Heuristic and Randomized Optimization for the Join Ordering Problem

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

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

[+]Universität Mannheim
Lehrstuhl für Praktische
Informatik III
68131 Mannheim, Germany
moer@pi3.informatik.uni-mannheim.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 approximate 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 only advisable for certain join graph types. 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, there are generally only few relations involved. The optimization of these 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 query-generating database system frontends and complex views. In both cases, very complex queries may be issued without the end user being aware of that fact. Even in object-oriented database systems [KM94], complex join expressions may be encountered: while forward traversal of object references are usually very well supported by specialized access mechanisms and would not be treated as ordinary join operations, this is not true for *backward* traversal. This would require appropriate index structures such as Access Support Relations [KM92], processing of which, in turn, involves handling of potentially very complex join expressions for both initial materialization and maintenance.

Hence, there is a demand for optimization techniques that can cope with such complex queries 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 evaluation strategies for join expressions has been addressed from the development of the first relational database systems [WY76, YW79, 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 relational database systems where queries are stated interactively, 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 generally NP-hard problem of determining the optimal order [IK84] cannot be solved exactly anymore. We have to rely on algorithms that compute (hopefully) good approximate 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 on whether the tuple is to be included in the result or not. The join selectivity is the ratio "number of tuples in the result/number of tuples in the cartesian product". 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 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 approximate 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 pages that have to be read from or written to secondary memory. The execution environment is not distributed. The database is assumed to be much larger than available main memory, so all costs besides I/O can be neglected without introducing too large an error. All cost models are based on parameters listed in Table 1. The join operations themselves are equijoins. A common term for each of the cost formulae below is the cost for writing the result of the join operation to secondary memory.

Parameter	Meaning
$\|R\|$	Cardinality (number of tuples) of relation R
ts_R	Tuple size of relation R (in bytes)
bs	Size of a disk block (in bytes)
ps	Size of a tuple reference (tuple identifier, TID)
ms	Main memory size (in bs units)
σ_{12}	Join selectivity for join $R_1 \bowtie R_2$ $\left(\sigma_{12} = \frac{\|R_1 \bowtie R_2\|}{\|R_1 \times R_2\|}\right)$
b_R	Number of blocks occupied by relation R
fo	Fanout of an internal B$^+$-tree node ($fo = \lfloor 0.69 \cdot bs/(ks + pr) \rfloor$) ($ks$ = key size, pr = size of a page reference)
x_R	Height of a B$^+$-tree index on the join attribute of R minus one $\left(x_R = \left\lceil \log_{fo} b_R \right\rceil - 1,\ \text{assuming } ps \approx ks + pr\right)$
s_R	Selection cardinality of R's join attribute (average number of tuples with the same value of the join attribute)

Table 1: Cost Model Parameters

This cost is

$$C_{write}(R_1 \bowtie R_2) = \frac{\sigma_{12} \cdot |R_1| \cdot |R_2|}{bs/ts_{R_{12}}}$$

2.2.1 Nested Loop Join

The cost for performing a nested loop join (depending on the presence of index structures) is [EN94]:

1. Without index support

$$C_{nl}(R_1 \bowtie R_2) = \underbrace{b_{R_1}}_{\text{read } R_1} + \underbrace{\left(\left\lceil \frac{b_{R_1}}{ms - 1} \right\rceil \cdot b_{R_2}\right)}_{\text{read } R_2 \text{ and perform join}}$$

2. Primary B$^+$-tree index on the join attribute of R_2

$$C_{nl}(R_1 \bowtie R_2) = \underbrace{b_{R_1}}_{\text{read } R_1} + \underbrace{|R_1| \cdot (x_{R_2} + 1)}_{\text{use index to find matching tuple in R}_2}$$

3. Secondary B$^+$-tree index on the join attribute of R_2

$$C_{nl}(R_1 \bowtie R_2) = \underbrace{b_{R_1}}_{\text{read } R_1} + \underbrace{|R_2| \cdot (x_{R_2} + s_{R_2})}_{\text{use index to find matching tuple in R}_2}$$

4. Hash index on the join attribute of R_2

$$C_{nl}(R_1 \bowtie R_2) = \underbrace{b_{R_1}}_{\text{read } R_1} + \underbrace{|R_1| \cdot h}_{\text{use index to find matching tuple in } R_2}$$

h is the average number of page accesses necessary to retrieve a tuple from R_2 with a given key. We use the value $h = 1.2$ for a primary hash index, and $h = 2.2$ for a secondary hash index.

2.2.2 Sort-Merge Join

The cost for performing a sort-merge join operation is [EN94]:

$$C_{sm}(R_1 \bowtie R_2) = C_{R_1} + C_{R_2}$$

where C_{R_1} (resp. C_{R_2}) is computed according to the following cases:

1. The relation is sorted on the join attribute (or there is a primary B$^+$-tree index on the join attribute)

$$C_{R_x} = b_{R_x}$$

i.e., there is only the cost for reading the relation.

2. There is a secondary B$^+$-tree index on the join attribute

$$C_{R_x} = \left\lceil |R_x| \cdot \frac{ps}{0.69 \cdot bs} \right\rceil + b_{R_x}$$

i.e., the leaf nodes of the index tree (assumed to be 69% full) have to be scanned for pointers to the tuples of the relation, and the blocks containing the tuples itself must be read at least once.

3. No sort order on the join attribute, explicit sorting is required

$$C_{R_x} = b_{R_x} \log_{ms} b_{R_x} + b_{R_x}$$

We assume the merge-sort algorithm is applied, where the number of merge passes depends on the amount of main memory available.

2.2.3 Hash Join

We assume that a "Hybrid Hash Join" is carried out. This algorithm performs very well over a large range of available main memory. The cost is [Sha86]:

$$b_{R_1} + b_{R_2} + 2 \cdot (b_{R_1} + b_{R_2}) \cdot (1 - q)$$

where q denotes the fraction of R_1 whose hash table fits into main memory. It is computed as:

$$q = \frac{ms - \left\lceil \frac{1.4 \cdot b_{R_1} - ms}{ms - 1} \right\rceil}{b_{R_1}}$$

The constant 1.4 accounts for the hash table's load factor of about 71%.

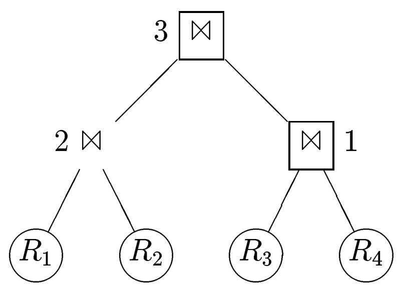


Figure 1: Computation of processing tree costs

2.2.4 Cost of an Entire Processing Tree

In order to estimate the cost for evaluating an entire processing tree, the cost for each node is computed recursively (bottom-up, right-to-left) 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.

If the outer relation (R_1 in the cost formulae) is not a base relation, and the join algorithm is a nested loop join, we assume that pipelining is possible, which saves the costs for writing an intermediate result to disk and to read it back into main memory. For instance, the processing tree in Figure 1 (where all join nodes are supposed to be nested loop joins) is evaluated as follows:

1. The join operation $R_3 \bowtie R_4$ (node 1) is performed. Because both operands are base relations, the cost for reading both R_3 and R_4 is included in the estimate. Furthermore, the result of this operation has to be written to disk as intermediate relation.

2. The join operation $R_1 \bowtie R_2$ (node 2) is performed. Again, both operands are base relations, so the cost for scanning them has to be counted. But in contrast to node 1, no intermediate result has to be written to disk, because the tuples can be pipelined to node 3, the root of the processing tree.

3. In node 3, both intermediate results $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$ are joined together in order to compute the final result. While $R_1 \bowtie R_2$ does not need to be read from disk due to the pipeline from node 2, $R_3 \bowtie R_4$ must be read back in, and the final result must be written to disk.

We note that the boxed join nodes' results (Figure 1) must be written to secondary memory. These considerations are valid if (and only if) the two processing nodes in question are both nested loop joins. If either node 2 or node 3 in the example tree in Figure 1 were anything else but nested loop joins, the cost for writing the intermediate result to disk and reading it back into memory would have to be charged.

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, which are mainly of interest in distributed computing environments (cf., e.g., [LVZ93]).

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, because such trees are capable of exploiting the cost-reducing pipelining technique on each of its join processing nodes. In case a processing tree consists solely of nested loop joins (either with or without index support), not a single intermediate result has to be materialized on secondary memory.

3.2 Bushy Trees

In this solution space, we also permit join nodes where both operands are "composites" (i.e., no base relations). Thus, the solutions in this space are in no way restricted. Consequently, this solution space includes left-deep as well as other special tree shapes as (strict) subsets. Because the shape of possible processing trees can be arbitrary, the cardinality of this set is much higher than the cardinality of the left-deep space: For n base relations, there are $\binom{2(n-1)}{n-1}(n-1)!$ different solutions. However, although the degrees of freedom in constructing bushy trees are much higher, the capability of exploiting the pipelining technique is restricted to a subset of the tree's join processing nodes. The more the shape of the tree tends toward right-deep (i.e., the join nodes' *left* operands are base relations), the smaller is the size of this subset. For a right-deep tree, none of its join processing nodes is capable of pipelining.

In [OL90], an adaptable plan enumeration strategy for linear (chain) and star-shaped join graphs is proposed that reduces the number of plans whose costs have to be evaluated considerably. If n denotes the number of relations in the join graph, there are $(n^3 - n)/6$ (bushy tree solution space) resp. $(n-1)^2$ (left deep tree solution space) feasible joins for linear graphs. For star graphs, there are $(n-1) \cdot 2^{n-2}$ (bushy tree solution space) feasible joins. However, this approach requires an especially tailored "join plan enumerator" for every class of join graphs that might be encountered, and for arbitrary join graphs still the entire solution space must be considered in order to guarantee that the optimal solution cannot be missed.

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

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

The algorithms discussed in this section either employ heuristics or a (pruned) search of the solution space in order to optimize the given join expression. We shall take a closer look at four different algorithms of this class with varying complexity and performance.

4.1.1 Dynamic Programming

This is the classical algorithm that has been used for join order optimization in System-R [SAC$^+$79]. It searches the solution space of left-deep processing trees. First, the set of partial solutions is initialized with all possible scan nodes for all relation attributes that participate in the query. For instance, if there is an index on attribute $R.A$, then both the index scan and the ordinary file scan are considered feasible partial processing trees. In the next step, every element with a cheaper, equivalent alternative is pruned from the set of possible partial solutions, where an alternative is considered "equivalent" if it joins the same set of relations and the sort order of the partial result is the same. In the following loop, the algorithm constructs in the kth iteration a set of k-relation partial solutions from a set of $(k-1)$-relation partial solutions. When this loop terminates, the set *partialsolutions* consists of at least one, possibly several equivalent, optimal solutions.

A pseudo code rendering of this algorithm is shown in Figure 2. Apart from the removal of all equivalent alternatives but the cheapest one, the original algorithm according to the cited reference performs further pruning of the search tree: it defers the introduction of cartesian products into partial solutions as long as possible, thus removing unlikely candidates for the optimal solution. However, although this strategy reduces the computational complexity, the result is no longer guaranteed to be optimal.

A major disadvantage of this algorithm is the high memory consumption for storing partial solutions. That (and the exponential running time) makes its application for queries that involve more than about ten to fifteen relations prohibitively expensive.

In a very recent work, Vance and Maier [VM96] devised a very efficient, so-called light-weight implementation of dynamic programming for bushy-tree join optimization. Their method allows to optimize join queries with up to about 18 relations—albeit with a rather simplified cost model.

```
function DynProg
inputs Rels "Set of relations to be joined"
outputs pt "Processing Tree"
  partialsolutions := {All scans for all attributes involved}
  "Remove all elements from partialsolutions with equivalent, lower-cost alternative"
  for $i := 2$ to |Rels|
    for all pt in partialsolutions
      for all $R$ in Rels such that $R$ not in pt
        pt := $\bowtie(\boxed{\text{pt}}, \textcircled{R})$
      end
    end
    "Remove all elements from partialsolutions with equivalent, lower-cost alternative"
  end
return "Arbitrary element from partialsolutions"
```

Figure 2: Algorithm "Dynamic Programming"

4.1.2 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. In this regard, this resembles Ingres' decomposition strategy [WY76]; however, unlike the decomposion strategy, which considers only the operands' cardinalities, the minimum selectivity heuristic makes use of the *selectivity factor* σ of the join $R_1 \bowtie R_2$ to achieve small intermediate results. 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

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

```
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 := $\textcircled{R_i}$
    else
      $R_i$ := "Relation from rels with smallest selectivity factor for
              the join with pt"
                  $\bowtie$
                 /     \
      pt :=     /       \
          $\boxed{pt}$      $\textcircled{R_i}$
    end
    rels := rels \ [$R_i$]
  while rels $\neq$ [ ]
return pt;
```

Figure 3: Algorithm "Minimum Selectivity"

is joined with the (so far) intermediate result and moved from $\mathcal{R}_{remaining}$ to $\mathcal{R}_{used}$. Figure 3 shows the complete algorithm for left-deep processing trees.

4.1.3 Krishnamurthy-Boral-Zaniolo Algorithm

On the foundation of [Law78] and [MS79], Ibaraki and Kameda showed in [IK84] 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. Based on 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 role 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, in principle, possible for nested loop join algorithms, but not for merge join or hash join algorithms. 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 sequences of J's nodes (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 $\text{rank}(S)$, 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 4, 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 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.

```
function Linearize

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

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

return chain;
```

Figure 4: Auxiliary Function "linearize"

```
function KBZ

inputs joingraph
outputs minorder "join order"

  tree := "Minimum spanning tree of joingraph"
  mincost := $\infty$

  forall node in tree
    lin := Linearize(node)
    "Undo normalization"

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

return minorder;
```

Figure 5: KBZ-Algorithm

In the main function *KBZ* (Figure 5), 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.

Due to its working principle, the KBZ algorithm requires the assignment of join algorithms to join graph edges *before* the optimization is carried out. This requirement and the restrictions concerning the cost model are the main drawbacks of the KBZ algorithm. The more sophisticated and detailed the cost model is, the more likely it is that KBZ's optimal result based on a (almost inevitably crude) approximation is different from the real optimum. Furthermore, separating the two tasks of join order optimization and join method assignment invalidates the main advantage of the KBZ algorithm, namely to yield the optimal solution in $O(n^2)$ time. In the following section, an algorithm is discussed that tries to remedy this situation.

4.1.4 AB Algorithm

The AB algorithm has been developed by Swami and Iyer [SI93]. It is based on the KBZ algorithm with various enhancements, trying to remove the restrictions that are imposed on the join method placement. The algorithm permits the use of two different join methods, namely nested loop and sort-merge. The sort-merge cost model has been simplified by Swami and Iyer such that it conforms 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 algorithm runs as follows (cf. Figure 6):

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.

2. The resulting tree query is optimized by the KBZ algorithm (*apply_KBZ*).

3. *change_order* attempts to further reduce the cost by swapping relations such that "interesting orders" can be exploited.

```
function AB

inputs joingraph
outputs minorder "join order"

  while number of iterations $\leq N^2$ do
  begin
    randomize_methods;
    while number of iterations $\leq N^2$ do
    begin
      apply_KBZ;
      change_order;
      change_methods;
    end;
  end;
  post_process;

return minorder;
```

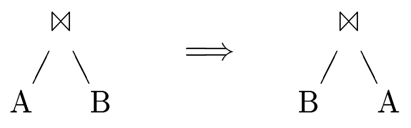
Figure 6: AB-Algorithm

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.

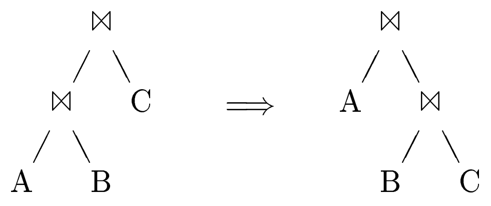
The AB algorithm comprises elements of heuristic and randomized optimizers. The inner loop searches heuristically for a local minimum, whereas in the outer loop several random starting points are generated in the manner of the Iterative Improvement algorithm (cf. Section 4.2.1). However, without ignoring the contribution of the KBZ algorithm, even with the AB extension it is hardly possible to make use of a sophisticated cost model.

4.2 Randomized Algorithms

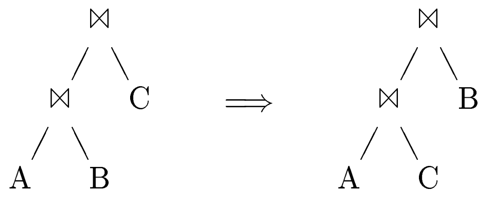
Randomized algorithms view solutions as *points* in a solution space and connect these points by edges that are defined by a set of *moves*. The algorithms discussed



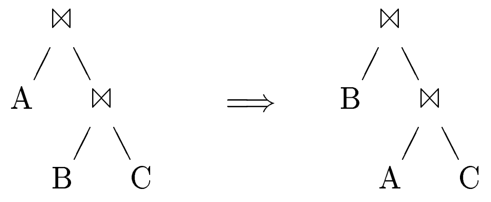
Commutativity
$A \bowtie B \Rightarrow B \bowtie A$



Associativity
$(A \bowtie B) \bowtie C \Rightarrow A \bowtie (B \bowtie C)$



Left Join Exchange
$(A \bowtie B) \bowtie C \Rightarrow (A \bowtie C) \bowtie B$



Right Join Exchange
$A \bowtie (B \bowtie C) \Rightarrow B \bowtie (A \bowtie C)$

Figure 7: Moves for Bushy-Tree Solution Space Traversal

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 [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 7 (introduced in [IK90]) are used for traversal of the solution space.

4.2.1 Iterative Improvement

If the solution space of the join optimization problem did contain but one global cost minimum without any local minima, we could use a simple hill-climbing algorithm for finding this minimum. However, because the solution space does contain local minima, hill-climbing would almost certainly yield one of them. The *Iterative Improvement Algorithm* [SG88, Swa89, IK90] tries to overcome this problem in the following way (Figure 8): After selecting a random starting point, the algorithm seeks a minimum cost point using a strategy similar to hill-

```
function IterativeImprovement
outputs minstate "Optimized processing tree"
  mincost := $\infty$
  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 8: Iterative Improvement

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** SimulatedAnnealing

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

  minstate := state; cost := Cost(state); mincost := cost
  temp := "Starting temperature"
  do
    do
      newstate := "state after random move"
      newcost := Cost(newstate)
      if newcost $\leq$ cost then
        state := newstate
        cost := newcost
      else "With probability $e^{\frac{newcost-cost}{temp}}$"
        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 9: Simulated Annealing

4.2.2 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 contains 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 9) is a variant on 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

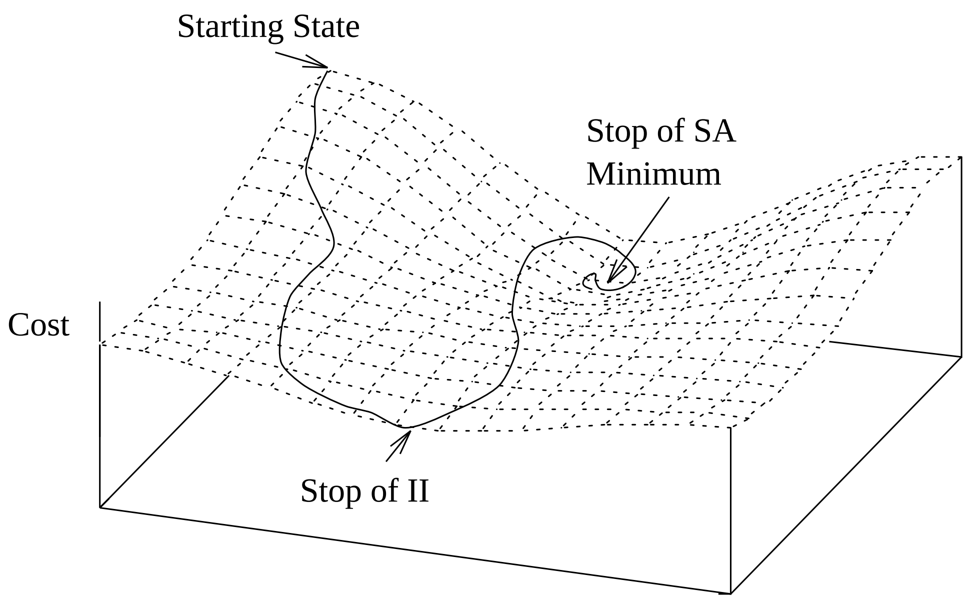


Figure 10: Iterative Improvement vs. Simulated Annealing

the annealing process of crystals. In this natural process, the system eventually reaches a state of minimum energy. The slower the temperature reduction is carried out, the lower the energy of the final state (one large crystal is of lower energy than several smaller ones combined). Figure 10 illustrates this behaviour: one iteration of 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, but also accepts moves that increase costs 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.

4.2.3 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 and descends rapidly into a local minimum, 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 8), and
2. from the lowest of these local minima, the Simulated Annealing algorithm (Figure 9) 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.

4.2.4 Toured Simulated Annealing

An approach similar to Two-Phase Optimization has been proposed in [LVZ93] in the context of a distributed computing environment. In *Toured Simulated Annealing*, several Simulated Annealing "tours" with different starting points are performed. Each starting point is derived from a deterministic algorithm that greedily builds processing trees using some augmentation heuristic. For instance, the Minimum Selectivity heuristic discussed in Section 4.1.2 could be used to provide these starting points.

Similarly to Two-Phase Optimization, the main benefit of Toured Simulated Annealing is the reduced running time. The starting temperature for the different tours is set much lower (0.1 times the initial plan's cost) than for Simulated Annealing with a random starting point, so the annealing process does not spend much time accepting moves that do not improve the current solution.

4.2.5 Random Sampling

In [GLPK94], a radically different idea is pursued. All randomized algorithms discussed so far are based on transformations that attempt to reduce a given solution's evaluation cost according to a set of rules until no further improvement can be achieved. However, an analysis of the cost distribution in the solution space reveals that a significant fraction of solutions is rather close to the optimum. An algorithm that draws a truly random sample of solutions should therefore contain the same fraction of good solutions as the entire space; however, designing such an algorithm that selects each processing tree with equal probability is not trivial. In the above mentioned work, such an algorithm (designed for acyclic join graphs) is presented; its application is most appropriate, when a reasonably good (evaluation cost of less than two times the minimum cost) evaluation plan has to be identified quickly, as the experimental results in [GLPK94] indicate.

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 propagate 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 evolution, 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).*

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 no improvement has been observed for a certain number of generations. 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

Because genetic algorithms were not nearly studied as intensively for join order optimization as other randomized algorithms, we shall discuss the questions associated with the employment of genetic algorithms for optimizing join expressions in more detail. In particular, we will not merely provide the techniques that we finally implemented, but some of the alternatives we considered (and tested) as well. Even if the basic algorithm remains unmodified, many variations for solution encoding, selection, crossover and mutation may be contemplated.

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. For encoding processing trees, we considered two different schemes:

1. Ordered list

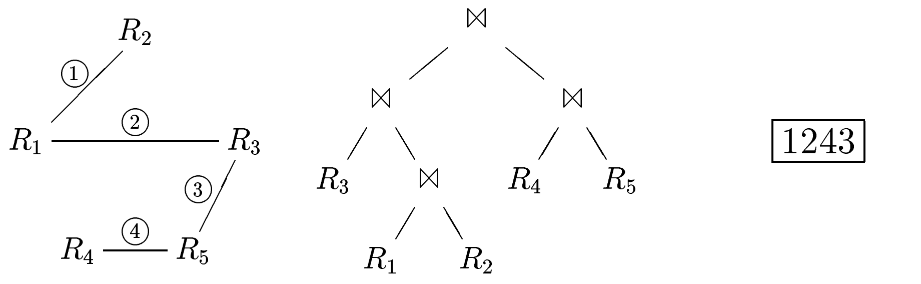
   (a) Left-deep trees:
   Solutions are represented as an ordered list of leaves. For instance, the processing tree $((((R_1 \bowtie R_4) \bowtie R_3) \bowtie R_2) \bowtie R_5)$ is encoded as "14325".

   (b) Bushy trees:
   Bushy trees *without* cartesian products are encoded as an ordered list of join graph edges. This scheme has been proposed in [BFI91]. As an example of this encoding scheme, we represent the processing tree depicted in Figure 11b as a character string. In a preliminary step, every edge of the join graph is labelled by an arbitrary number, such as in Figure 11a. 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 11c).

2. Ordinal number encoding

   (a) Left-deep trees:
   A chromosome consists of a sequence of ordinal numbers of the processing tree's list of leaves. For instance, the processing tree $((((R_1 \bowtie R_4) \bowtie R_3) \bowtie R_2) \bowtie R_5)$ is encoded as follows:

   - An ordered list L of all participating relations is made (for instance, based on their indices), such as $L = [R_1, R_2, R_3, R_4, R_5]$.



(a) Join Graph (b) Processing Tree (c) Encoded Tree

Figure 11: Encoding of Bushy Processing Trees

- 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 gene of the chromosome. 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 gene of the chromosome. 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 for the processing tree is "13211".

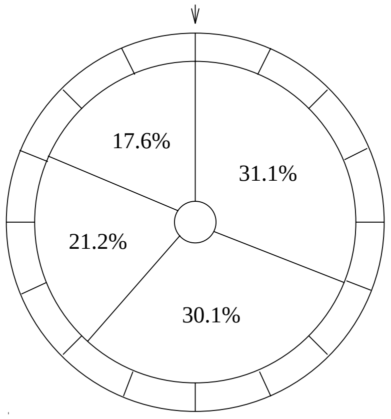
(b) Bushy trees:
For bushy trees, the ordinal numbers in the chromosome denote join nodes similar to the ordered list of join edges described above. But instead of specifying the join node by the corresponding join graph edge, the join's operands are used for that purpose. For instance, the processing tree in Figure 11b is encoded as follows:

- An ordered list of all participating relations is made exactly as for left-deep tree encoding: $L := [R_1, R_2, R_3, R_4, R_5]$.
- The first join node in the processing tree is $R_1 \bowtie R_2$, which involves R_1 and R_2 with index "1" and "2", respectively, so "12" becomes the first gene of the chromosome. R_1 and R_2 are replaced by R_{12}, so $L := [R_{12}, R_3, R_4, R_5]$.
- The next node in the processing tree joins relation R_3 with the result $R_1 \bowtie R_2$ (index 2 and 1), yielding gene "21" and $L := [R_{123}, R_4, R_5]$.
- Repeating this process finally leads to the complete chromosome "12 21 23 12".

In the actual implementation, the chromosome's genes carry additional information, namely operand order (encoding (1b)) and join algorithm (all encoding

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 12: Selection

schemes).

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 12. 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 12. 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 the premature convergence to a mediocre solution, because of the 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, we use *ranking based selection*. That means that not the value of the objective function itself but only its rank is used for biasing the selection wheel. In Figure 12, for instance, not the cost values would determine the fraction of the wheel a solution is assigned to, but

$$\begin{tikzcd} 3\boxed{154}2 \arrow[dr] & 3\boxed{451}2 \\ 4\boxed{532}1 \arrow[ur] & 4\boxed{352}1 \\ \text{“Parents”} & \text{“Offspring”} \end{tikzcd}$$

Figure 13: Crossover 1 – Subsequence Exchange for Ordered List Encoding

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 are not violated. Such a characteristic is the uniqueness of each character in the string for the Ordered List 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 considered two different crossover operators, namely *Subsequence Exchange* and *Subset Exchange*. They work as follows:

1. Subsequence Exchange (Ordered List Encoding)
   An example of the application of this operator is shown in Figure 13. It assumes the “ordered list” encoding scheme. In each of the two offspring chromosomes, a random subsequence is permuted according to the genes' order of appearance in the other parent. For instance, in Figure 13, the subsequence “532” is selected from the string “45321”. The first gene of its offspring remains the same as in the parent (4). The second gene is taken from the first gene of the other parent (3). The second gene of the other parent (1) cannot be used, because it is already present, so the third gene of the offspring is taken from the third gene of the other parent. Continuing this process yields at last the offspring chromosome “43521”. Determining the second offspring is carried out similarly.

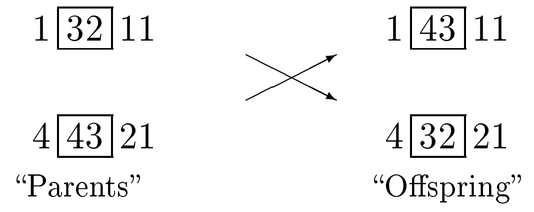


Figure 14: Crossover 2 – Subsequence Exchange for Ordinal Number Encoding

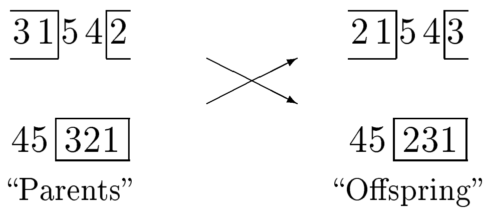


Figure 15: Crossover 3 – Subset Exchange

2. Subsequence Exchange (Ordinal Number Encoding)
   This operator is a slight variation of the above. It is intended for use in conjunction with the Ordinal Number 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 Ordinal Number Encoding, because we do not have to worry about duplicated characters. Figure 14 shows a sample application of this operator.

3. Subset Exchange (Ordered List Encoding)
   The basic idea for this operator is to avoid any potential problems with duplicated characters by simply selecting two random subsequences with equal 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 15 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 chromosome. If the operator must not introduce duplicate characters, as in ordered list chromosomes, two random genes are simply swapped in order to carry out the mutation; with Ordinal Number Encoding, a random gene of the chromosome 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.

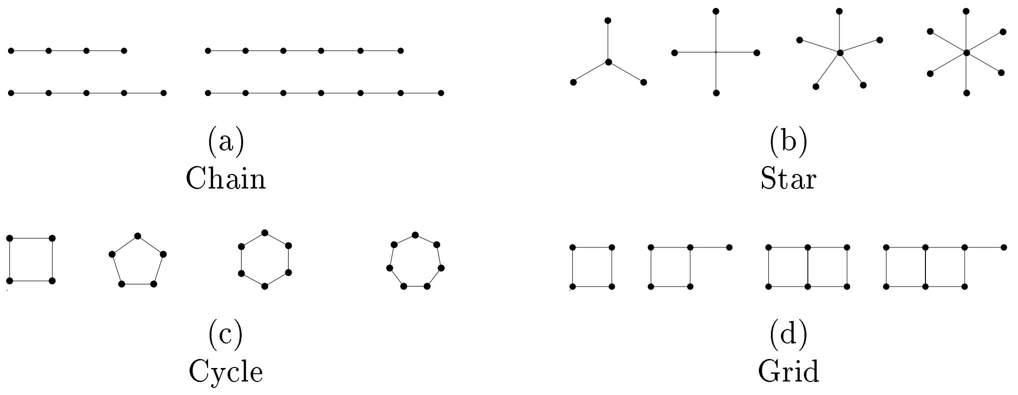


Figure 16: Join Graphs

Usually, only a 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.

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

The shape of the *join graph* can be chosen from the following four classes: chain, star, cycle and grid (cf. Figure 16a–d, respectively).

Relation cardinalities and *domain sizes* fall into four categories: S, M, L, XL as specified in Table 2; for instance, 35% of all relations comprise between 1000 and 10,000 tuples. These figures were chosen such that join results and cost values are neither too small (because of the error that would be introduced due to the page granularity of the cost model) nor too large (loss of accuracy due to limited floating point arithmetic resolution).

The query itself is specified such that all relations from a particular join graph are to be joined; the selectivities that are associated with the graph’s edges are computed according to the estimate used in System-R [SAC+79], i.e., $\sigma = 1/\min(\text{dom}(\textit{attribute}_1), \text{dom}(\textit{attribute}_2))$. Index structures (either hash tables or B^+-trees) facilitate read access on twenty percent of all relation attributes. While constructing a join graph, relation cardinalities and attribute domain sizes are

Class	Relation Cardinality	Percentage
S	10–100	15%
M	100–1000	30%
L	1000–10000	35%
XL	10000–100000	20%
		100%

(a)
Relation Cardinalities

Class	Domain Size	Percentage
S	2–10	5%
M	10–100	50%
L	100–500	30%
XL	500–1000	15%
		100%

(b)
Domain Sizes

Table 2: Relation Cardinalities and Domain Sizes

drawn independently; however, various "sanity checks" ensure that, for instance, a relation's cardinality cannot exceed the product of its attribute domain sizes.

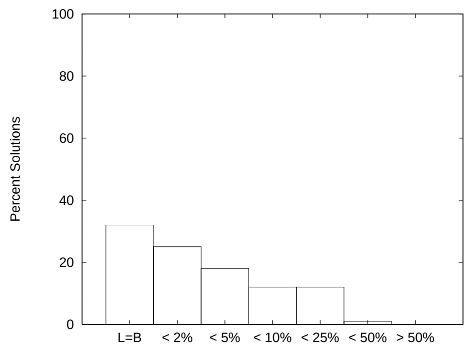
Each point in the following diagrams represents the average of at least thirty optimized queries, which proved to be a good compromise between the conflicting goals "avoidance of spurious results" (due to atypical behaviour of single runs) and "running time," as preliminary tests showed.

From the optimization strategies discussed in Section 4, we implemented the following algorithms: The System-R Algorithm, The Minimum Selectivity Heuristic and the KBZ Algorithm from the class of deterministic optimizers, and Simulated Annealing, Iterative Improvement and Genetic (all three in several variants) from the class of randomized/genetic optimizers. All deterministic algorithms yield solutions in the subspace of left-deep processing trees, whereas some of the randomized/genetic algorithms operate in the entire solution space (bushy trees).

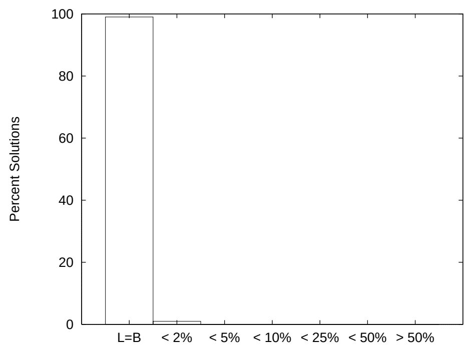
All cost figures are scaled with respect to the best solution available apart from System-R (because the System-R Algorithm could not be run for all parameter settings due to its high running time). For instance, a solution with a scaled cost of two is twice as expensive to evaluate as the best plan computed by any algorithm for that particular query. However, a curve for the algorithm that actually did compute the best solution is not necessarily shown in every plot. In other words, the set of algorithms that compete for the best solution is always the same, regardless of the subset that is depicted in a particular plot.

Solution Spaces Before presenting the benchmark results, we will take a closer look at the two solution spaces. The left-deep tree space is a subset of the bushy tree space, so we can expect lower running times of optimizers that operate in the left-deep space. On the other hand, there is the danger of missing good solutions that are not left-deep trees.

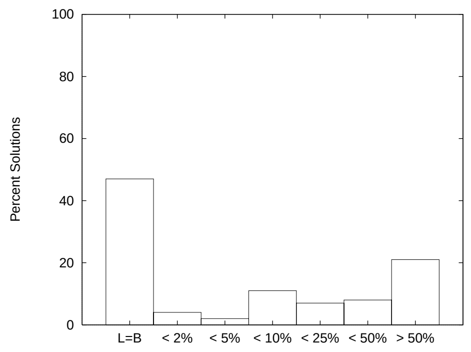
In order to get some insight into the advantages of using one solution space instead of the other, we determined both the "left-deep optimal" and "bushy optimal" solutions for one hundred randomly selected queries with six participating relations. The histograms for the four different join graph types in Figure 17a–d



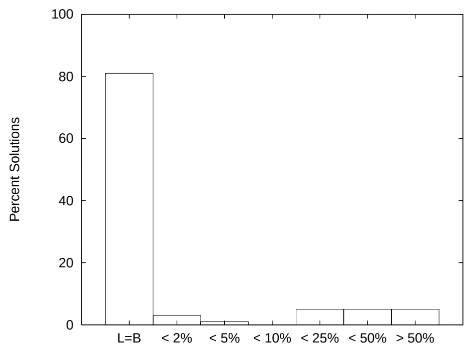
(a)
Chain



(b)
Star



(c)
Cycle



(d)
Grid

Figure 17: Left-Deep vs. Bushy Processing Tress

show the percentage of cases where the left-deep tree optimum and bushy tree optimum is of equal cost (i.e., the optimal solution is in fact a left-deep tree; labelled "L=B"). Following from left to right: the percentages of cases where the bushy tree optimum has less than two percent, between two and five percent, etc., lower cost than the left-deep tree optimum.

Considering those histograms, it becomes apparent that the shape of the join graph makes a big difference: for chain and cycle, we can find in half of all cases a better solution in the bushy tree solution space; for cycle, about one fifth even more than 50% cheaper than the best left-deep tree solution. Consequently, the investment in searching the bushy tree solution space should be profitable. On the other hand, for star join graphs in most of the cases the optima are left-deep trees anyway, because other tree shapes necessarily comprise cartesian products. Finally, for the grid join graph, the situation is not as clear as for the other three: 80% of the optima are left-deep trees, but a non-neglectable fraction of the bushy tree optima are far cheaper than their left-deep counterparts. A choice in favour of the bushy tree solution space would depend heavily on the

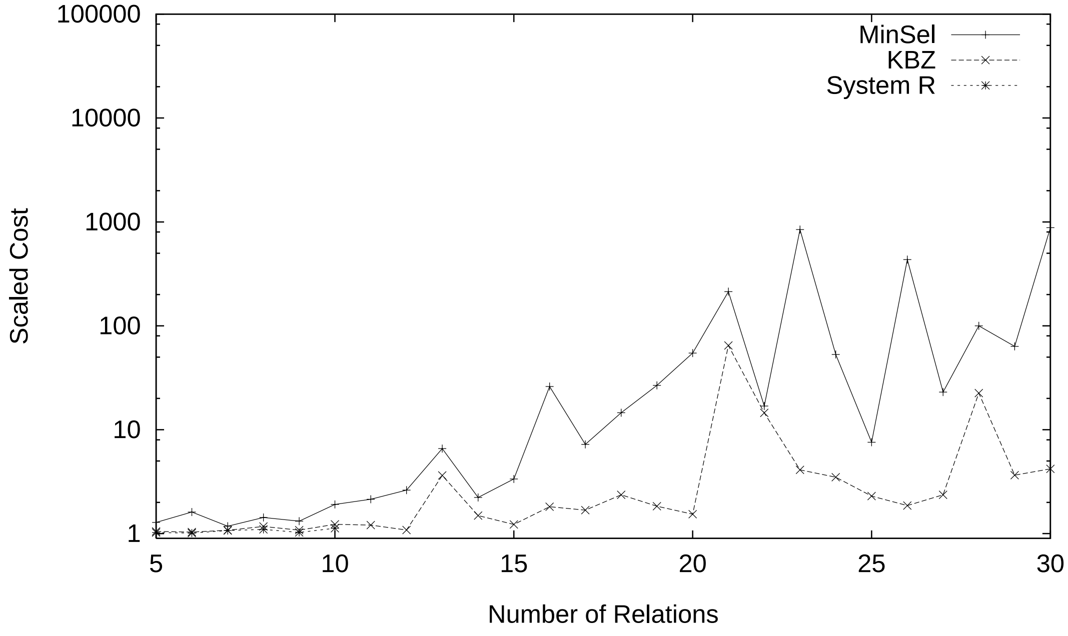


Figure 18: Deterministic Algorithms; Chain Join Graph

optimization algorithms' capability to locate these solutions (cf. [IK91]). In the remainder of this section, we will investigate whether the bushy tree optimizers can exploit the potential of good solutions in the bushy tree solution space.

Benchmark Results for Deterministic Algorithms In the first series of benchmarks, we shall examine deterministic algorithms. Figures 18 to 21 show the results for the *System-R Algorithm* (Section 4.1.1), the *Minimum Selectivity Heuristic* (Section 4.1.2), and the *KBZ algorithm* (Section 4.1.3). Because none of the cost formulae in Section 2.2 fulfils the KBZ algorithm's requirement, we used a simple approximation that counts the processed tuples for a nested-loop join (without considering index structures) in order to be able to run the algorithm. The cost of a complete evaluation plan thus derived, however, was computed according to the exact formulae.

On each of the diagrams, the scaled cost (cost of the optimized evaluation plan divided by the cost of the best plan derived by *any* of the optimization algorithms discussed in this section except System-R) is plotted against the number of relations participating in the query; the join graph type is noted in the respective caption. Please note the smaller scale in the y-axis for the star join graph. All deterministic algorithms yield left-deep processing trees; in addition, the best join method is determined locally for each join node, i.e., proceeding bottom-up and selecting the least costly join method for each node. The results for the System-R optimization are plotted for five to ten participating relations in order to provide some "absolute" basis for comparison purposes.

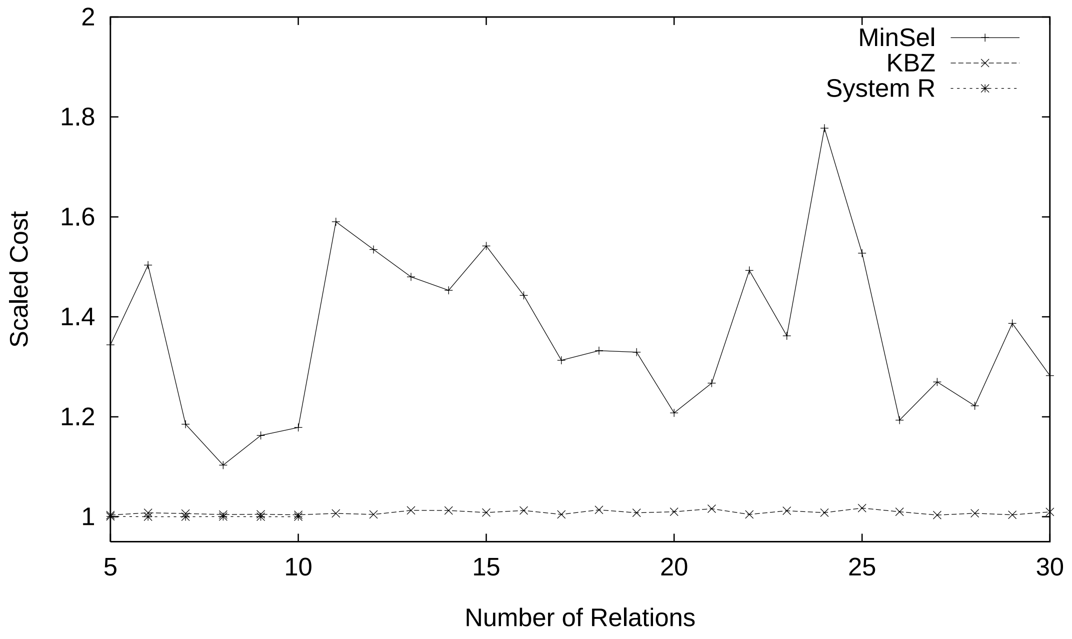


Figure 19: Deterministic Algorithms; Star Join Graph

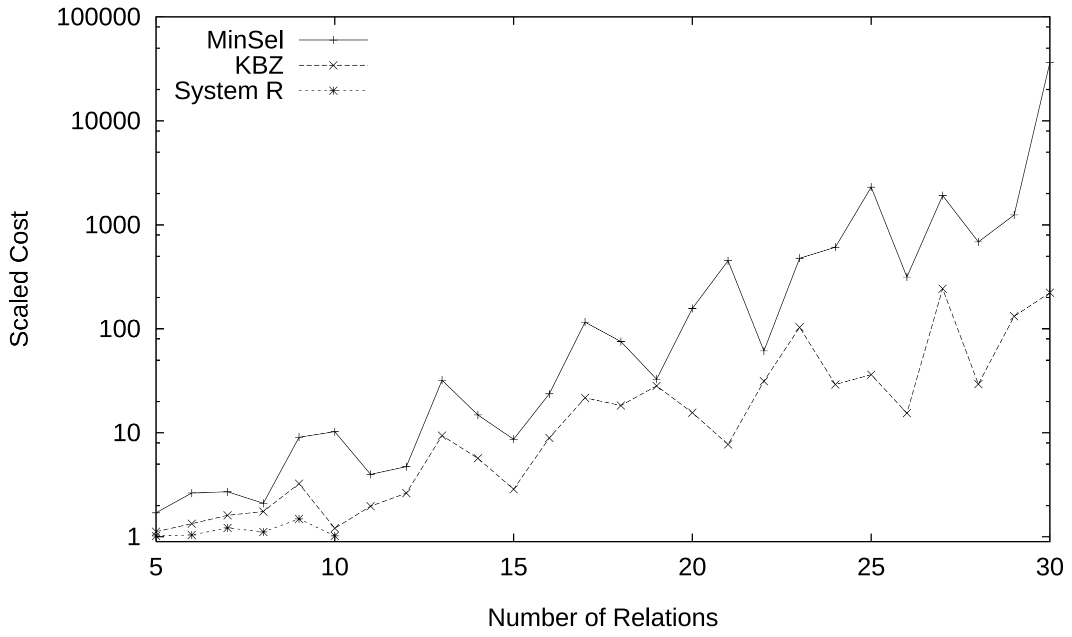


Figure 20: Deterministic Algorithms; Cycle Join Graph

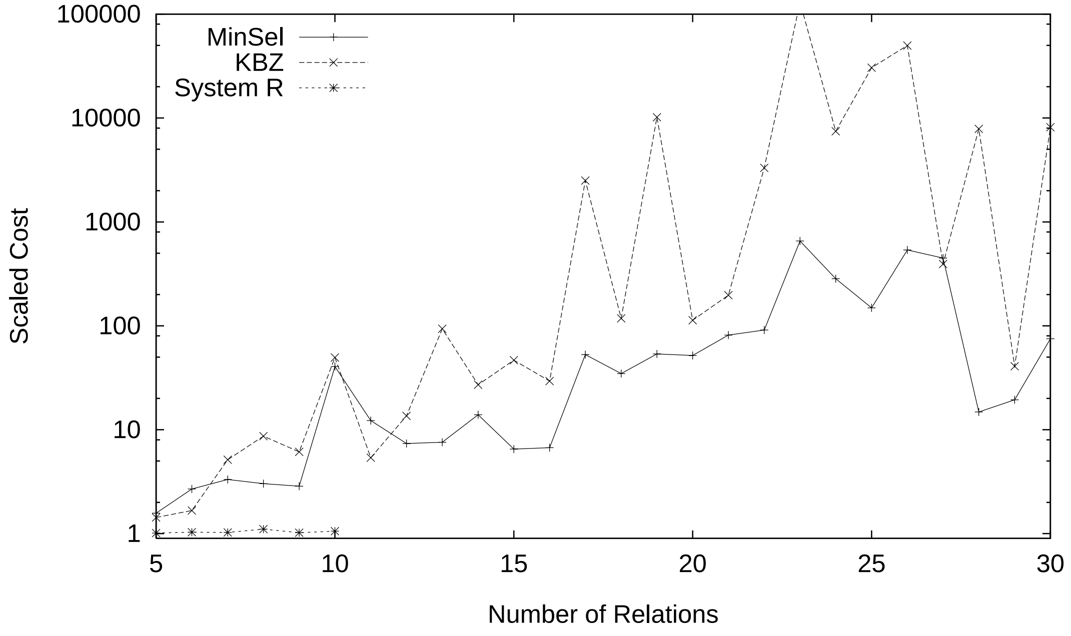


Figure 21: Deterministic Algorithms; Grid Join Graph

Despite the simple cost approximation for running the KBZ algorithm, this optimizer turns out to be the best of the two (heuristic) deterministic optimizers for the chain, star and cycle join graphs. Especially the solutions for the star join graph can hardly be improved by any of the other algorithms we tested—be it deterministic, randomized or genetic. For cycle and grid, the results are not quite as competitive, because these join graphs are cyclic and a spanning tree must be selected prior to the application of the KBZ algorithm. This effect becomes especially apparent for the grid join graph, where the Minimum Selectivity Heuristic performs best. The System-R Algorithm, which computes the optimal left-deep processing tree without cartesian products, achieves on the average a cost factor slightly above one, even though some solutions have cost factors below one, i.e., better than any of the approximate solutions. The reason why this cost factor is often higher than one is due to the limitation of the System-R Algorithm to left-deep processing trees without cartesian products.

To summarize the results for heuristics optimizers, we can note the following points: first, both of the discussed optimizers have a very short running time—the KBZ algorithm managed to compute the results for the thirty-relation star queries in less than two CPU-seconds each, the Minimum Selectivity Heuristic in less than a tenth of a second for each one of the same queries. Second, the performance in terms of quality is—except for star join graphs—only for small queries competitive, where KBZ performs best for the join graphs with low connectivity and Minimum Selectivity for the grid join graph. However, for small queries, one would probably not want to rely on heuristic algorithms, but compute the optimal solution using some kind of search strategy.

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 will compare three variants of Iterative Improvement (called IIJ, IIH [SG88], IIIO [IK90]), and Simulated Annealing (called SAJ, SAH [SG88], SAIO [IK90]) and two variants of genetic algorithms (Genetic, BushyGenetic). Furthermore, the results of the System-R optimization are shown as well for five to ten participating relations. The parameters for each algorithm are derived from the cited references (II, SA) or they were determined in preliminary tests (Genetic, BushyGenetic). In addition, for all algorithms generating left-deep trees, a search proceeding from the leaves of the tree to the root is performed for all trial solutions in order to determine the most appropriate join method on each join node.

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—please note the different scale in Figure 23. 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 joins in the query.
   - 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 outer loop iterations (i.e., temperature reductions) 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 inner loop is performed until the cost distribution of the generated solutions is sufficiently stable (for details cf. [SG88]).
- 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 equals the maximum change in cost in any accepted move at the current temperature.

3. SAIO

   - Moves are chosen from "Join Method Change," "Commutativity," "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 (randomly selected) starting state.
   - The number of iterations of the inner loop is sixteen times the number of joins in the query.
   - 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 encountered so far could not be improved in four subsequent outer loop iterations (i.e., temperature reductions) and the temperature has fallen below one.

4. Iterative Improvement (IIH, IIJ, IIIO)

   - All starting points are chosen randomly.
   - For an algorithm IIx, moves are chosen from the same set as the corresponding SAx algorithm.
   - Local minima are determined according to [SG88] (IIH, IIJ) and [IK90] (IIIO), i.e., a solution is considered a local minimum if k randomly selected neighbours fail to improve the result. k is the number of join graph edges for IIH and IIJ; for IIIO, k is the number of neighbouring states.
   - In order to perform a "fair" comparison between Iterative Improvement and Simulated Annealing, the total number of solutions considered is approximately the same for both the corresponding IIx and SAx algorithms.

5. Two-Phase Optimization (IIIO+SAIO)

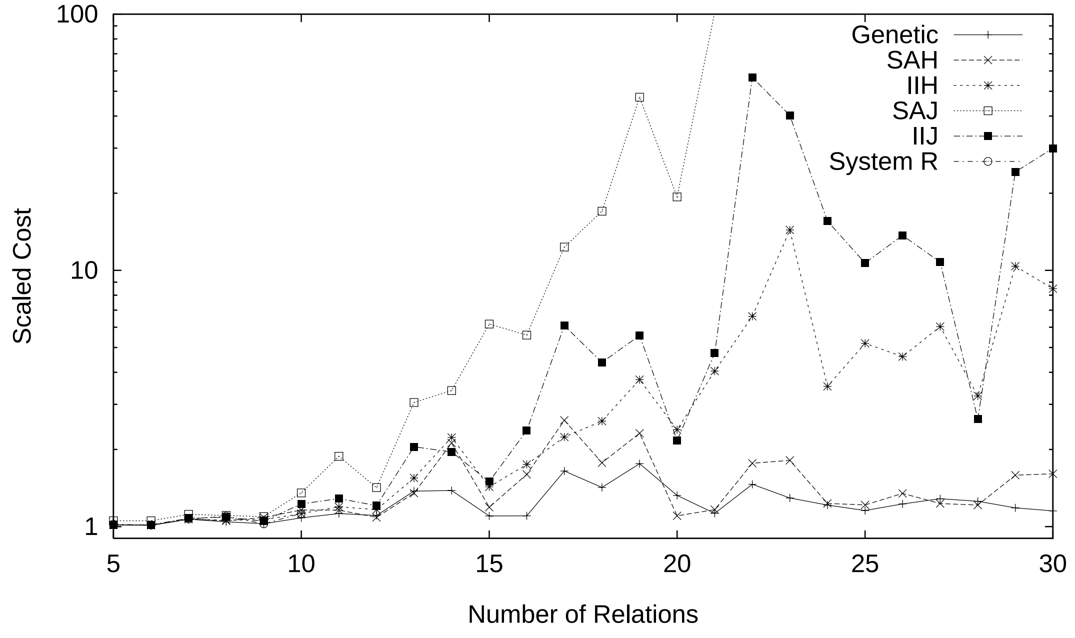


Figure 22: Randomized Algorithms, Left-Deep Tree Solution Space;
Chain Join Graph

- Ten random starting points 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 (Genetic/BushyGenetic)

   - Solution space: Left-deep processing trees/
     Bushy processing trees
   - Encoding: Ordered list of leaves/
     Ordinal Number Encoding
   - Ranking-based selection operator
   - Sequence exchange crossover operator
   - Population: 128
   - Crossover rate 65% (65% of all members of the population participate in crossover)
   - Mutation rate 5% (5% of all solutions are subject to random mutation)
   - Termination condition: 30 generations without improvement/
     50 generations without improvement

In Figures 22 to 25, the results for the left-deep tree optimizers are depicted. Although the parameter setting for SAH/SAJ and IIH/IIJ is similar, we note

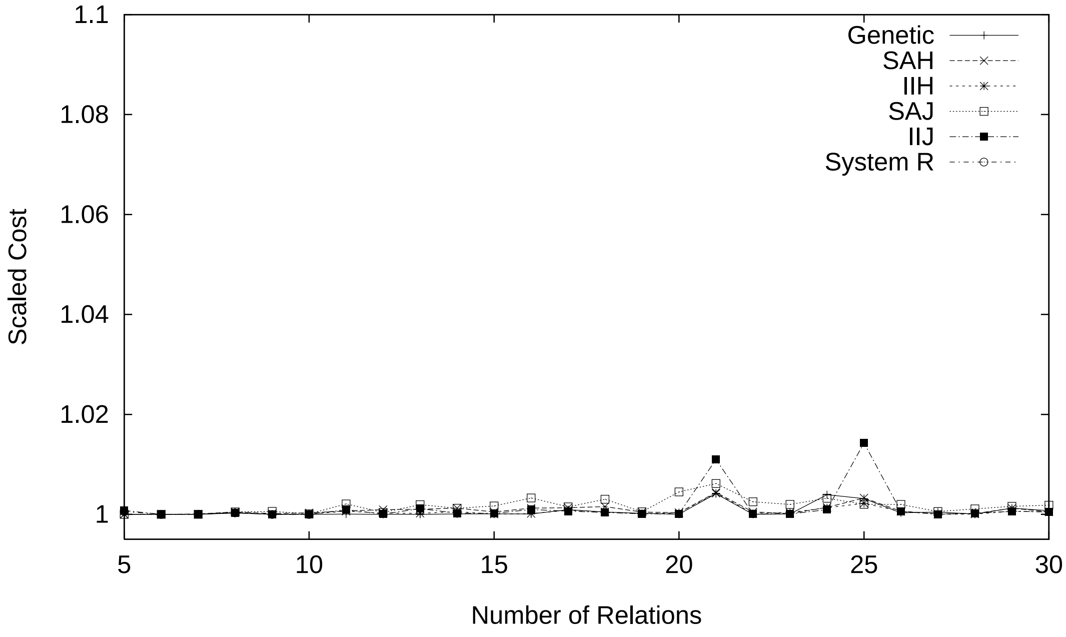


Figure 23: Randomized Algorithms, Left-Deep Tree Solution Space; Star Join Graph

that the “J” variants perform poorly for all but one of the join graph types. SAH and IIH perform much better, where, in turn, SAH is superior to IIH. In all cases, SAH and the genetic algorithm computed the best evaluation plans among the left-deep tree optimizers, with a slight superiority of the genetic algorithm. Apparently, the sophisticated equilibrium/freezing condition for SAH is the main reason for its good results. A closer look at the benchmark data revealed that indeed SAJ visited much less solution alternatives than SAH. The Iterative Improvement variants that were designed to consider about as many different solutions as the respective Simulated Annealing algorithms reflect this fact: IIH achieves better results than IIJ. Apart from the quality of the derived results, another important criterion for selecting an optimizers is its running time, which we will investigate later. In the meantime, we will look at the performance of those optimizers that operate in the bushy tree solution space.

These optimizers, namely SAIO, IIIO, 2PO and a genetic algorithm (Bushy-Genetic) are being compared in Figures 26 to 29. In addition, the best two left-deep optimizers' curves (SAH and Genetic) are included as well in order to facilitate direct comparison. It turns out that, in terms of quality, none of the implemented algorithms performed better than the Two-Phase Optimization algorithm (2PO)—regardless of the join graph type, although the gap between Simulated Annealing, Iterative Improvement and Two-Phase Optimization is quite narrow. In contrast to the left-deep case, where the genetic algorithm showed a slight superiority over the Simulated Annealing results, this is not the case in

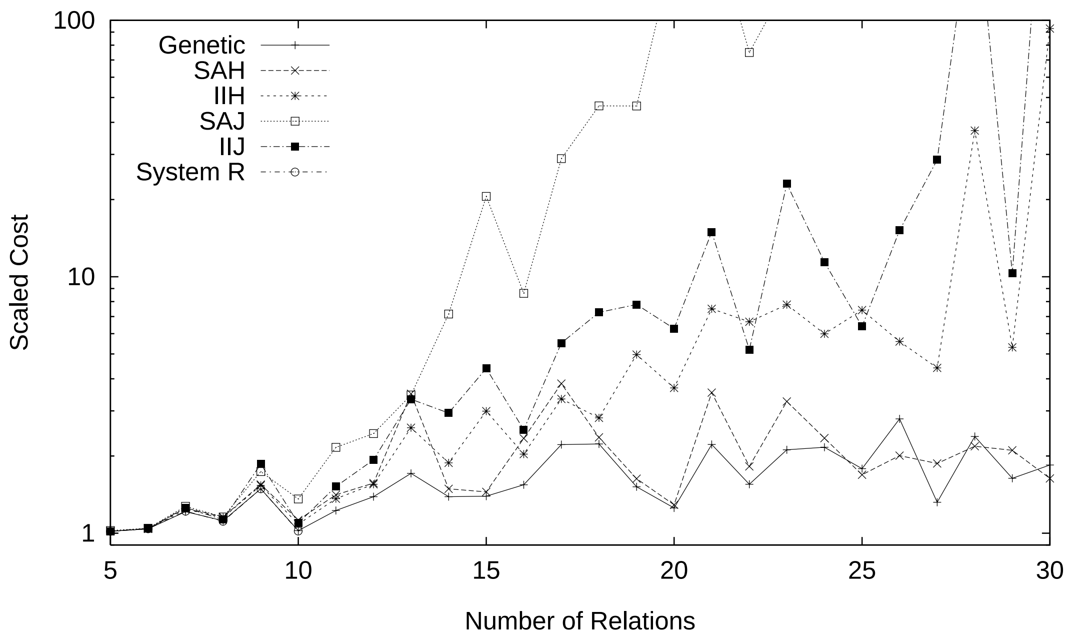


Figure 24: Randomized Algorithms, Left-Deep Tree Solution Space; Cycle Join Graph

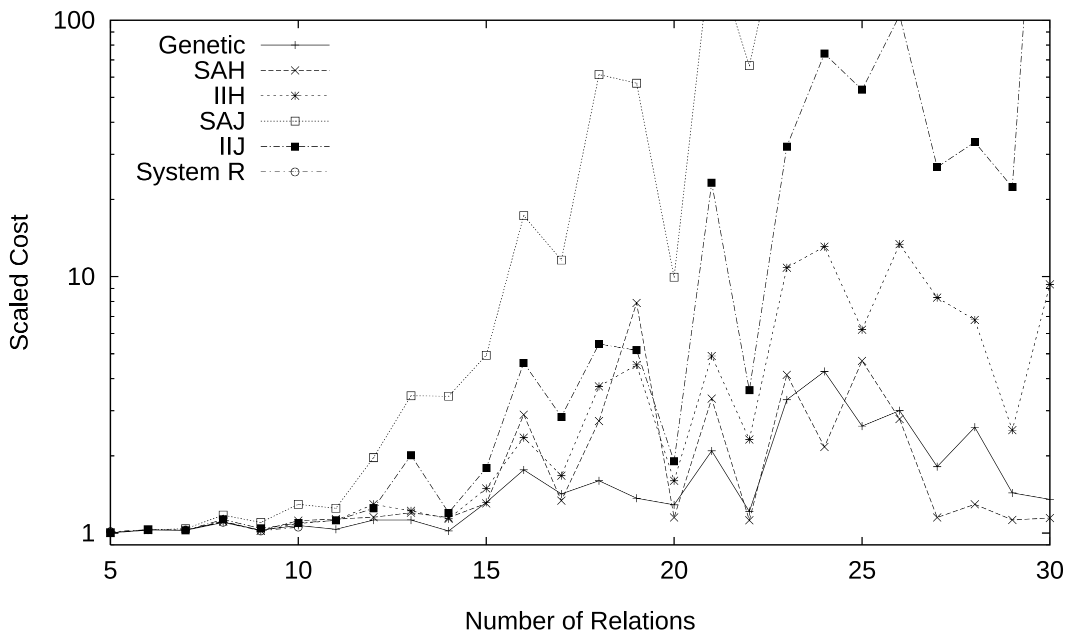


Figure 25: Randomized Algorithms, Left-Deep Tree Solution Space; Grid Join Graph

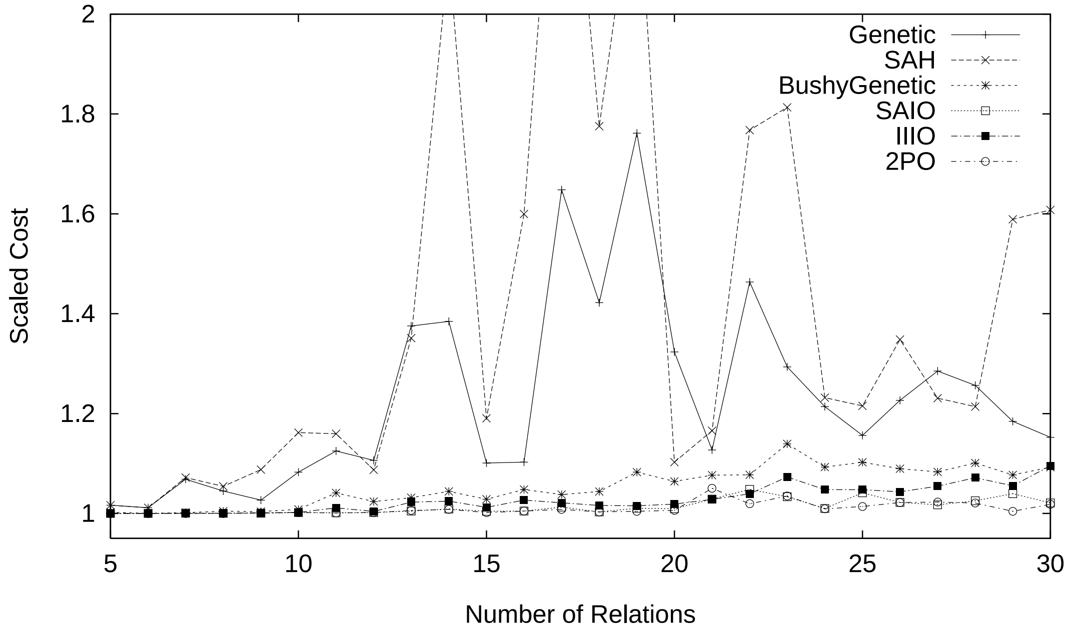


Figure 26: Randomized Algorithms, Bushy Tree Solution Space; Chain Join Graph

the bushy tree solution space. Although the genetic algorithm does not perform particularly poor, it cannot quite equal the quality of Simulated Annealing or Two-Phase Optimization.

Only for star queries, all algorithms exhibit a very similar behaviour (divergence just about one percent), so the algorithms' running time would be the decisive factor in this case. For all other join graphs, every bushy tree optimizer easily outperforms even the best implemented left-deep tree optimizer, which confirms that these algorithms are indeed capable of locating the superior solutions of the bushy tree solution space.

Let us now look at the running times for the different optimizers. Although the quality of the generated solutions is a very important characteristic, the running time of an algorithm has a considerable impact on the final choice. The intended application area determines how much time can be spent on the optimization: queries that are stated interactively and run only once do not warrant the same amount of optimization than compiled queries that are repeated hundreds or thousands of times. In Figure 30, average running times for Genetic, SAH, BushyGenetic, SAIO, IIIO, 2PO and System-R are plotted against the number of relations participating in the queries ("chain" join graph). The running times for the various algorithms were determined on a SPARCstation 20/612MP and denote CPU time.

From the six randomized/genetic algorithms, SAIO has the longest running times with up to 2800 CPU-seconds for thirty relation queries. Although the

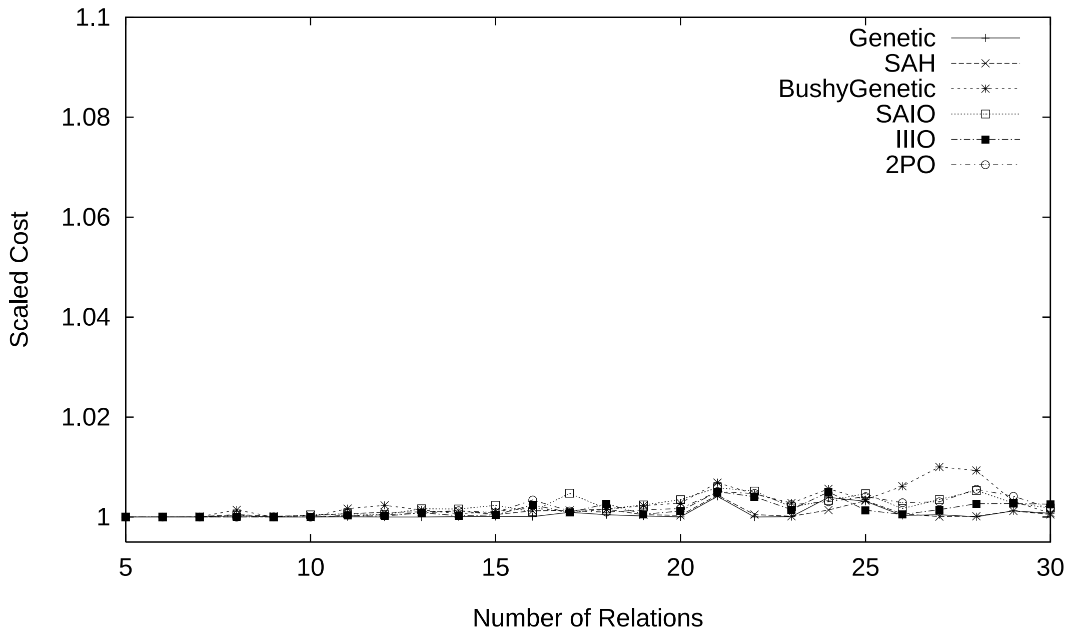


Figure 27: Randomized Algorithms, Bushy Tree Solution Space;
Star Join Graph

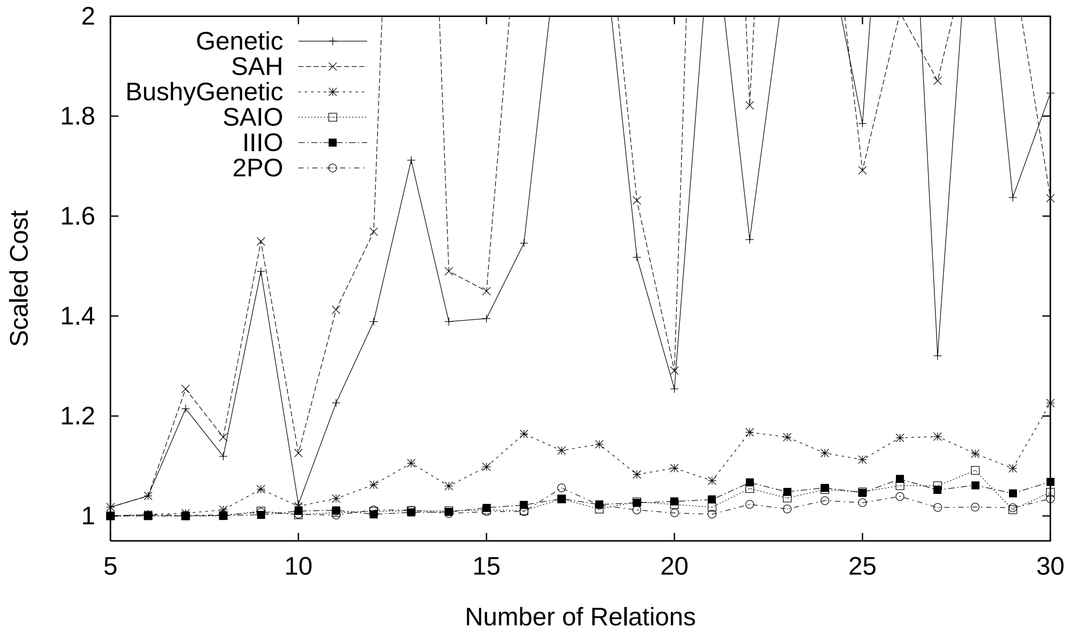


Figure 28: Randomized Algorithms, Bushy Tree Solution Space;
Cycle Join Graph

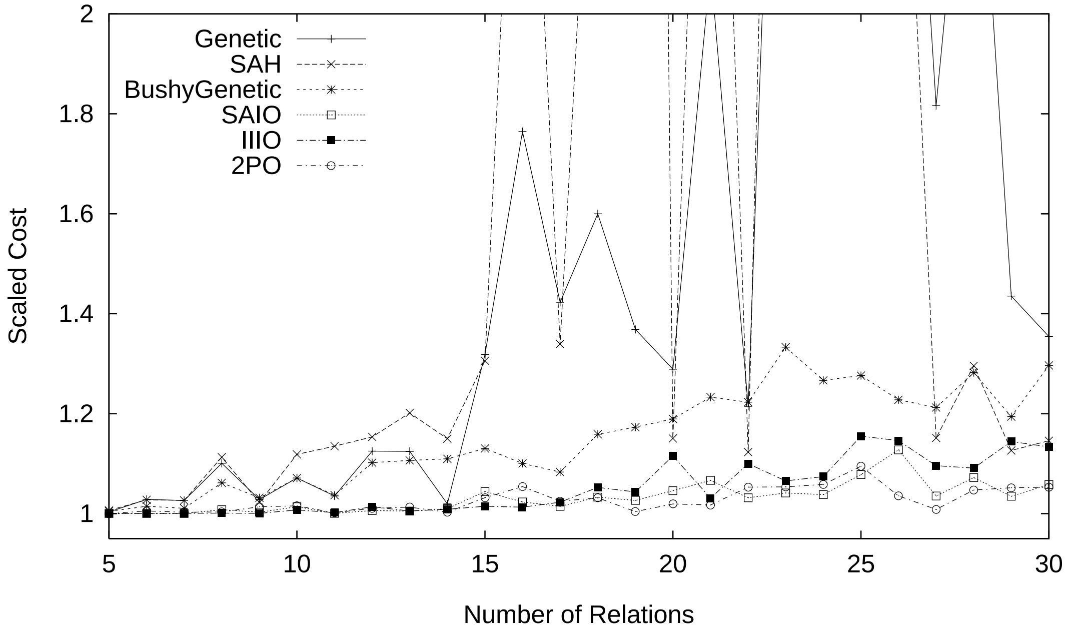


Figure 29: Randomized Algorithms, Bushy Tree Solution Space; Grid Join Graph

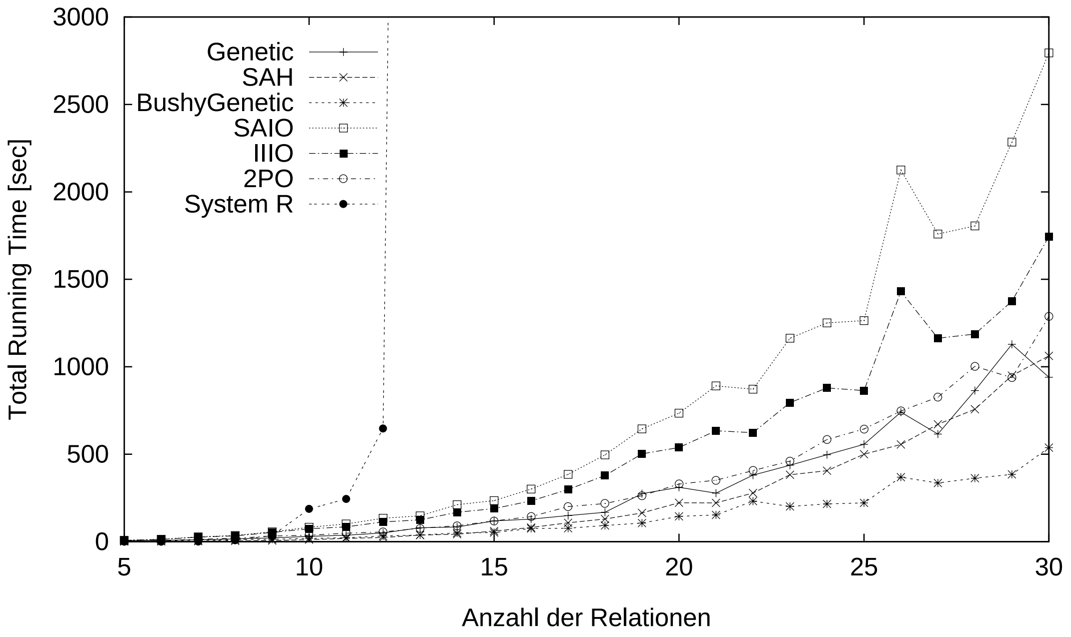


Figure 30: Total running times

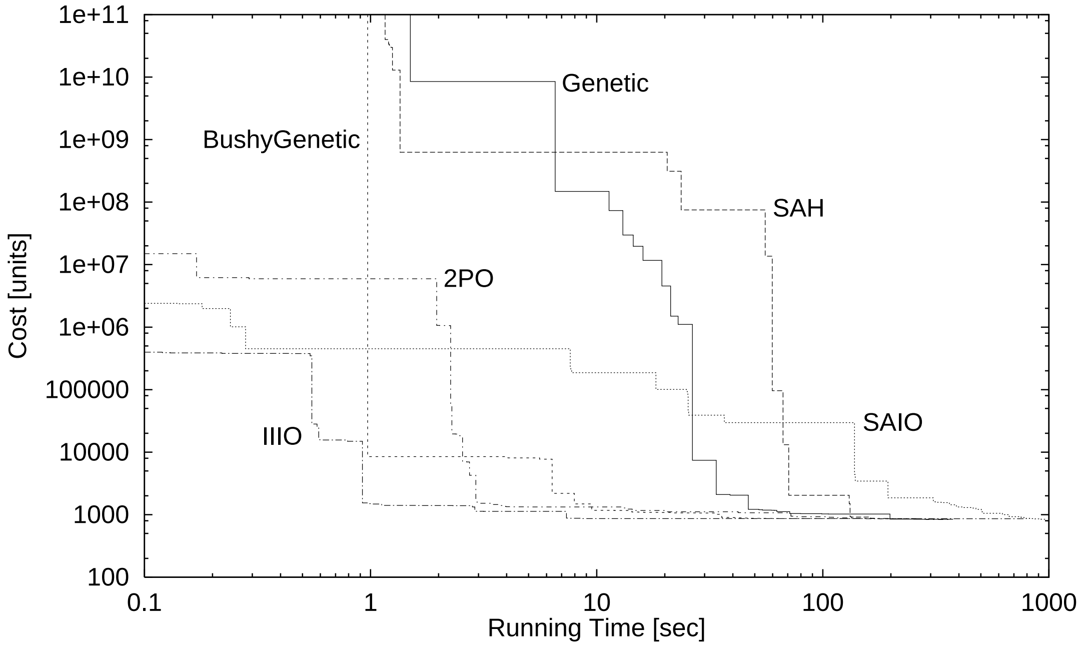


Figure 31: Approach to the final solution

Two-Phase Optimization algorithm (2PO) yields slightly better solutions, it requires a running time of about 1300 CPU seconds, only half of SAIO's time. As expected, left-deep tree optimizers (SAH, Genetic) run faster than bushy tree optimizers, but the gain of speed must be paid by loss of quality. Surprisingly, the BushyGenetic algorithm runs even faster than both the left-deep optimizers, even though it yields solutions that are at least as good as theirs: it can handle the thirty relation queries on the average in just about 500 seconds. In contrast to the randomized/genetic algorithms, the computation of the optimal left-deep tree without cartesian products is in our implementation only for queries involving up to twelve relations feasible; for instance, thirteen-relation queries already require on the average an optimization time of more than one hour.

In Figure 31, for the same six algorithms the approach to their respective final solutions is shown. Each time the currently best solution is improved, the gain in absolute cost units is noted (y-axis) together with the time of its occurrence (x-axis). Because only a single optimization run for a twenty relation query (chain join graph) is plotted, we cannot draw any far reaching conclusions, but nevertheless the curves reflect the algorithms' typical behaviour we observed quite well. Both Simulated Annealing algorithms, SAIO as well as SAH, spend a good deal of the total running time investigating high-cost processing trees; SAIO required more than 150 seconds to reach a cost level of less than ten times the cost of the final solution. SAH ran faster, but it still took a very long time for the approach to its final solution. On the other hand, both Two-Phase Optimization (2PO) as well as Iterative Improvement (IIIO) achieved a very good result within less than one resp. three seconds running time. For the genetic algorithms, especially BushyGenetic can reach acceptable solutions very quickly. Even the initial population consisted of at least one member with an evaluation cost that is as low as SAIO's after running more than a hundred times as long. Although the drawing of the initial population is not guaranteed to be unbiased in our implementation, we can note that genetic algorithms nicely supplement the approach in [GLPK94] (Section 4.2.5): in a first step, a random sample could be drawn using the algorithm presented in [GLPK94], which can be used in a second step as the initial population for the genetic algorithm.

Summary Comparing the performance of the various optimization algorithms, we can draw the following conclusions:

Algorithms that perform an exhaustive or near exhaustive enumeration of the solution space, such as dynamic programming, can compute the optimal result, but the high running time makes their application only feasible for queries that are not too complex (i.e., less than about ten to fifteen relations for left-deep processing trees). For the same reason, searching the bushy tree solution space can be carried out only for very simple queries, (in our experiments, about six to seven relations), so the advantages of this solution space can hardly be exploited.

Heuristic optimizers avoid the high time complexity of exhaustive enumeration, but the results are, especially for complex queries with many participating relations, rarely acceptable. The KBZ algorithm, although yielding the optimal left-deep solution under certain circumstances, is difficult to apply in practice: the need for cost model approximations and problems concerning join method assignment limits its usefulness. We found that only for star queries the KBZ algorithm is competitive; its short running time compared to alternative algorithms (especially randomized/genetic) makes it the solution of choice.

Finally, randomized and genetic algorithms operating in the bushy tree solution space are the most appropriate optimizers in the general case provided the problems are too complex to be tackled by exhaustive enumeration. Which one of the discussed algorithms is the most adequate depends on the particular application area, namely whether short running time or best optimization performance is the primary goal. If good solutions are of highest importance, Two-Phase Optimization, the algorithm that performed best in our experiments, is a very good choice; other Simulated Annealing variants, for instance Toured Simulated Annealing (TSA, [LVZ93]), that we did not implement, are likely to achieve quite similar results. The "pure" Simulated Annealing algorithm has a much higher running time without yielding significantly better solutions. If short running time is more important, Iterative Improvement (IIIO), the genetic algorithm (BushyGenetic), and, to a lesser extent, Two-Phase Optimization (2PO) are feasible alternatives. Especially the first two degrade gracefully if they are preempted: in the example run in Figure 31, they achieved acceptable results in less than a second. Moreover, as mentioned above, genetic algorithms can be combined very well with the transformationless approach in [GLPK94].

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 approximate 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 much better suited for join optimizations; although they require a longer running time, the results are far better.

For the question of the adequate solution space, we have found that, with the exception of the star join graph, the bushy tree solution space is preferable in spite of the fact that "pipelining" (avoiding to write intermediate results to secondary memory) can be carried out mainly by left-deep processing trees.

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 We should like to thank the referees for their valuable comments and suggestions on an earlier draft of this paper, and our colleagues for countless CPU-hours on their workstations. This work was partially supported by grant DFG Ke 401/6-1 and SFB 346.

References

[BFI91] K. Bennet, M. C. Ferris, and Y. E. Ioannidis. A genetic algorithm for database query optimization. In *Proc. of the Fourth Intl. Conf. on Genetic Algorithms*, pages 400–407, San Diego, USA, 1991.

[EN94] E. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, Redwood City, CA, USA, second edition, 1994.

[GLPK94] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Fast, randomized join-order selection—why use transformations? In *Proc. of the Conf.*

on Very Large Data Bases (VLDB), pages 85–95, Santiago, Chile, September 1994.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley, Reading, MA, USA, 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, April 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, May 1987.

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

[KM92] A. Kemper and G. Moerkotte. Access Support Relations: an indexing method for object bases. *Information Systems*, 17(2):117–146, 1992.

[KM94] A. Kemper and G. Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science.* Prentice Hall, Englewood Cliffs, NJ, USA, 1994.

[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.

[Law78] E. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Ann. Discrete Math.*, 2:75–90, 1978.

[LVZ93] R. Lanzelotte, P. Valduriez, and M. Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 493–504, Dublin, Ireland, 1993.

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

[MS79] C. Monma and J. Sidney. Sequencing with series-parallel precedence constraints. *Mathematics of Operations Research*, 4:215–224, 1979.

[OL90] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 314–325, Brisbane, Australia, 1990.

[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, May 1979.

[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, IL, USA, May 1988.

[Sha86] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. on Database Systems*, 11(9):239–264, September 1986.

[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, Austria, April 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, OR, USA, May 1989.

[VM96] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian product. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, June 1996.

[WY76] E. Wong and K. Youssefi. Decomposition—A strategy for query processing. *ACM Trans. on Database Systems*, 1(3):223–241, 1976.

[YW79] K. Youssefi and E. Wong. Query processing in a relational database management system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 409–417, Rio de Janeiro, Brasil, 1979.