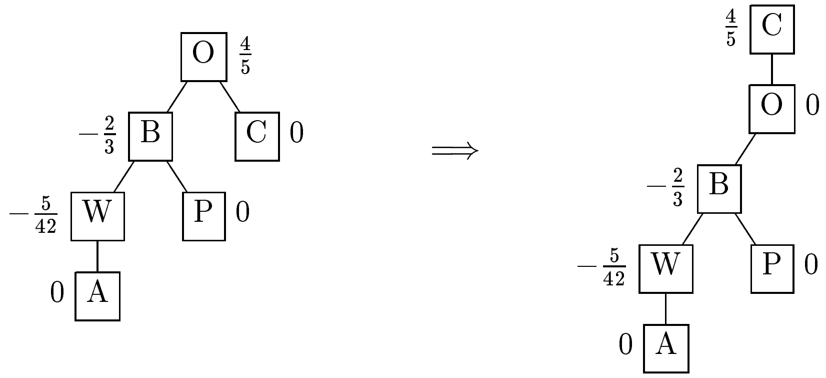


Figure 10: Changing the root node

Join order: $((((P \bowtie B) \bowtie O) \bowtie W) \bowtie A) \bowtie C$, Cost: $C_{hl} = 15.6$

4. Relation W as root node:
   Join order: $((((W \bowtie A) \bowtie B) \bowtie O) \bowtie P) \bowtie C$, Cost: $C_{hl} = 25.2$

5. Relation A as root node:
   Join order: $((((A \bowtie W) \bowtie B) \bowtie O) \bowtie P) \bowtie C$, Cost: $C_{hl} = 22.8$

The order for root node O is the optimal solution for left-deep processing trees and this particular spanning tree of the join graph; it can be verified (by exhaustive enumeration of all $6! = 720$ possibilities) that this is indeed the optimal left-deep processing tree solution. However, for this simple example, all algorithms presented in the previous two sections succeeded in finding this solution, too.

4.1.4 Relational Difference Calculus

The relational difference calculus (RDC) is a novel heuristic that emerged from the Boolean difference calculus [Sch89]. The Boolean difference calculus is used to determine the most influential variable in a Boolean expression in order to build a processing tree with low average cost. The details of this technique were presented in [KMS92]; it proved to be very effective in optimizing predicates in object-oriented DBMS applications where predicates can become arbitrarily expensive to evaluate. The relational difference calculus is the analogue of the Boolean difference calculus applied to the relational algebra, so the basic idea is the same, namely attempting to find the most influential relation in a join expression of the relational algebra.

Before we present the algorithm itself, we shall sketch the translation of the Boolean difference calculus from the Boolean algebra to the relational algebra, yielding the *relational difference*. First, we need a relational "1" and a "0" for

Boolean Algebra	Relational Algebra Symbol	Relational Algebra Definition
false	$0(sch(R_i))$	$\emptyset_{R_i}$
true	$1(sch(R_i))$	$R_i^* = \prod_{attribute \in \mathrm{sch}(R)} \mathrm{dom}(attribute)$
$\overline{x_i}$	$\overline{R_i}$	$R_i^* \backslash R$
$x_i \wedge x_j$	$R_i \bowtie R_j$	$R_i \bowtie R_j$
$x_i \vee x_j$	$R_i \oplus R_j$	$(R_i \bowtie R_j^*) \cup (R_i^* \bowtie R_j)$
$x_i \not\equiv x_j$	$R_i \Delta R_j$	$(R_i \bowtie \overline{R_j}) \cup (\overline{R_i} \bowtie R_j)$
$x_i \equiv x_j$	$R_i \bigtriangledown R_j$	$(R_i \bowtie R_j) \cup (\overline{R_i} \bowtie \overline{R_j})$

Table 2: Mapping of Boolean Algebra onto Relational Algebra

the Boolean "true" and "false." The empty set $\emptyset$ serves as the element "0" in the relational algebra, but "1" depends on the particular schema. Table 2 shows the translation for each property of the Boolean algebra into the corresponding property of the relational algebra. The symbol "$\bowtie$" denotes the natural join, and R^* denotes the *completion* of R, the cartesian product of all of R's attribute domains. The following lemmata base on these properties and state some facts of the relational algebra. The proofs are analogous to the Boolean algebra and are omitted in this paper.

Lemma 4.1

$$\begin{aligned} R_i^* &= R_i \cup \overline{R_i} \\ \overline{\emptyset_{R_i}} &= R_i^* \\ R_i \bowtie \emptyset_{R_i} &= \emptyset_{R_i} \\ R_i \bowtie R_i^* &= R_i \\ \overline{\overline{R_i}} &= R_i \end{aligned}$$

Lemma 4.2

$$\begin{aligned} R_i \bowtie R_j &= R_j \bowtie R_i \\ (R_i \bowtie R_j) \bowtie R_k &= R_i \bowtie (R_j \bowtie R_k) \\ R_i \bowtie (R_j \oplus R_k) &= (R_i \bowtie R_j) \oplus (R_i \bowtie R_k) \end{aligned}$$

Lemma 4.3

$$\begin{aligned} \overline{(R_i \bowtie R_j)} &= \overline{R_i} \oplus \overline{R_j} \\ \overline{(R_i \oplus R_j)} &= \overline{R_i} \bowtie \overline{R_j} \end{aligned}$$

In the Boolean algebra, $\{\not\equiv, \wedge\}$ is a complete system of operators. The corresponding operators $\{\Delta, \bowtie\}$ of the relational algebra do not constitute a complete system, because the projection cannot be expressed by means of symmetrical difference and join. However, we can prove the following lemma:

Lemma 4.4 *The System $\{\Delta, \bowtie, \pi\}$ is complete.*

Proof: Δ, $\bowtie$ and π are already contained in the system. The other operators are derived from these three as follows:

$$\begin{aligned} R_i \oplus R_j &= R_i \Delta (R_j \Delta (R_i \bowtie R_j)) \\ \overline{R_i} &= R_i \Delta R_i^* \end{aligned}$$

The *intersection* is a special case of the join: joining two relations with the same schema is equivalent to the intersection.

The *union* is a special case of the $\oplus$ operator; the union of two relations (with the same schema) is equivalent to applying the $\oplus$ operator.

As the *set difference* $A \setminus B$ can also be expressed as $A \cap \overline{B}$, this operation can also be performed in our system.

The selection operation on a relation R with selection predicate p can be expressed as a natural join of R with a second relation R' that consists of the join attributes whose values are such that they all satisfy p. Clearly, the result of the join is the same as the selection on R. $\square$

Shannon's decomposition theorem is also valid in the relational algebra:

Theorem 4.1 *Let E be a relational expression without projections. Then the following equivalence holds ($E[R_i \leftarrow a]$ denotes the expression that is derived by replacing R_i by a):*

$$E \;=\; (R_i \bowtie E[R_i \leftarrow R_i^*])\ \Delta\ (\overline{R_i} \bowtie E[R_i \leftarrow \emptyset_{R_i}])$$

We are now able to define the *relational difference* of a relational expression:

Definition 4.1 *Let $E(R_1, \ldots, R_n)$ be a relational expression. Then the* relational difference *of E with respect to R_i is defined as:*

$$\Delta_{R_i} E(R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_n) := E\ \Delta\ E[R_i \leftarrow \overline{R_i}]$$

If E does not contain projections, we can apply Shannon's theorem to obtain:

$$\Delta_{R_i} E(R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_n) := E[R_i \leftarrow 1(R_i)]\ \Delta\ E[R_i \leftarrow 0(R_i)]$$

Furthermore, if E is a join expression (no other operators but joins), the following definition is equivalent:

$$\Delta_{R_i} E := E[R_i \leftarrow 1(R_i)]\ \Delta\ E[R_i \leftarrow 0(R_i)]$$

In order to distinguish the relational difference Δ_{R_i} and the set-operation symmetrical difference Δ, the relational difference is always written with index. Some properties of the relational difference follow:

Lemma 4.5

$$\begin{aligned}
\Delta_{R_i} E &= \Delta_{\overline{R_i}} E \\
\Delta_{R_i} \overline{E} &= \Delta_{R_i} E \\
\Delta_S E(R_1, \ldots, R_n) &= \emptyset_E \quad (S \neq R_i, i = 1, \ldots, n) \\
E &= \Delta_{R_i} E \,\Delta\, E[R_i \leftarrow \overline{R_i}] \\
\Delta_{R_i}(E \Delta F) &= \Delta_{R_i} E \,\Delta\, \Delta_{R_i} F
\end{aligned}$$

If the relational difference is to be applied more than once to a relational expression, there are two possibilities for doing so: subsequent and simultaneous application. The following two definitions reflect these possibilities:

Definition 4.2 (Sequencing of the Relational Difference)

$$\Delta^2_{R_i, R_j} E \;:=\; \Delta_{R_j}(\Delta_{R_i} E)$$

Definition 4.3 (Joint Relational Difference)

$$\Delta_{R_i, R_j} E \;:=\; E \,\Delta\, E[R_i \leftarrow \overline{R_i}, R_j \leftarrow \overline{R_j}]$$

Another analogy to the Boolean difference calculus is the existence of the *chain rule* of the relational difference calculus.

Before the chain rule of relational difference calculus can be stated, we need Shannon's decomposition theorem:

Theorem 4.2 (Shannon Decomposition Theorem) *Let $E(R_1, \ldots, R_n)$ be a relational expression without projection operation. Then E is* decomposable*, i.e., there exist expressions $\tilde{E}$, D such that*

$$E(R_1, \ldots, R_n) = \tilde{E}(R_1, \ldots, R_i, D(R_{i+1}, \ldots, R_m), R_{m+1}, \ldots, R_n)$$

Theorem 4.3 (Chain Rule) *Let E be a decomposable relational expression, (i.e. without projections), then for $i+1 \leq k \leq m$ the following equation holds:*

$$\begin{aligned}
&\Delta_{R_l} E(R_1, \ldots, R_i, R_{i+1} \ldots, R_l, \ldots, R_m, R_{m+1} \ldots, R_n) = \\
&\quad \Delta_D \tilde{E}(R_1, \ldots, D(R_{i+1}, \ldots, R_m)) \bowtie \Delta_{R_l} D(R_{i+1}, \ldots, R_m)
\end{aligned}$$

Now that we are confident that the translation of Boolean algebra to relational algebra is indeed possible, we are going to develop an algorithm for the optimization of join expressions as a special case of general relational expressions. The basic idea is to use the relational difference as a "detector" for "important" relations that are to be processed early in the evaluation process of a join expression. But before presenting the algorithm for join optimization, we shall illustrate the working principle by means of the related Boolean difference calculus for the optimization of Boolean expressions, because it becomes more apparent this way.

In a Boolean expression like, for instance, $\phi = (x_0 \vee x_1) \wedge x_2$, not all variables have the same influence on the final result. x_2 is much more important, because if x_2 evaluates to "false," the entire expression evaluates to "false," whereas neither value of x_0 or x_1 has a similar impact on the result. Therefore, it may be advantageous to evaluate x_2 first, and, if the value turns out to be "false," to omit the evaluation of x_0 or x_1 thus saving evaluation costs. The question whether it is really advantageous to evaluate x_2 first depends on the probability for being "true" and the actual evaluation cost for x_2. The probability for being "true" of the variable's Boolean difference being true is a measure for the influence on the expression's result independently of the evaluation cost. Therefore, the quotient of this probability and the variable's evaluation cost defines a *weight* w that determines the evaluation order for the variables of the expression, namely variables with high weight are evaluated prior to variables with low weight. For the function ϕ, the computation of the weight yields

$$\begin{aligned} w_0 &= \frac{\text{probability}(\Delta_{x_0}\phi)}{\text{cost}(x_0)} \\ w_1 &= \frac{\text{probability}(\Delta_{x_1}\phi)}{\text{cost}(x_1)} \\ w_2 &= \frac{\text{probability}(\Delta_{x_2}\phi)}{\text{cost}(x_2)} \end{aligned}$$

and the variable with the lowest weight is evaluated first. A detailed description of this algorithm can be found in [KMS92].

A similar weighing for relations of a join expression can be carried out by the relational difference, namely by computing $\Delta_{R_i}(E)$ for every relation R_i in the join expression E. The cardinality of $\Delta_{R_i}(E)$ serves the same purpose as the probability of $\Delta_{x_i}(\phi)$ for the Boolean optimization. Thus, we get

$$w_i = |\Delta_{R_i}(E)|$$

as the weight factor; the complete algorithm for the relational difference optimization (RDC) is depicted in Figure 11.

There is one major difference to the algorithm for optimizing Boolean expressions: A cost factor is not taken into account. The reason for this—at first glance

function SortRDC

inputs rels "*List of relations to be joined*"
outputs rels "*Join Order*"

forall $\mathrm{R_i}$ **in** rels
$\mathrm{w_i} := \left|\Delta_{\mathrm{R_i}}\left(\bowtie_{\mathrm{R_j}\in\mathrm{rels}} \mathrm{R_j}\right)\right|;$
end

"*Sort relations 'rels' by descending w_i*"

return rels

Figure 11: RDC Optimization

surprising—fact is the structure of the resulting fraction. A sensible cost factor would measure the join cost of R_i with the join results of all other relations of the join expression, i.e.,

$$\mathrm{cost}_i = \left| R_i \bowtie \left(\bowtie_{R_j \in rels \setminus R_i} R_j \right) \right|$$

Comparing this cost factor with the definition of the relational difference, we recognize many similarities that lead to a cancelling down of a significant part of the fraction. This observation is also backed by preliminary experiments that we carried out; as a matter of fact, computing the weight as $w_i = \frac{|\Delta_{R_i}(E)|}{\mathrm{cost}_i}$ results in such a bad performance that we did not pursue this variant any further.

Instead, the effect of the weight computed as the pure cardinality of the relational difference can be compared to the minimum selectivity heuristic: relations that join with low selectivity factor are preferred. But, in contrast to the minimum selectivity heuristic, the number of join attributes is also taken into account, yielding an algorithm that is superior to simple "Minimum Selectivity."

Example: We illustrate the RDC algorithm by optimizing the join operation of our sample relations C, O, A, W, B and P. For each of these six relations, the weight w must be computed by means of the relational difference calculus.

According to the algorithm in Figure 11, the following values are necessary:

$$
\begin{array}{lclcl}
w_C & = & |\Delta_C(C \bowtie O \bowtie A \bowtie W \bowtie P \bowtie B)| & = & |C^* \bowtie O \bowtie A \bowtie W \bowtie P \bowtie B| \\
w_O & = & |\Delta_O(C \bowtie O \bowtie A \bowtie W \bowtie P \bowtie B)| & = & |C \bowtie O^* \bowtie A \bowtie W \bowtie P \bowtie B| \\
w_A & = & |\Delta_A(C \bowtie O \bowtie A \bowtie W \bowtie P \bowtie B)| & = & |C \bowtie O \bowtie A^* \bowtie W \bowtie P \bowtie B| \\
w_W & = & |\Delta_P(C \bowtie O \bowtie A \bowtie W \bowtie P \bowtie B)| & = & |C \bowtie O \bowtie A \bowtie W^* \bowtie P \bowtie B| \\
w_P & = & |\Delta_P(C \bowtie O \bowtie A \bowtie W \bowtie P \bowtie B)| & = & |C \bowtie O \bowtie A \bowtie W \bowtie P^* \bowtie B| \\
w_B & = & |\Delta_B(C \bowtie O \bowtie A \bowtie W \bowtie P \bowtie B)| & = & |C \bowtie O \bowtie A \bowtie W \bowtie P \bowtie B^*|
\end{array}
$$

The completion R^* for a relation R with attributes $A_1, \ldots, A_n$ is defined as the cartesian product

$$\mathrm{dom}(A_1) \times \mathrm{dom}(A_2) \times \ldots \times \mathrm{dom}(A_n)$$

However, those attribute that do not participate in the join must be excluded (projection without elimination of duplicates) in order to avoid spuriously large weights for relations with few participating attributes. For instance, we must exclude the attribute "Qty." from the computation of O^* and get:

$$
\begin{array}{l}
\{2532, 2533\} \times \{\text{0-22233}, \text{0-22691}, \text{0-04434}, \text{0-20691}\} \times \{\text{DOV}, \text{GRA}, \text{FPR}\} \\
\quad = \quad \{(2532, \text{0-22233}, DOV), (2532, \text{0-22233}, \text{GRA}), (2532, \text{0-22233}, \text{FPR}), \\
\qquad\qquad \ldots \\
\qquad\quad (2533, \text{0-20691}, \text{DOV}), (2533, \text{0-20691}, \text{GRA}), (2533, \text{0-20691}, \text{FPR})\}
\end{array}
$$

Thus, the cardinality $|C \bowtie O^* \bowtie A \bowtie W \bowtie P \bowtie B|$ (and the weight w_O) can be determined as $w_O = 8$.

Similarly, we get $w_C = 1$, $w_A = 1$, $w_W = 3$, $w_P = 1$ and $w_B = 5$. Since the relations are sorted in decreasing order, we obtain several possible join orders with equal cost starting with O, B and W, for instance $((((O \bowtie B) \bowtie W) \bowtie P) \bowtie A) \bowtie C$ with associated cost $C_{hl} = 10.8$, that is, as we already have mentioned, the optimum.

4.1.5 AB Algorithm

The AB algorithm has been developed recently by Swami and Iyer [SI93]. It is based on the KBZ algorithm with various enhancements. The algorithm permits the use of two different cost models, namely nested loop and sort-merge. The sort-merge cost model has been simplified by Swami and Iyer such that it confirms to the requirements of the KBZ algorithm ($C(R_1 \bowtie R_2) = |R_1| \cdot g(|R_2|)$ for some function g, cf. Section 4.1.3). The choice for the application of these cost models takes place at the level of individual joins rather than for all joins of he expression as was assumed so far. The algorithm runs as follows (cf. Figure 12):

1. In `randomize_methods`, each join in the join graph is assigned a randomly selected join method. If the join graph is cyclic, a random spanning tree is selected first.

```
function AB
inputs joingraph
outputs minorder "join order"
  while number of iterations ≤ N^2 do
  begin
    randomize_methods;
    while number of iterations ≤ N^2 do
    begin
      apply_KBZ;
      change_order;
      change_methods;
    end;
  end;
  post_process;
return minorder;
```

Figure 12: AB-Algorithm

2. The resulting tree query is optimized by the KBZ algorithm (`apply_KBZ`). The algorithm is extended in such a way that different cost models (as mentioned above) are permitted at each edge of the join graph. However, the assignment of join methods to edges is not changed during in the KBZ optimization phase.

3. The KBZ algorithm does not take "interesting orders" (tuple orders that are advantageous for later joins) into account. Therefore, `change_order` attempts to further reduce the cost by swapping relations such that interesting orders can be exploited.

4. The next step comprises a single scan through the join order achieved so far. For each join, an attempt is made to reduce the total cost by changing the join method employed (`change_method`).

5. Steps 2 to 4 are iterated until no further improvement is possible or N^2 iterations are performed (N = number of joins in the join graph).

6. Steps 1 to 5 are repeated as long as the total number of iterations of the inner loop does not exceed N^2.

7. In a post-processing step (`post_process`), once more the order of the relations is changed in an attempt to reduce the cost. In contrast to

`change_order`, cartesian products are now permitted because the result need not be capable of serving as input for the KBZ algorithm.

The AB algorithm comprises elements of heuristic and randomized optimizers. The inner loop searches heuristically for a local minimum, whereas in the outer loop several random starting points are generated in the manner of the Iterative Improvement algorithm (cf. Section 4.2.2).

4.2 Randomized Algorithms

Randomized algorithms view solutions as *points* in a solution space and connect these points by edges that are defined by a set of *moves*. The algorithms discussed below perform some kind of random walk through the solution space along the edges defined by the moves. The kind of moves that are considered depend on the solution space: if left-deep processing trees are desired, each solution can be represented uniquely by an ordered list of relations participating in the join. Two different moves are proposed in the literature [SG88, Swa89] for modifying these solutions: "Swap" and "3Cycle." "Swap" exchanges the positions of two arbitrary relations in the list, and "3Cycle" performs a cyclic rotation of three arbitrary relations in the list. For instance, if $R_1R_2R_3R_4R_5$ was a point in the solution space, application of "Swap" might lead to $R_1R_4R_3R_2R_5$, whereas "3Cycle" could yield $R_5R_2R_1R_4R_3$.

If the complete solution space with arbitrarily shaped (bushy) processing trees is considered, the moves depicted in Figure 13 (introduced in [IK91]) are used for traversal of the solution space.

4.2.1 Random Walk

The most simple randomized algorithm performs a random walk through the solution space starting from a randomly selected point. In each step of the algorithm, a random move is carried out, and if the new point constitutes a better solution (in terms of the chosen cost function), it is stored as the new result. The algorithm terminates after a certain predetermined number of moves are made, and the best solution encountered is the final result. In principle, this algorithm draws a random sample from the set of all possible solutions and delivers the best solution from the sample as result. The performance of this algorithm depends heavily on the fraction of good to bad solutions in the solution space and on the size of the sample drawn. However, it is obvious that such a strategy is a waste of computing resources, because only a small area in close proximity of the starting point is covered and no attempt is made to seek a path that approaches a (at least local) minimum. Therefore, we mention this strategy only for the sake of completeness and do not assess its (rather doubtful) merits in detail.

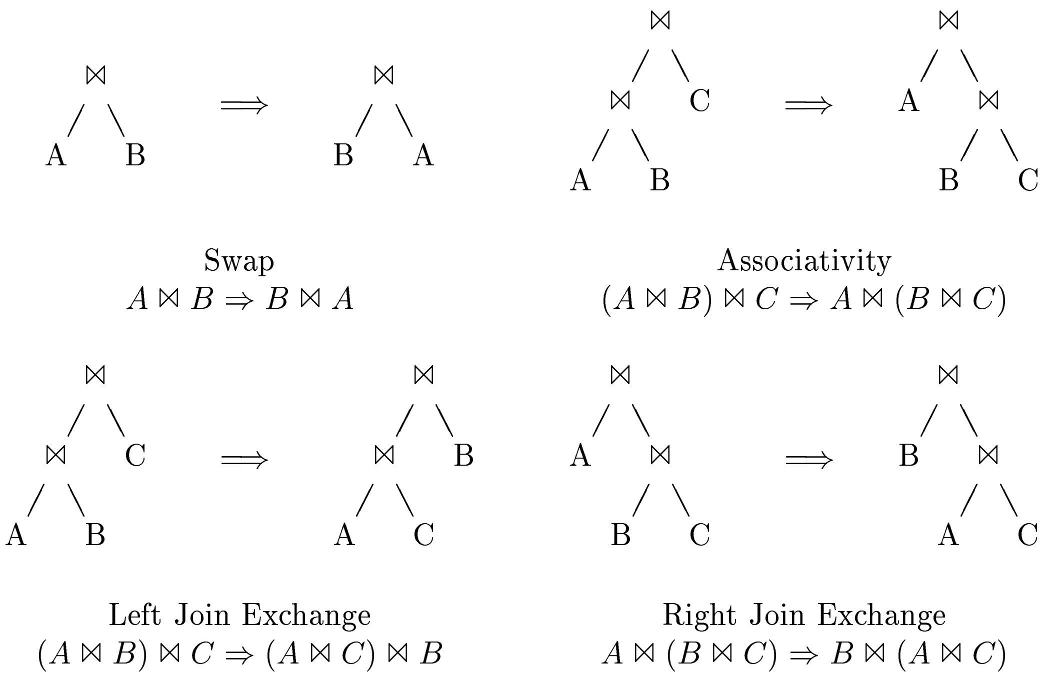


Figure 13: Moves for Bushy-Tree Solution Space Traversal

4.2.2 Iterative Improvement

A more sophisticated approach is the *Iterative Improvement Algorithm* [SG88, Swa89, IK90] (Figure 14). After selecting a random starting point, the algorithm seeks a minimum cost point using a strategy similar to hill-climbing. Beginning at the starting point, a random neighbour (i.e., a point that can be reached by exactly one move) is selected. If the cost associated with the neighbouring point is lower than the cost of the current point, the move is carried out and a new neighbour with lower cost is sought. This strategy is insofar different from genuine hill-climbing, as no attempt is made to determine the neighbour with lowest cost. The reason for this behaviour is the generally very high number of neighbours that would have to be checked. The same holds for the check whether a given point is a local minimum or not. Instead of systematically enumerating all possible neighbours and checking each one individually, a point is assumed to be a local minimum if no lower-cost neighbour can be found in a certain number of tries.

This procedure is repeated until a predetermined number of starting points are processed or a time limit is exceeded. The lowest local minimum encountered is the result.

```
function IterativeImprovement
outputs minstate "Optimized processing tree"
  mincost := ∞
  do
    state := "Random starting point"
    cost := Cost(state)
    do
      newstate := "state after random move"
      newcost := Cost(newstate)
      if newcost < cost then
        state := newstate
        cost := newcost
      end
    while "Local minimum not reached"
    if cost < mincost then
      minstate := state
      mincost := cost
    end
  while "Time limit not exceeded"
return minstate
```

Figure 14: Iterative Improvement

4.2.3 Simulated Annealing

Iterative Improvement suffers from a major drawback: Because moves are accepted only if they improve the result obtained so far, it is possible that even with a high number of starting points the final result is still unacceptable. This is the case especially when the solution space does not show a single, global minimum, but a large number of high-cost local minima. In this case, the algorithm gets easily "trapped" in one of the high-cost local minima.

Simulated Annealing (Figure 15) is a refinement of Iterative Improvement that removes this restriction [IW87, SG88]. In Simulated Annealing, a move may be carried out even if the neighbouring point is of higher cost. Therefore, the algorithm does not get trapped in local minima as easily as Iterative Improvement. As the name "Simulated Annealing" suggests, the algorithm tries to simulate the annealing process of crystals. In this natural process, the system eventually reaches a state of minimum energy. The slower the temperature reduction is car-

```
function SimulatedAnnealing

inputs state "Random starting point"
outputs minstate "Optimized processing tree"

  minstate := state
  cost := Cost(state)
  mincost := cost
  t := "Starting temperature"
  do

    do
      newstate := "state after random move"
      newcost := Cost(newstate)
      if newcost ≤ cost then
        state := newstate
        cost := newcost
      else "With probability e^((newcost−cost)/t)"
        state := newstate
        cost := newcost
      end

      if cost < mincost then
        minstate := state
        mincost := cost
      end
    while "Equilibrium not reached"

    "Reduce Temperature"
  while "Not frozen"

return minstate
```

Figure 15: Simulated Annealing

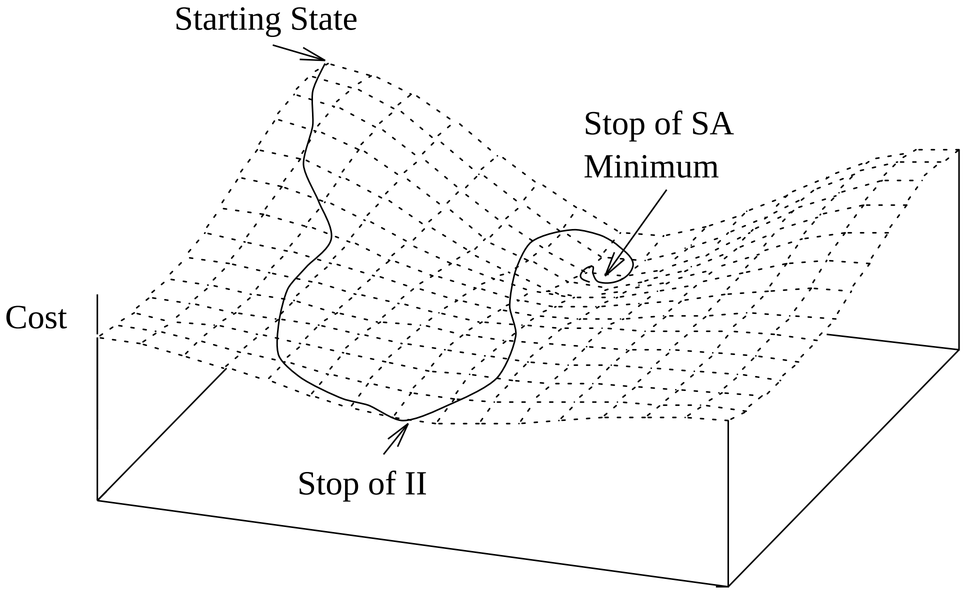


Figure 16: Iterative Improvement vs. Simulated Annealing

ried out, the lower the energy of the final state (one large crystal is of lower energy than several smaller ones combined). Figure 16 illustrates this behaviour: Iterative Improvement stops in the first local minimum, whereas Simulated Annealing overcomes the high-cost barrier that separates it from the global minimum, because the SA algorithm always accepts moves that lead to a lower cost state, whereas moves that increase costs are accepted with a probability that depends on the temperature and the difference between the actual and the new state's cost.

Of course, the exact behaviour is determined by parameters like starting temperature, temperature reduction and stopping condition. Several variants have been proposed in the literature—we shall present the detailed parameters in the next section where we analyse and compare those SA variants.

4.2.4 Two-Phase Optimization

The basic idea for this variant is the combination of Iterative Improvement and Simulated Annealing in order to combine the advantages of both [IK90]. Iterative Improvement, if applied repeatedly, is capable of covering a large part of the solution space, whereas Simulated Annealing is very well suited for thoroughly covering the neighbourhood of a given point in the solution space.

Thus, *Two-Phase Optimization* works as follows:

1. for a number of randomly selected starting points, local minima are sought by way of Iterative Improvement (Figure 14), and

2. from the lowest of these local minima, the Simulated Annealing algorithm

(Figure 15) is started in order to search the neighbourhood for better solutions.

Because only the close proximity of the local minimum needs to be covered, the initial temperature for the Simulated Annealing pass is set lower than it would be for the Simulated Annealing algorithm run by itself.

4.3 Genetic Algorithms

Genetic algorithms are designed to simulate the natural evolution process. As in nature, where the fittest members of a population are most likely to survive and inherit their features to their offspring, genetic algorithms propagate solutions for a given problem from generation to generation, combining them to achieve further improvement. We provide a brief overview of the terminology and the working principles of genetic algorithms. For a comprehensive introduction, the reader is referred to, e.g., [Gol89].

4.3.1 Terminology

Because genetic algorithms are designed to simulate biological processes, much of the terminology used to describe them is borrowed from biology. One of the most important characteristics of genetic algorithms is that they do not work on a single solution, but on a set of solutions, the *population*. A single solution is sometimes called *phenotype*. Solutions are always represented as *strings (chromosomes)*, composed of *characters (genes)* that can take one of several different *values (alleles)*. The *locus* of a gene corresponds to the *position* of a character in a string. Each problem that is to be solved by genetic algorithms, must have its solutions represented as character strings by an appropriate *encoding*.

The “fitness” of a solution is measured according to an objective function that has to be maximized or minimized. Generally, in a well-designed genetic algorithm both the average fitness and the fitness of the best solution increases with every new generation.

4.3.2 Basic Algorithm

The working principle of the genetic algorithm that we use to optimize join expressions is the same as the generic algorithm described below.

First, a population of random character strings is generated. This is the “zero” generation of solutions. Then, each next generation is determined as follows:

- A certain fraction of the fittest members of the population is propagated into the next generation *(Selection)*.
- A certain fraction of the fittest members of the population is *combined* yielding offspring *(Crossover)*.

- And a certain fraction of the population (not necessarily the fittest) is altered randomly *(Mutation)*.

These fractions are chosen such that the total number of solutions in the population remains invariant.

This loop is iterated until the best solution in the population has reached the desired quality, a certain, predetermined number of generations has been produced or the improvement from one generation to the next has fallen below a certain threshold. In the next section, we shall examine how this generic algorithm can be adapted to the problem of optimizing join expressions.

4.3.3 Genetic Algorithm for Optimizing Join Expressions

Encoding Before a genetic algorithm can be applied to solve a problem, an appropriate encoding for the solution and an objective function has to be chosen. For join optimization, the solutions are processing trees, either left-deep or bushy, and the objective function is the evaluation cost of the processing tree that is to be minimized.

The encoding of left-deep processing trees is straightforward: every processing tree of this kind can be represented uniquely by an ordered list of its leaves, as in the following example:

$$((((R_1 \bowtie R_4) \bowtie R_3) \bowtie R_2) \bowtie R_5) \stackrel{\text{Encoding}}{\Longrightarrow} \boxed{14325}$$

Another possible encoding works as follows (we encode the same processing tree $((((R_1 \bowtie R_4) \bowtie R_3) \bowtie R_2) \bowtie R_5))$:

- An ordered list L of all participating relations is made (for instance, based on their indices), such as $L = [R_1, R_2, R_3, R_4, R_5]$.
- The first relation in the processing tree, R_1, is also the first relation in our list L, so its index "1" is the first component of the encoding string. R_1 is then removed from the list L, so $L := [R_2, R_3, R_4, R_5]$.
- The second relation in the processing tree, R_4, is the third relation in the list L, so "3" becomes the second component of the encoding string. After removal of R_4, L becomes $[R_2, R_3, R_5]$.
- This process is repeated until the list L is exhausted. In our example, the final encoding is:

$$((((R_1 \bowtie R_4) \bowtie R_3) \bowtie R_2) \bowtie R_5) \stackrel{\text{Encoding}}{\Longrightarrow} \boxed{13211}$$

We shall refer to this encoding strategy as "Salesman Encoding", because an encoding of this kind has been successfully applied to the approximative solution of the "Travelling Salesman Problem" with genetic algorithms.

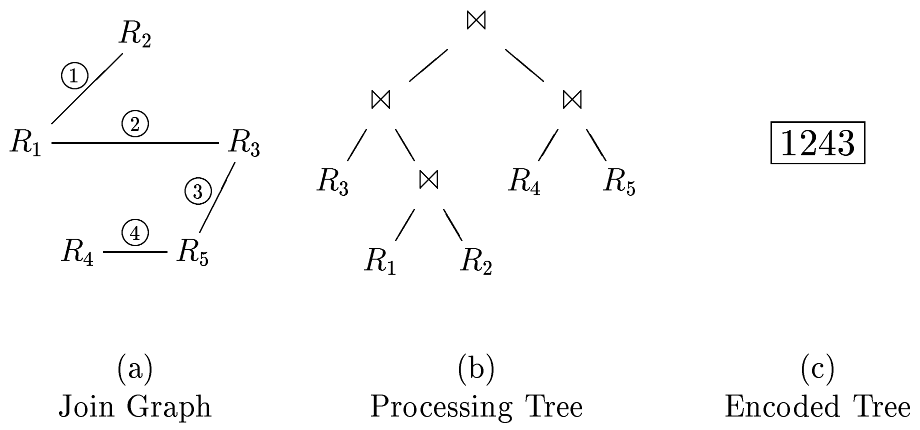


Figure 17: Encoding of Bushy Processing Trees

Encoding bushy processing trees is slightly more tricky, because we do not want crossover or mutation operators that work on the strings to get overly complex. A good scheme has been used in [BFI91], where the characters of the string represent edges of the join graph. Thus, as a side effect, processing trees that specify cartesian products of relations as part of the evaluation plan cannot be encoded that way.

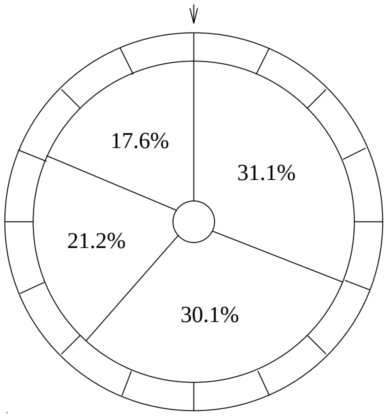
As an example of this encoding scheme, we represent the processing tree depicted in Figure 17b as a character string. In a preliminary step, every edge of the join graph is labelled by an arbitrary number, such as in Figure 17a. Then, the processing tree is encoded bottom-up and left-to-right, just the way as it would be evaluated. So, the first join of the tree joins relations R_1 and R_2, that is edge 1 of the join graph. In the next steps, R_{12} and R_3 are joined, then R_4 and R_5, and finally R_{123} and R_{45}, contributing edges 2, 4 and 3, respectively. Thus, the final encoding for our sample processing tree is "1243" (Figure 17c).

Selection The selection operator is used to separate good and bad solutions in the population. The motivation is to remove bad solutions and to increase the share of good solutions. Mimicking nature, selection is realized as shown in Figure 18. The sample population consists of four solutions, the objective function, cost, has to be minimized. The cost value for each of the solutions is listed in the table in Figure 18. Each solution is assigned a sector of size inverse proportional to its cost value on a biased roulette wheel. Four spins of the wheel might yield the result in the second table, where Solution 4 has not been selected—it "became extinct due to lack of adaptation."

This selection scheme is based on the fitness ratio of the members of the population: The better a member satisfies the objective function, the more it dominates the wheel, so one (relative) "super" population member may cause

Population

Nr.	String	Cost	Fraction of the Wheel
1	31542	31784	31.1%
2	45321	46924	30.1%
3	51234	174937	21.2%
4	14325	227776	17.6%
Sum			100.0%



Spinning the Wheel $\Longrightarrow$

New Generation

Nr.	String
1	31542
1	31542
2	45321
3	51234

Figure 18: Selection

the premature disappearance of other members' features. Those features may be valuable, even if the solution as a whole is not of high quality. To avoid this, selection can also be based on *ranking*. That means that not the value of the objective function itself but only its rank is used for biasing the selection wheel. In Figure 18, for instance, not the cost values would determine the fraction of the wheel a solution is assigned to, but just its rank value, i.e., 4 for Solution 1, 3 for Solution 2, 2 for Solution 3 and 1 for Solution 4.

General experience shows that ranking based selection usually makes the evolution process advance more slowly, but the risk of untimely losing important information contained in weaker solutions is much lower.

Another variant is to keep the best solution in any case. This strategy (sometimes referred to as "elitist") helps speeding up the convergence to the (near) optimal solution, because the risk of losing an already very good solution is eliminated.

Crossover The crossover operator is a means of combining partially good solutions in order to obtain a superior result. The realization of a crossover operator depends heavily on the chosen encoding. For instance, the crossover operator has to make sure that the characteristics of the particular encoding is not violated.

$3\boxed{154}2$ $\qquad\qquad$ $3\boxed{451}2$

$4\boxed{532}1$ $\qquad\qquad$ $4\boxed{352}1$

"Parents" $\qquad\qquad$ "Offspring"

Figure 19: Crossover 1 – Subsequence Exchange for Standard and Bushy Encoding

Such a characteristic is the uniqueness of each character in the string for the standard and the bushy encoding scheme. The crossover operator and the encoding scheme are tightly coupled, because often the implementation of a particular crossover operator is facilitated (or even made possible at all) if a particular encoding scheme is used. Basically, we employed two different crossover operators, namely *Subsequence Exchange* and *Subset Exchange*. They work as follows:

1. Subsequence Exchange (Standard and Bushy Encoding)
   An example of the application of this operator is shown in Figure 19. It assumes the standard or the bushy encoding scheme. For each of the two parents, a random subsequence is selected. The selected sequence is removed and the gap is filled with the characters of the other parent in the order of their appearance. For instance, in Figure 19, the subsequence "532" is selected from the string "45321". The first character of its offspring remains the same as in the parent (4). The second character is taken from the first character of the other parent (3). The second character of the other parent (1) cannot be used, because it is already present, so the third character of the offspring is taken from the third character of the other parent. Continuing this process yields at last the offspring "43521". Determining the second offspring is carried out similarly.

2. Subsequence Exchange (Salesman Encoding)
   This operator is a slight variation of the above. It is intended for use in conjunction with the Salesman Encoding. In contrast to the first version of the sequence exchange operator, the two subsequences that are selected in the two parents must be of equal length. These subsequences are then simply swapped. This is only feasible with the Salesman Encoding, because we do not have to worry about duplicated characters. Figure 20 shows a sample application of this operator.

3. Subset Exchange (For Standard and Bushy Encoding)
   The basic idea for this operator is to avoid any potential problems with duplicated characters by simply selecting two random subsequences with equal

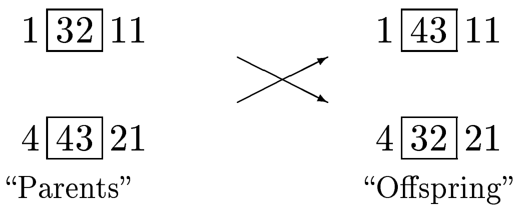


Figure 20: Crossover 2 – Subsequence Exchange for Salesman Encoding

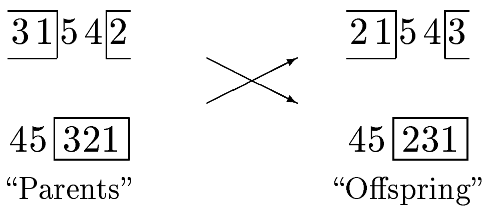


Figure 21: Crossover 3 – Subset Exchange

length in both parents that consist of the same set of characters. These two sequences are then simply swapped between the two parents in order to create two offspring. Figure 21 depicts an example of the application of this crossover operator.

Mutation The mutation operator is needed for introducing features that are not present in any member of the population. Mutation is carried out by random alteration of a randomly selected string. If the operator must not introduce duplicate characters, as in standard or bushy encoded strings, two random characters are simply swapped in order to carry out the mutation; with Salesman Encoding, a random character of the string is assigned a new, random value.

Due to the character of mutation as the "spice" of the evolution process, it must not be applied too liberally lest the process may be severely disrupted. Usually, only very few mutations are performed in one generation.

If the "elitist" variant of the selection operator is used, one might also consider to except the best solution in the population from being mutated. The reasons for doing so are explained in the paragraph above describing the selection operator.

The realization of the mutation operator is straightforward: it consists simply of randomly altering one or more characters of a (randomly from the population selected) string.

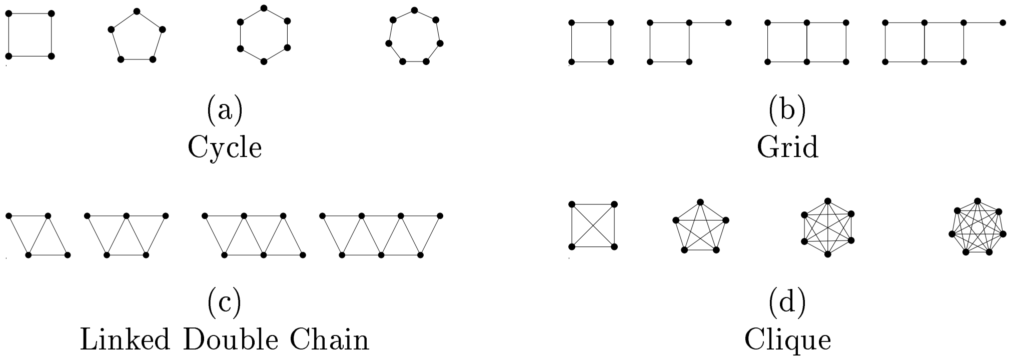


Figure 22: Join Graphs

5 Quantitative Analysis

Preliminaries The generation of queries for the benchmarks permits independent setting of the following parameters:

- Class of the join graph
- Distribution of relation cardinalities
- Attribute domains
- Cost model

The shape of the *join graph* can be chosen from the following four classes: cycle, grid, chain and clique (cf. Figure 22a–d, respectively). Disregarding cartesian products, these join graph classes provide from "cycle" to "clique" increasing connectivity, and, thus, increasing degree of freedom for the derivation of evaluation plans. In contrast to, e.g, [IK91], none of these comprise tree-graphs, as the optimal solution for these queries can be computed in polynomial time.

Relation cardinalities and *domain sizes* fall into four categories: S, M, L, XL as specified in Table 3; for instance, 16% of all relations comprise between 1000 and 10,000 tuples, and the domain size of 50% of all their attributes is between 100 and 1000.

The query itself is specified such that matching attribute names in different relations define an edge of the join graph with selectivity drawn randomly (uniform distribution) from the interval $\left[\frac{1}{|\operatorname{dom}(attribute)|}, \frac{1}{|\operatorname{dom}(attribute)|^{1/8}}\right]$. These figures admittedly do look somewhat arbitrary, but they worked well in practice.

Each point in the following diagrams represents the average of 200 optimized queries, which proved to be a good compromise between accuracy and running time, as preliminary tests showed.

Class	Relation Cardinality	Percentage
S	10–100	19%
M	100–1000	64%
L	1000–10.000	16%
XL	10.000–100.000	1%
		100%

(a)
Relation Cardinalities

Class	Domain Size	Percentage
S	2–10	5%
M	100–1000	50%
L	1000–10.000	30%
XL	10.000–100.000	15%
		100%

(b)
Domain Sizes

Table 3: Relation Cardinalities and Domain Sizes

An additional choice is the selection of the underlying *cost model*. In Section 2.2, several basic cost models are described. Since the performance of many algorithms depends heavily on the cost model, we tried to emphasize this effect by choosing two "extremes" for carrying out the benchmarks, namely the *nested loop cost model* (Section 2.2.2) and the *hash loop cost model* (Section 2.2.4). In the form as presented in Section 2.2.2, the nested loop model is symmetric—it does not make much of a difference if the two relations are exchanged. On the other hand, the hash loop model is highly asymmetric: Only the characteristics of one relation determine the cost of the join. Therefore, these two cost models are very well suited for focusing on the question of cost-model dependencies.

Benchmark Results for Deterministic Algorithms In the first series of benchmarks, we shall examine deterministic algorithms, i.e., the heuristics described in Section 4.1. In Figure 23, 24, 25 and 26, the benchmark results are depicted. "MinSel," "RDC," "TopDown" and "KBZ" refer to the Minimum Selectivity, Relational Difference Calculus, Top Down and Krishnamurthy-Boral-Zaniolo heuristic, respectively, as presented in Section 4.1. "Random" and "WorstCase" serve as the basis for comparison: a strategy that is chosen at random (can be interpreted as "performs no optimization at all") and the worst evaluation strategy possible. Since KBZ is not capable of generating bushy trees, all solutions are left-deep processing trees.

On each of the diagrams, the scaled costs (cost of optimized evaluation plan divided by the cost of the optimal evaluation plan) are plotted against the number of relations participating in the query. The underlying join graph type and cost model are noted on top of the diagram. For each of the two cost models, the connectivity of the join graph increases from "cycle" to "clique" (Figure 23 and 24 resp. Figure 25 and 26).

First, we notice that the basic goal of optimization ("avoiding the worst case") is easily achieved by each of the heuristics; in addition, all of them perform much better than a random evaluation strategy. The Minimum Selectivity heuristic

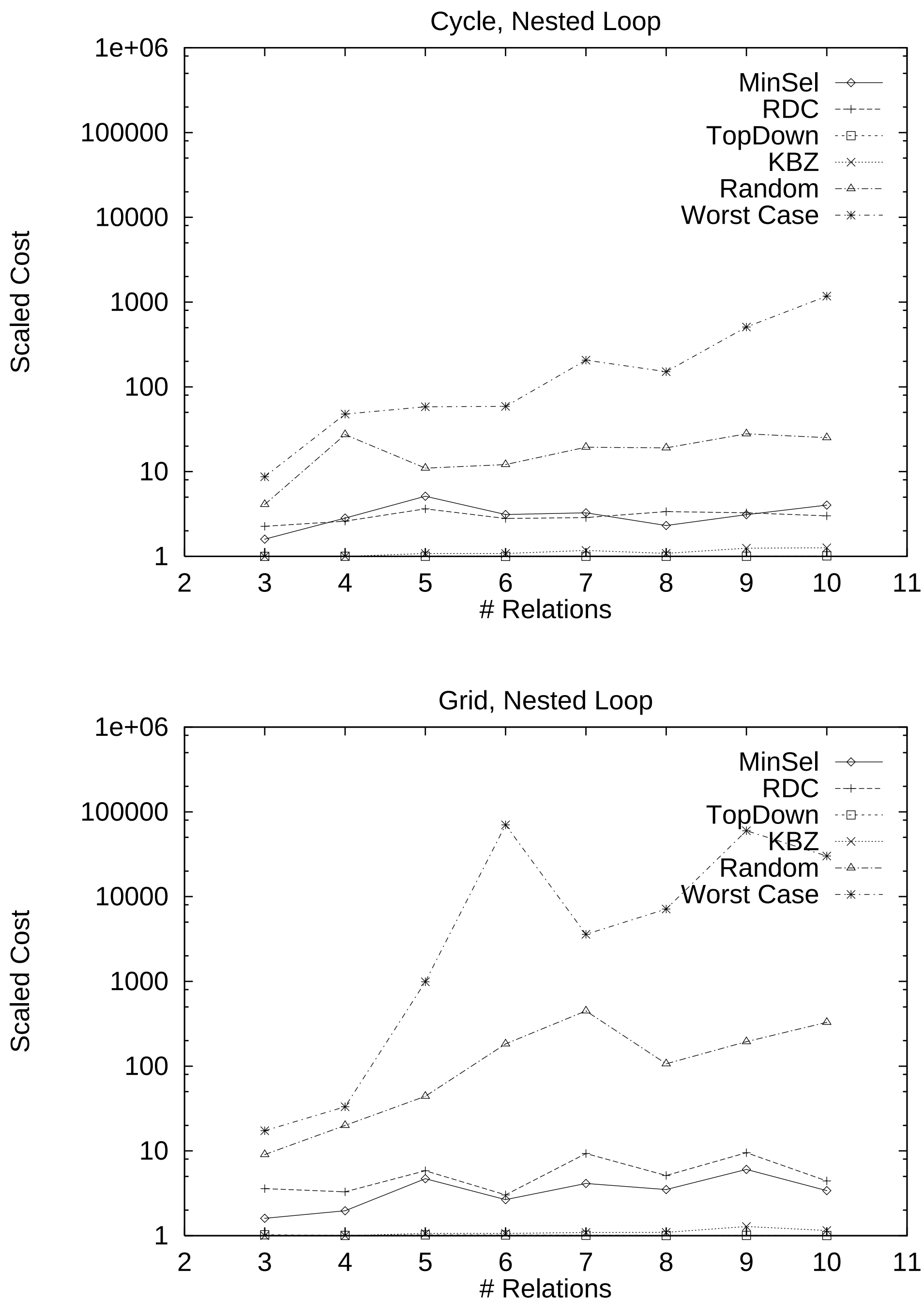
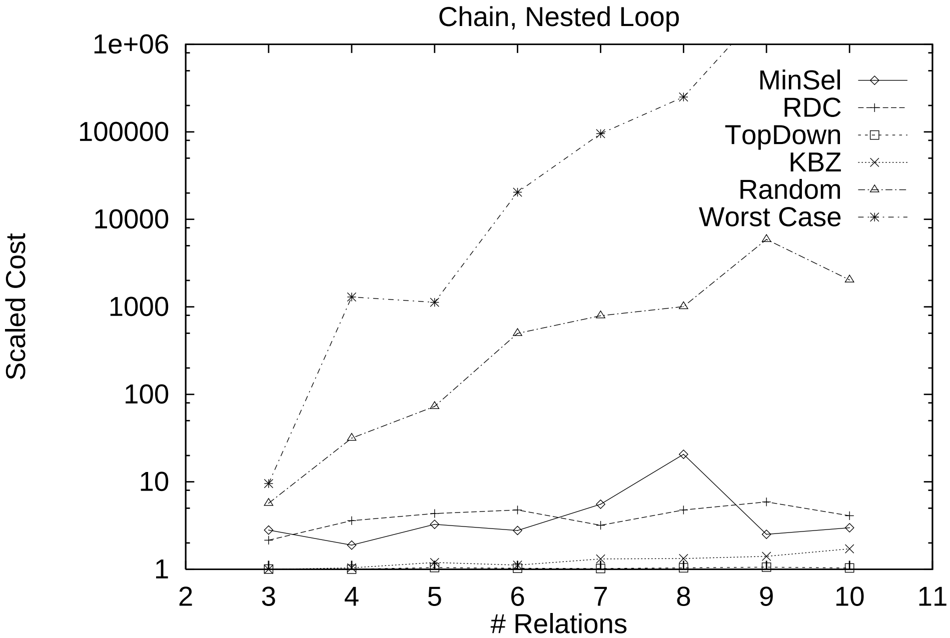


Figure 23: Deterministic Strategies 1: Nested Loop Cost Model



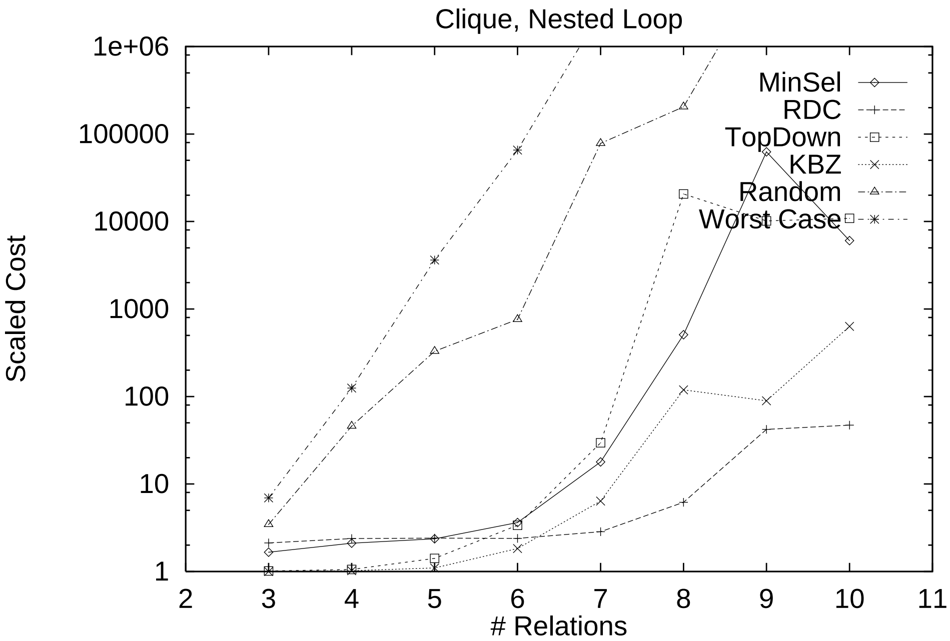


Figure 24: Deterministic Strategies 2: Nested Loop Cost Model

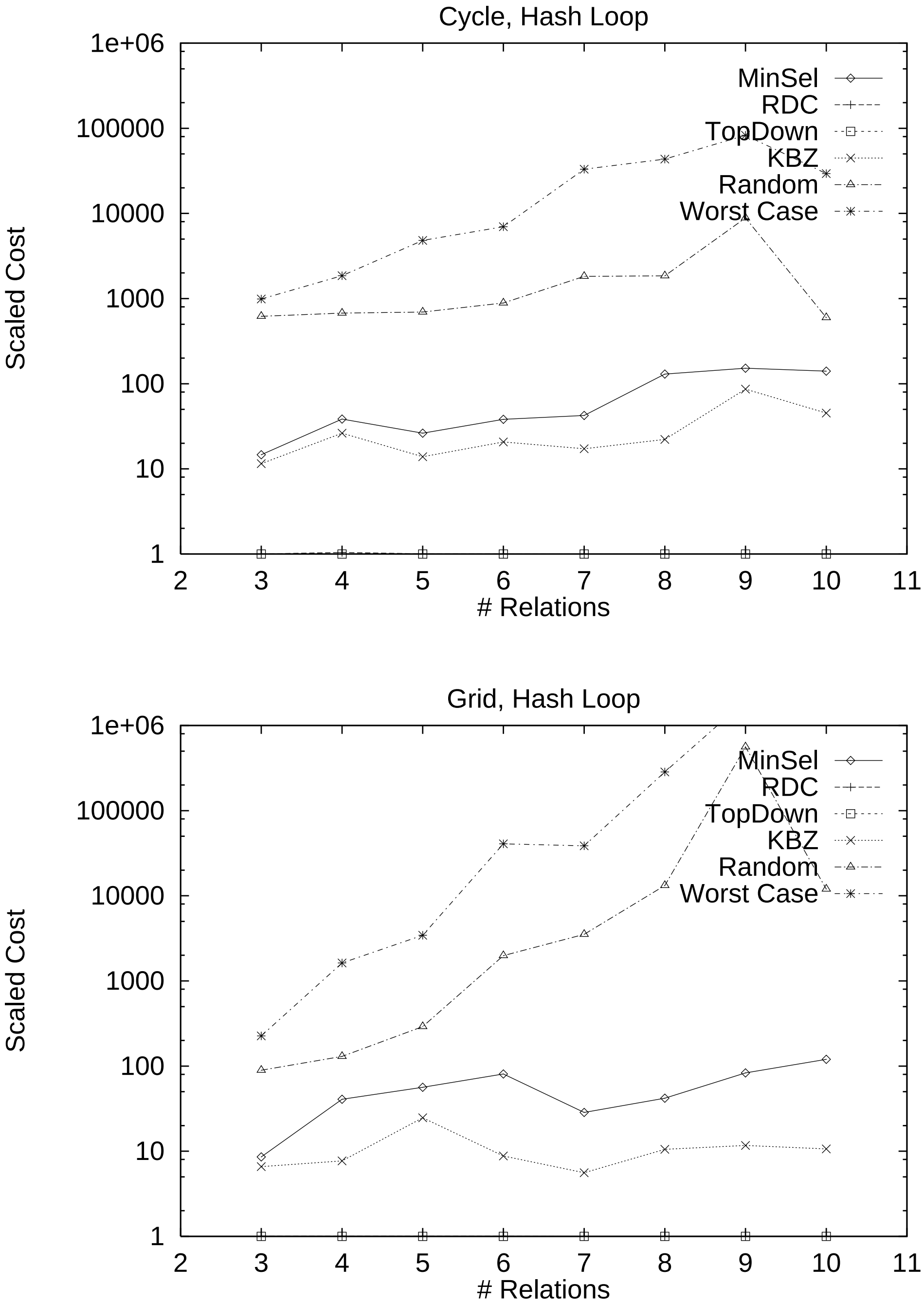
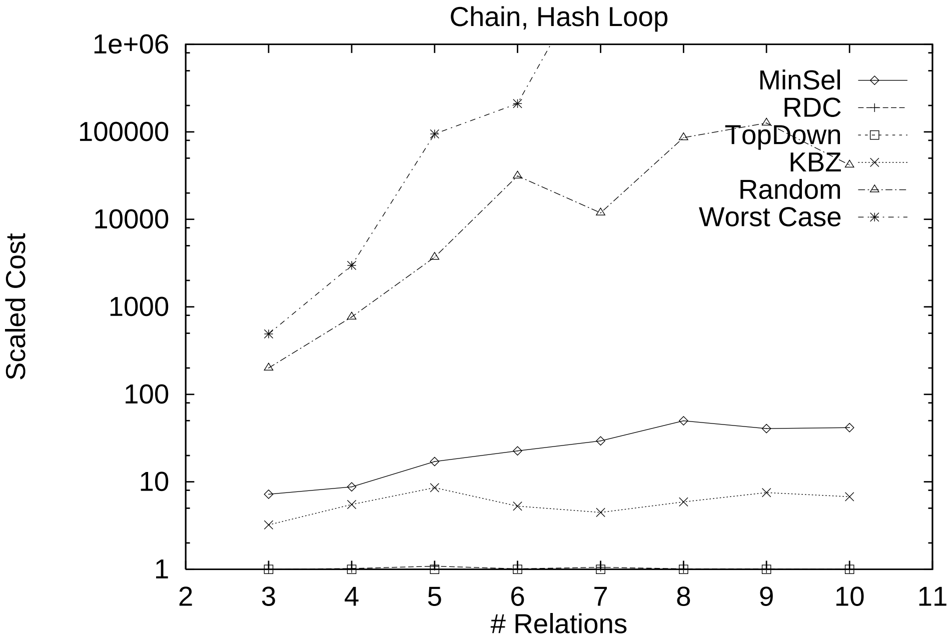


Figure 25: Deterministic Strategies 3: Hash Loop Cost Model



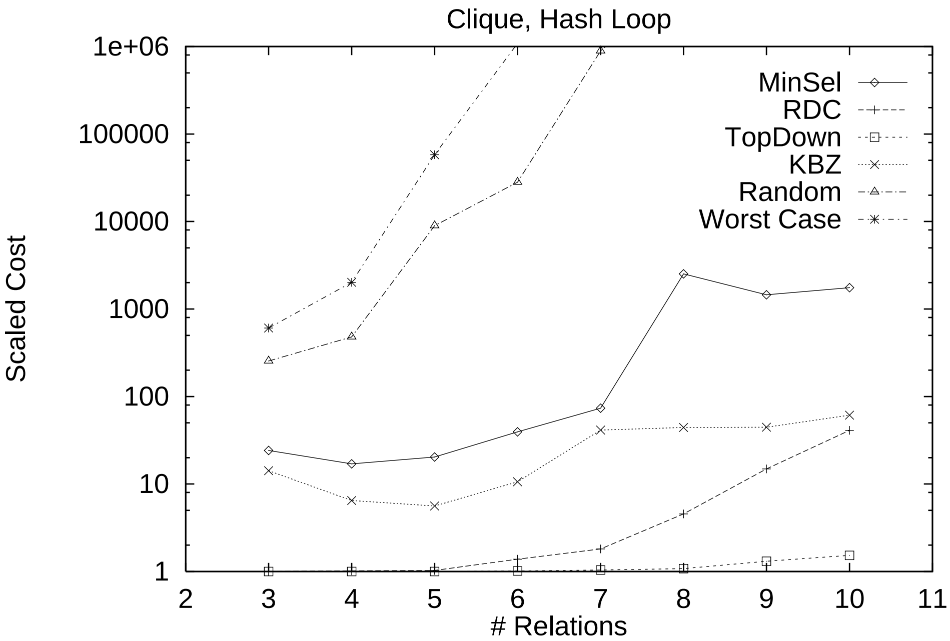


Figure 26: Deterministic Strategies 4: Hash Loop Cost Model

performs rather well for low connectivities of the join graph, as long the the nested loop cost model is considered. For higher connectivities, such as for "chain" and "clique" join graphs and for the asymmetric hash loop cost model, the heuristic ends up on the lowest rank.

Although the Relational Difference Calculus heuristic does not, just like MinSel, take the particular cost model into account, the overall performance of this heuristic is very good. Especially the asymmetric hash loop cost model "suits" very well (results are close to the optimum) and even for high connectivities, RDC yields good results.

The Top-Down heuristic does take the cost model into account, and performs very well for nearly all join graph/cost model combinations, where an (almost) optimal left-deep processing tree can be found. Only the clique join graph in conjunction with the nested loop cost model poses a very hard problem to this heuristic.

The last heuristic plotted in the diagrams is the KBZ heuristic. It is insofar special as KBZ is capable of computing the optimal left-deep solution for tree graphs. But as all considered join graphs are cyclic, in a first step a tree has to be derived by removing some edges of the join graph. As suggested in [KBZ86], we compute the minimum spanning tree (minimum of join selectivity product) of the join graph before applying the KBZ algorithm. However, we note that this approach works very well for low connectivity join graphs, but it fails spectacularly for the completely connected clique graph.

To summarize the figures, we note that RDC and TopDown yield the best results. Both heuristics generate solutions that are in many cases close to the optimum. However, for highly connected join graphs either TopDown (Clique, Nested Loop) or RDC (Clique, Hash Loop) yield bad results, where RDC's "failure" is moderate compared to TopDown. These findings suggest an extension to heuristic optimizers that adapts the particular optimization method to the shape of the join graph and the join method that is to be employed. The low running time of heuristic optimizers makes even a combination feasible where the best result from several optimization algorithms is used. However, if only one algorithm can be implemented, the choice is clearly in favour of the RDC optimizer because of its superior overall performance with least variance over the different join graph types and cost models.

Recently, another heuristic algorithm that is based on KBZ has been proposed [SI93], the AB algorithm. The working principle comprises not only the computation of a join order, but also the incorporation of different join algorithms. Therefore, within the framework of our benchmarks, we could only implement a simplified version of this algorithm that does not take different join methods into account. A sample of the results achieved with this simplified AB algorithm is shown in Figure 27. The algorithm generates left-deep processing trees. However, the figures should be taken with a grain of salt, as the choice of join methods is a major part of the AB algorithm; one could expect a consider-

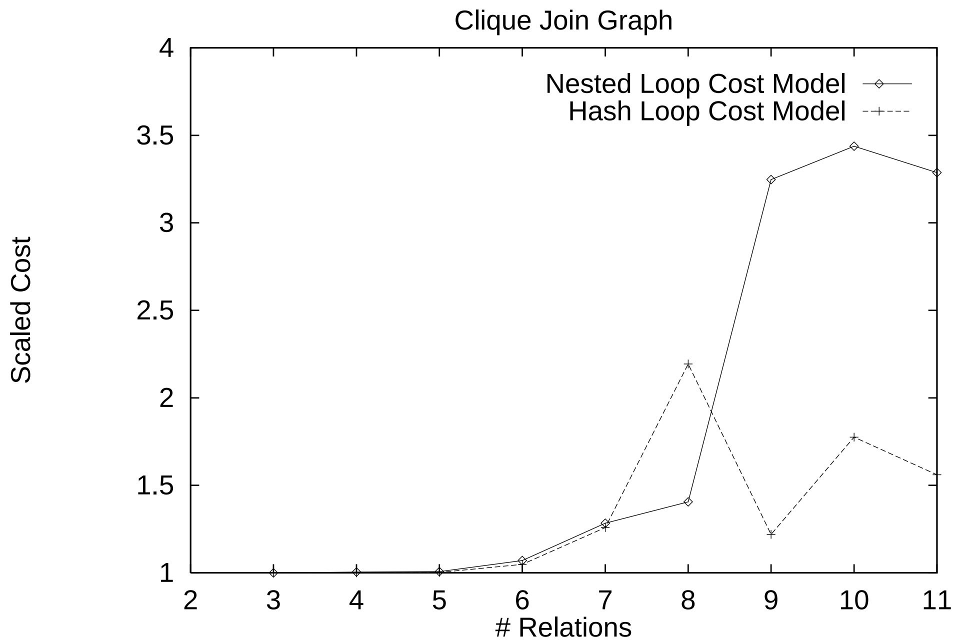


Figure 27: AB Algorithm

ably higher performance for the genuine algorithm, as the experimental results in the cited reference indicate.

Benchmark Results for Randomized and Genetic Algorithms The next set of benchmarks is carried out with randomized algorithms (cf. Section 4.2) and genetic algorithms (cf. Section 4.3). We compare two variants of Iterative Improvement (called IIJ and IIH [SG88]), three variants of Simulated Annealing (called SAJ, SAH [SG88], SAIO [IW87]) and two variants of genetic algorithms (Gen, GenBushy). The parameters for each algorithm are derived from the cited references (II, SA) or they were determined in preliminary tests (Gen, GenBushy). Exactly as in the first set of benchmarks with the heuristic algorithms, the scaled cost is plotted against the number of relations participating in the join. The parameters of the algorithms mentioned above are as follows:

1. SAJ

   - A move is either a "Swap" or a "3Cycle," i.e., only left-deep processing trees are considered.
   - The starting temperature is chosen such that at least 40% of all moves are accepted.

- The number of iterations of the inner loop is the same as the number of participating relations.
- After every iteration of the inner loop, the temperature is reduced to 97.5% of its old value.
- The system is considered frozen when the best solution encountered so far could not be improved in five subsequent iterations and less than two percent of the generated moves were accepted.

2. SAH

   - A move is either a "Swap" or a "3Cycle," i.e., only left-deep processing trees are considered.
   - The starting temperature is determined as follows: the standard deviation σ for the cost is estimated from a set of sample solutions and multiplied by a constant value (20).
   - The number of iterations of the inner loop is the same as the number of participating relations.
   - After every iteration of the inner loop, the temperature is multiplied by $\max(0.5, e^{-\frac{\lambda t}{\sigma}})$ ($\lambda = 0.7$, σ see above).
   - The system is considered frozen when the difference between the minimum and maximum costs among the accepted states at the current temperature is equals the maximum change in cost in any accepted move at the current temperature.

3. SAIO

   - Moves are chosen from "Swap," "Associativity," "Left Join Exchange" and "Right Join Exchange." The entire solution space (bushy processing trees) is considered.
   - The starting temperature is twice the cost of the starting state.
   - The number of iterations of the inner loop is four times the number of participating relations.
   - After every iteration of the inner loop, the temperature is reduced to 95% of its old value.
   - The system is considered frozen when the best solution encoutered so far could not be improved in four subsequent moves and the temperature has fallen below one.

4. Iterative Improvement (IIH, IIJ, IIIO)

- In order to perform a "fair" comparison between Iterative Improvement and Simulated Annealing, the number of iterations of the respective outer loops (cf. Figure 14 and 15) is the same. For instance, the number of iterations of the outer loop of IIH and SAH is equal for equal queries.

5. Two-Phase Optimization (IIIO+SAIO)

   - Ten iterations for the II phase
   - SA phase starts with the minimum from the II phase and the starting temperature is 0.1 times its cost.

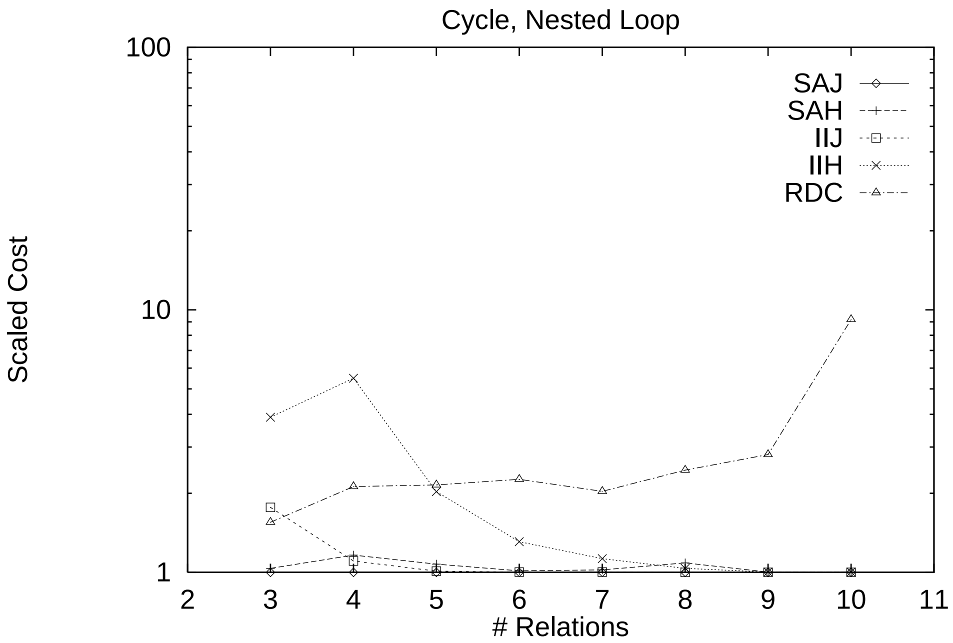
6. Genetic Algorithms (Gen/GenBushy)

   - Solution space: Left-deep processing trees/
bushy processing trees
   - Encoding: Ordered list of leaves/
Bushy Encoding
   - Ranking-based selection operator
   - Sequence exchange crossover operator
   - Population: 60
   - Crossover rate 60% (60% of all members of the population participate in crossover)
   - Mutation rate 1% (1% of all solutions are subject to random mutation)

In Figure 28 and 29, the performance of Iterative Improvement and Simulated Annealing is depicted. For comparison, the result for the RDC heuristic—the overall best deterministic heuristic—is plotted on each diagram. Generally, the deterministic heuristics like RDC generate worse processing trees than II and SA, except for the cycle join graph and the hash loop cost model. Comparing Figures 23–26 and 28–29, we note that the processing trees derived by randomized algorithms incur costs that are about an order of magnitude lower than the costs of solutions of deterministic algorithms.

On the upper diagram, in Figure 28, SAH and IIH show an acceptable performance, but the solutions generated for the pair clique/nested loop (lower diagram) are rather bad. This phenomenon is also apparent in Figure 29, where the hash loop cost model is applied: for the cycle query graph, IIH generates very bad solutions, especially for low numbers of relations. SAH works slightly better, but the clique query graph cannot be managed by SAH either.

The reason for this behaviour is the (too) low number of iterations that IIH/SAH perform—for instance, only one iteration in a query where three relations participate. The only remedy would be a modification of SAH's termination condition.



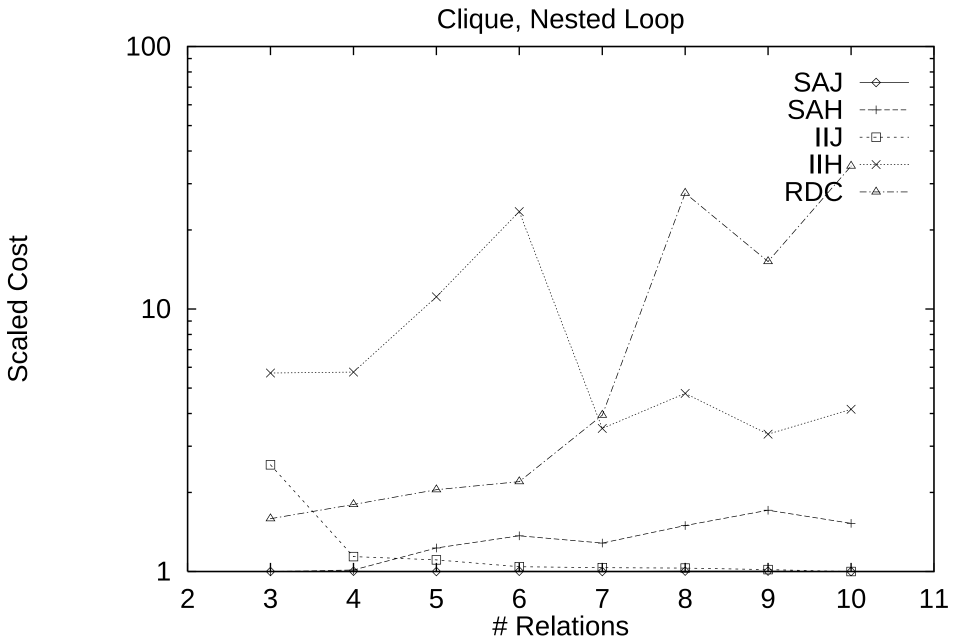


Figure 28: Randomized Algorithms 1: Nested Loop Cost Model

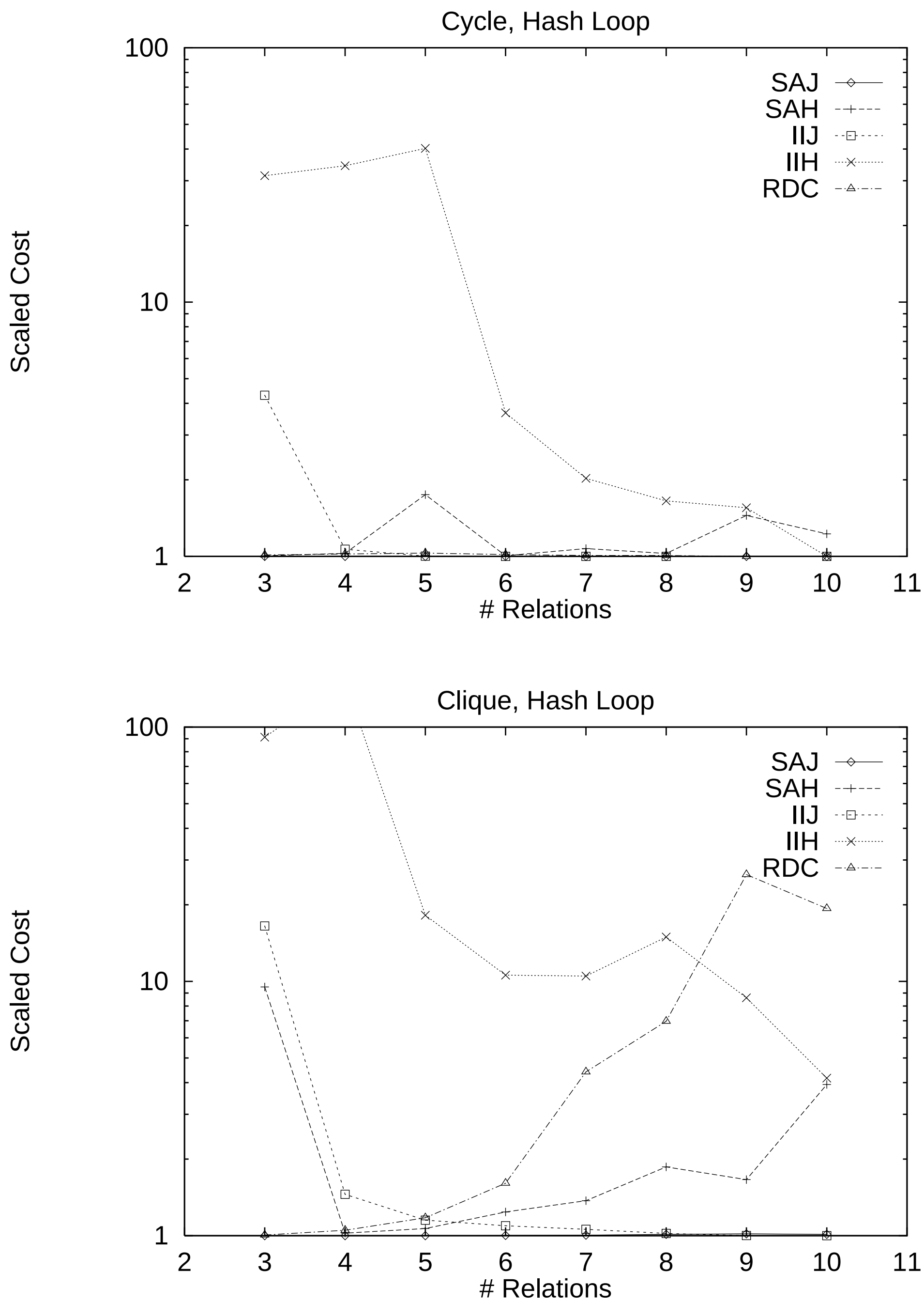


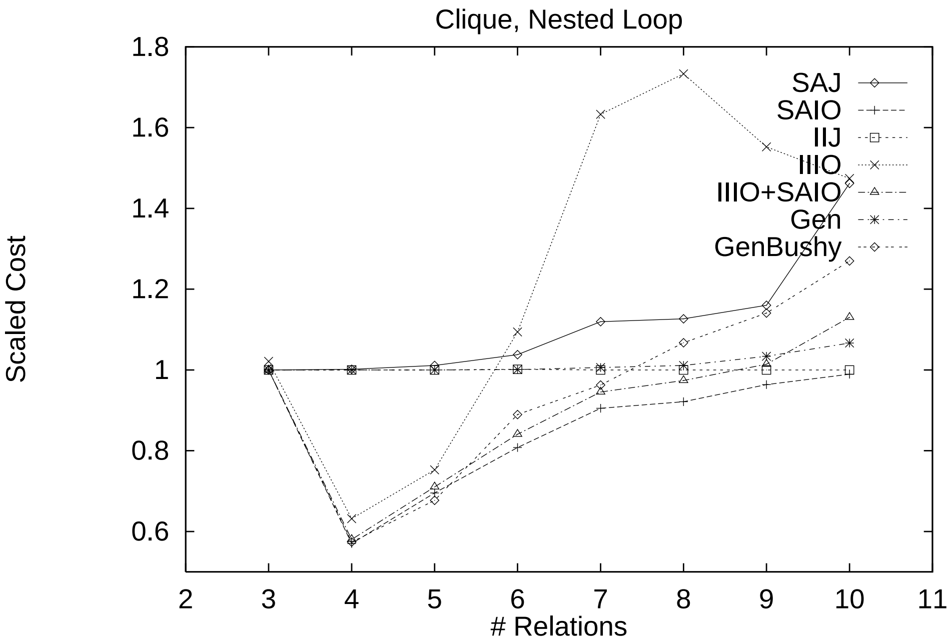
Figure 29: Randomized Algorithms 2: Hash Loop Cost Model

On the other hand, IIJ and SAJ perform many more iterations, and the diagrams show that the effort is well invested: IIJ as well as SAJ find near optimal solutions, no matter what kind of query graph or cost model is used. Obviously, SAJ's termination condition is much more suitable than SAH's for avoiding plateaux and sub-optimal minima in the solution space.

Next, we examine the impact of an extended solution space, namely the space of general (bushy) processing trees as opposed to left-deep processing trees, as before. As we showed in Section 3, from a purely statistical point of view the bushy tree solution space is preferable for the hash loop cost model. However, the *shape* of the solution space is an important point for algorithms that perform some kind of "walk" through the space; a "well" is the most preferable [IK91]. For the completely connected clique graph, Figure 30 shows the performance of the best randomized left-deep optimizers (SAJ resp. IIJ), randomized optimizers for bushy trees (SAIO, IIIO) and genetic optimizers for left-deep (Gen) and bushy trees (GenBushy). Costs are scaled with respect to the left-deep optimum, so it is possible for bushy tree optimizers to achieve results better than 1.

In Figure 30, the quality of all bushy tree optimizers' solutions decreases with increasing number of relations and approaches the left-deep optimum. Obviously, the exploding cardinality of the bushy tree solution space makes it very difficult for the optimizers to find good solutions. Even the best algorithm, SAIO, is on an average for ten relations only slightly better than the left-deep optimum. Surprisingly, the combination IIIO+SAIO (denoted "2PO" for "Two-Phase Optimization" in [IK90]) cannot beat SAIO for either cost model. This is a difference to the results in [IK90], where 2PO performed best. The reason for this result is the different type of join graph that we used in our experiments (not acyclic as in [IK90]). We can deduce from this result that the shape of the solution space for those join graphs that we used is different from the solution space of tree join graphs. Minima must be further apart, (i.e., more evenly distributed throughout the solution space), so the combination IIIO+SAIO that searches only the close proximity of the best IIIO solution cannot obtain solutions that are as good as those from SAIO by itself with a higher starting temperature.

Both variants of the genetic optimizer (Gen and GenBushy) show a very stable performance that depends directly on the number of participating relations (and, thus, the cardinality of the solution space). Because a "shape" of the solution space is not defined for this type of algorithm (there is no notion of a "neighbouring solution"), we can concentrate solely on its statistical characteristics. Unfortunately, the optimization potential, especially for the hash loop cost model, is not fully exploited. According to the analysis of the solution space for the clique join graph and the hash loop cost model (cf. Figure 1b on Page 7), the bushy tree solution space ought to be preferable, but the genetic algorithm (GenBushy) fails to take advantage of that; the version that generates left-deep trees (Gen) is superior. For the nested loop cost model, all algorithms perform worse (cf. upper part of Figure 30), except SAIO. These results indicate that the



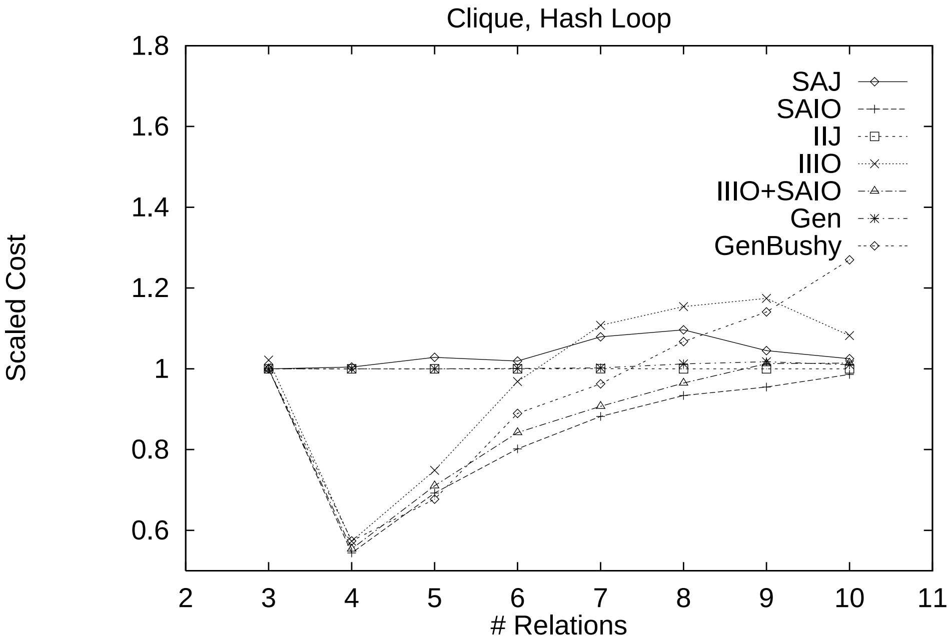


Figure 30: Randomized/Genetic Algorithms: Bushy vs. Left-Deep Processing Trees

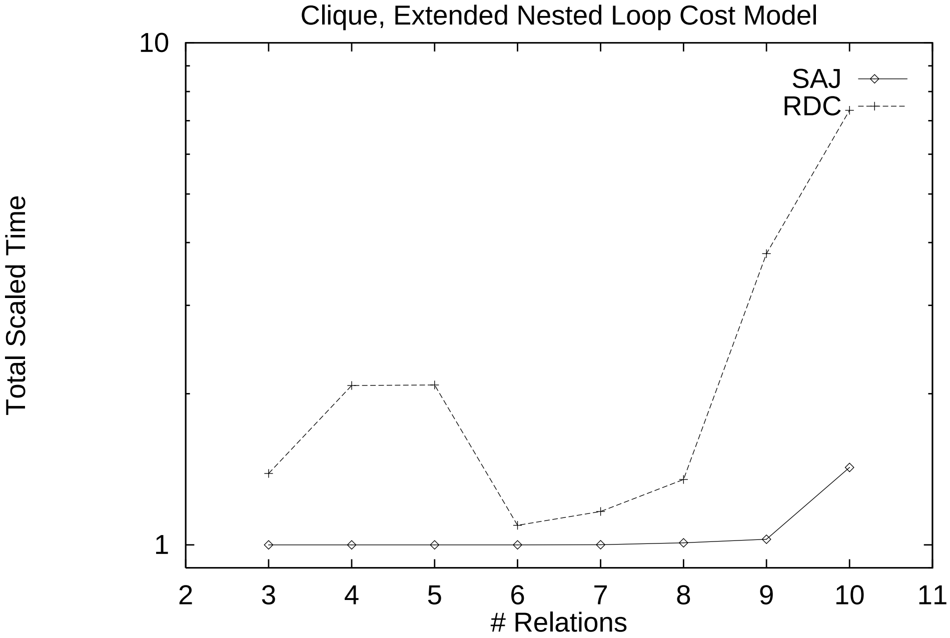


Figure 31: Total Running Time: SAJ vs. RDC

bushy tree solution space is not necessarily the ideal choice, especially for highly connected join graphs.

Another important criterion is the running time of the optimization. We have already mentioned that deterministic (heuristic) algorithms, although needing only little running time, yield solutions that are frequently inferior in comparison to most randomized algorithms: the total time (optimization plus evaluation of the solution) may be much higher than for, e.g., Simulated Annealing. In Figure 31, the total scaled time for RDC and SAJ optimization/evaluation for the clique join graph is plotted against a varying number of relations. The extended nested loop cost model as described in Section 2.2.5 is used. The factor "$y = 1$" corresponds to the evaluation time for the optimal left-deep processing tree solution.

Although SAJ runs much longer (about an order of magnitude) the investment in optimization pays: The total time is up to five times higher than for the short-running RDC optimization. Hence, the value of an optimization strategy should first be weighed according to the quality of the solutions generated; the running time of the optimization is less important if, as in our benchmarks, the cardinality of the solution is high.

However, if differences in the performance of several optimization algorithms are not too great, the choice depends on the running time. The upper part of

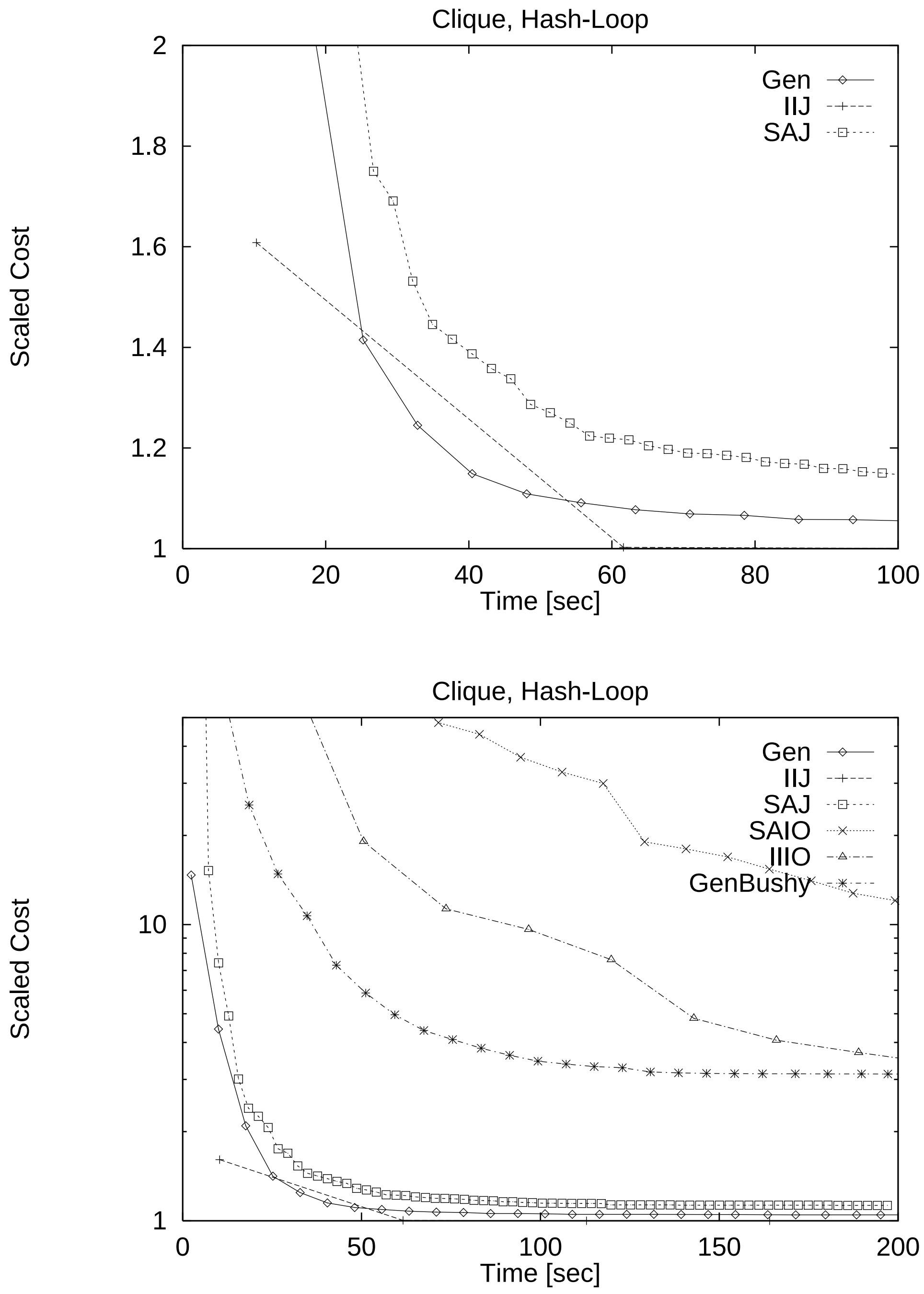


Figure 32: Randomized/Genetic Algorithms: Convergence

Figure 32 shows the convergence of three left-deep tree optimizers to the optimal solution, namely the genetic algorithm (Gen), Iterative Improvement (IIJ) and Simulated Annealing (SAJ). Plotted are the average scaled costs of the solution generated after a certain elapsed time. For both plots in Figure 32, join expressions consisting of ten relations were optimized, based on the clique join graph and the hash loop cost model. Thus, the plots correspond to the lower part of Figure 30.

We note that Iterative Improvement approaches the optimum very quickly: after about 60 seconds, the intermediate solution is little worse than the optimum. The improvement over time is slower for the genetic algorithm, but the figure shows that eventually a result is achieved that is also very close to the optimum. The worst performance must be noted for Simulated Annealing (SAJ), although the difference is not too great.

In the lower part of Figure 32, algorithms for bushy trees are included as well. In order to capture a wide range of different optimizers, the y-axis is now scaled logarithmically. $y = 1$ corresponds, exactly as in the upper part, to the left-deep optimum—none of the algorithms is capable of yielding a better solution, at least not within the first 200 seconds of its run. SAIO, the best performer with respect to the quality of the solution, is after 200 seconds still as bad as ten times the left-deep optimum. The other two optimizers for bushy trees make better progress, but even they (IIIO, GenBushy) cannot achieve the results of the left-deep optimizers. Considering the fact that for a higher number of relations the bushy tree optimizers (cf. Figure 30) generate solutions that are hardly better than the left-deep optimum (at least for highly connected join graphs), the left-deep optimizers are serious competitors.

There is a further point in favour of randomized and genetic algorithms, apart from the good performance: both are "scalable," since they can be extended to optimize complete queries, and Iterative Improvement and genetic algorithms can easily make use of parallel computer architectures due to their working principle. Hence, the flexibility and performance of these algorithms make them the strategies of choice.

6 Conclusion

We have studied several algorithms for the optimization of join expressions. Due to new database applications, the complexity of the optimization task has increased; more relations participate in join expressions than in traditional relational database queries. Enumeration of all possible evaluation plans is no longer feasible. Algorithms that compute approximative solutions, namely heuristic, randomized and genetic algorithms, show different capabilities for solving the optimization task. Heuristic algorithms compute solutions very quickly, but the evaluation plans are in many cases far from the optimum. Randomized and genetic algorithms are better suited for join optimizations; although they require a longer running time, the solutions are mostly near optimal. Since the overall time, i.e., optimization plus evaluation, is likely to be dominated by the evaluation part, randomized and genetic algorithms are generally preferable.

For the question of the adequate solution space, we have found that for highly connected join graphs the space of left-deep trees is preferable since bushy tree optimizers are generally not capable of exploiting the inherent advantage of the bigger solution space with potentially better solutions. This is especially true for join expressions with a higher number of participating relations.

Another consideration is the extensibility of randomized and genetic algorithms: both can be designed to optimize not merely pure join expressions, but complete relational queries. In addition, some of them (namely Iterative Improvement and genetic algorithms) can be easily modified to make use of parallel computer architectures.

Acknowledgements This work was partially supported by grant DFG Ke 401/6-1 and SFB 346. We thank A. Eickler, S. Helmer and B. König-Ries for implementing the optimization algorithms and for extensive benchmarking.

References

[BFI91] K. Bennet, M. C. Ferris, and Y. E. Ioannidis. A genetic algorithm for database query optimization. Computer Sciences Technical Report 1004, University of Wisconsin-Madison, February 1991.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison Wesley, Reading, MA, 1989.

[IK84] T. Ibaraki and T. Kameda. Optimal nesting for computing N-relational joins. *ACM Trans. on Database Systems*, 9(3):482–502, 1984.

[IK90] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 312–321, Atlantic City, USA, May 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. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 168–177, Denver, USA, May 1991.

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

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of non-recursive queries. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 128–137, Kyoto, Japan, 1986.

[KMS92] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing Boolean expressions in object bases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 79–90, Vancouver, B.C., Canada, Aug 1992.

[Kru56] J. B. Kruskal. On the shortest spanning subtree of a graph and the travelling salesman problem. *Proc. of the Amer. Math. Soc.*, 7:48–50, 1956.

[ME92] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, Mar 1992.

[MHKR93] G. Moerkotte, S. Helmer, and B. König-Ries. The relational difference calculus. unpublished manuscript, Universität Karlsruhe, 1993.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, Jun 1979.

[Sch89] W. G. Schneeweis. *Boolean Functions with Engineering Applications and Computer Programs.* Springer, 1989.

[SG88] A. Swami and A. Gupta. Optimization of large join queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 8–17, Chicago, USA, 1988.

[SI93] A. Swami and B. Iyer. A polynomial time algorithm for optimizing join queries. In *Proc. IEEE Conf. on Data Engineering*, pages 345–354, Vienna, 1993.

[Swa89] A. Swami. Optimization of large join queries: Combining heuristics and combinational techniques. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 367–376, Portland, USA, May 1989.

[Ull88] J. D. Ullman. *Principles of Data and Knowledge Bases*, volume I and II. Computer Science Press, Woodland Hills, CA, 1988.