A New Heuristic for Optimizing Large Queries

Leonidas Fegaras

Department of Computer Science and Engineering
The University of Texas at Arlington
416 Yates Street, P.O. Box 19015
Arlington, TX 76019-19015
email: *fegaras@cse.uta.edu*

Abstract

There is a number of OODB optimization techniques proposed recently, such as the translation of path expressions into joins and query unnesting, that may generate a large number of implicit joins even for simple queries. Unfortunately, most current commercial query optimizers are still based on the dynamic programming approach of System R, and cannot handle queries of more than ten tables. There is a number of recent proposals that advocate the use of combinatorial optimization techniques, such as iterative improvement and simulated annealing, to deal with the complexity of this problem. These techniques, though, fail to take advantage of the rich semantic information inherent in the query specification, such as the information available in query graphs, which gives a good handle to choose which relations to join each time.

This paper presents a polynomial-time algorithm that generates a good quality order of relational joins. It can also be used with minor modifications to sort OODB algebraic operators. This algorithm is a bottom-up greedy algorithm that always performs the most profitable joins first. Our heuristic is superior to related approaches in that it generates bushy trees, it takes into account both the sizes of intermediate results and the predicate selectivities, and it always updates the query graph to reflect the new sizes and the new selectivities.

1 Introduction

One of the most important tasks of a relational query optimizer is finding a good join order to evaluate an n-way join. Since a DBMS typically provides binary physical algorithms to evaluate joins, a relational optimizer must map an n-way join into a sequence of binary joins. To find the best join sequence, an optimizer needs to consider many different join orders, various physical algorithms, and all the available access paths. A naive generate-and-test approach would require exponential time ($\mathcal{O}(n! \times k^n)$ for an n-way join and for k different ways of evaluating a join), which becomes infeasible for large n. The bottom-up approaches adopted by commercial relational optimizers (mostly based on the dynamic programming approach of System R [17]) are better than the generate-and-test approach, but they are still exponential ($\mathcal{O}(2^n)$ in the worst case). As the result of that, most commercial systems cannot handle joins of more than ten tables.

Even though queries of more than ten tables do not appear very often in traditional database applications, there are some emerging applications, such as decision support systems and on-line analytical processing (OLAP) for analyzing data warehouses, that require the use of complex queries with aggregates. In addition, many recent DBMSs are in fact object-oriented (OODBs), object-relational, or deductive database systems, and they support many advanced modeling capabilities. Queries expressed in these systems, when evaluated, often require a large number of implicit joins, which the programmer may be unaware of. For example, most modern OODB languages, such as ODMG-93 OQL [1], support an SQL-like interface for associative access of data. These languages combine the benefits of object-oriented programming, where one can navigate through objects using path expressions and methods, and the benefits of relational databases, where one can retrieve data using a high-level declarative query language. The best way to evaluate a path expression in an OODB query is to use pointer joins between the type extents of the objects involved in the path expression [2, 11]. This approach requires that a cascade of n projections in a path expression be mapped into a join of n extents. An alternative is to use a naive pointer chasing, which resembles the query evaluation in network databases and does not leave many opportunities for optimization. In addition, OODB queries can be arbitrarily nested. If a nested query is evaluated as is, its execution would resemble a nested-loops join. Some forms of nesting can be avoided by using outer-joins combined with grouping [13, 6]. In most OODB algebras, where grouping is an explicitly supported operator and can appear in any place in an algebraic form, any form of query nesting can be unnested by promoting the inner queries one level up in the query nesting hierarchy [3, 4]. This results to a large flat join, which is the concatenation of all the joins from every level in the nested query. Furthermore, all unnesting methods proposed in the literature introduce additional operators, such as outer-joins and grouping, thus increasing the number of operators to be considered each time. Other sources of implicit operations include views and methods, which are often expanded during query optimization.

There is a number of heuristics already proposed in the literature for optimizing large relational queries, even though most of them deal with acyclic query graphs and generate deep-left join trees only [18, 10]. As it is widely believed now in the database community, the goal of an optimizer is not to find the best evaluation plan to execute a query, since this is infeasible for non-trivial queries, but to avoid the worst plans. Cost-guided searching based on combinatorial optimization techniques, such as iterative improvement and simulated annealing [9], can be used to extract a solution in a fixed amount of time. Unfortunately, none of the cost-based searching methods proposed so far actually takes advantage of the semantic information inherent in queries, such as the information available in the query graphs, which gives a good handle to choose which relations to join each time. We believe, that a heuristic algorithm, mostly based on the semantic knowledge inherent in the query graphs, and to a lesser extent on a cost-based searching, would be a more appropriate choice than cost-based searching alone.

This paper presents a polynomial-time heuristic algorithm, called *GOO*, that generates a "good quality" order of relational joins. It can also be used with minor modifications to sort OODB algebraic operators. GOO is a bottom-up greedy algorithm that always performs the most profitable joins first. The measure of profit is the size of the intermediate results, but GOO can be easily modified to use real cost functions. GOO manipulates query graphs and takes into account both the sizes of the joins constructed in earlier steps and the predicate selectivities. It generates bushy join trees that have a very low total size of intermediate results. The difference between our algorithm and other related bottom-up heuristics is that, after two nodes are selected from the query graph to be joined, our algorithm always updates the query graph to reflect the new sizes and the new predicate selectivities.

GOO has complexity $\mathcal{O}(n^3)$, where n is the number of relations, but it can derive a join order for 100 joins in 32 milliseconds. When compared to iterative improvement (II), GOO is far superior and considerably more stable and consistent. When II is allowed to run ten times longer than GOO, the plans derived by GOO are better than those derived by II by a factor between 10^{20} and 10^{30} for 50-100 joins (this is the scaled cost improvement for random query graphs using real cost functions). When II is allowed to run one hundred times longer than GOO, the factor is between 10^{10} and 10^{20} for 50-100 joins.

The paper is organized as follows. Section 2 presents our polynomial-time heuristic that finds a join order with a low total size of intermediate results. Section 3 extends our join ordering algorithm to handle OODB query graphs, which may include unnest and nest operators along with joins. Section 4 compares the performance of our join ordering algorithm with that of the iterative improvement algorithm. Finally, Section 5 compares our work with related work.

2 The Join Ordering Algorithm

2.1 Terminology

We are interested in optimizing simple SQL queries of the form:

$$\begin{array}{r l} \textbf{select} & \cdots \\ \textbf{from} & r_1,\, r_2, \ldots, r_n \\ \textbf{where} & p_1 \textbf{ and } p_2 \textbf{ and } \cdots \textbf{ and } p_m \end{array}$$

This query uses n (not necessarily different) relations $r_1, \ldots, r_n$ and m uncorrelated (independent) predicates $p_1, \ldots, p_m$. A predicate p_k is either:

1. a *local predicate* on r_i, if it references the relation r_i only, or
2. a *join predicate* between r_i and r_j, if it references r_i and r_j only.

Our join ordering algorithm considers join predicates only, since any local predicate p_k on r_i can be safely ignored if we replace r_i by $\sigma_{p_k}(r_i)$. Thus, we represent a query Q as a pair $(\mathcal{R}, \mathcal{P})$, where $\mathcal{R} = \{r_1, r_2, \ldots, r_m\}$ is the set of relations referenced in Q and $\mathcal{P} = \{p_1(r_{11}, r_{12}), p_2(r_{21}, r_{22}), \ldots, p_m(r_{m1}, r_{m2})\}$ is the set of the join predicates in Q where $r_{ij} \in \mathcal{R}$.

The *selectivity* of a join predicate $p_k(r_i, r_j) \in \mathcal{P}$ is defined as follows [17]:

$$\text{selectivity}(p_k(r_i, r_j)) \quad = \quad \frac{\|r_i \bowtie_{p_k(r_i,r_j)} r_j\|}{\|r_i\| \times \|r_j\|}$$

where $\|X\|$ is the cardinality of X. Selectivities range between 0 and 1 (the predicate *true* has selectivity 1 while the predicate false has selectivity 0). For uncorrelated predicates p_1 and p_2, a good approximation of the selectivity of $p_1 \wedge p_2$ is [17]:

$$\text{selectivity}(p_1 \wedge p_2) \;=\; \text{selectivity}(p_1) \times \text{selectivity}(p_2)$$

For example, if 50% of the tuples satisfy p_1 and 60% satisfy p_2, then 30% (which is the 50% of 60%) of the tuples will satisfy both p_1 and p_2. The predicate P_{ij} and its selectivity S_{ij} between two relations r_i and r_j in Q is defined as follows:

$$\begin{aligned} P_{ij} &= \bigwedge_{p_k(r_i,r_j)\in\mathcal{P}} \text{selectivity}(p_k(r_i, r_j)) \\ S_{ij} &= \text{selectivity}(P_{ij}) \end{aligned}$$

If there is no $p_k(r_i, r_j) \in \mathcal{P}$, then P_{ij} = true and $S_{ij} = 1$.

An *operator tree* P has the following syntax:

$$\begin{array}{rrcl} & P & ::= & r_i \\ \text{or} & P & ::= & P_1 \bowtie_{pred} P_2 \end{array}$$

where P_1 and P_2 are operator trees and *pred* is either true or is a conjunction of join predicates that reference the relations in P_1 and P_2 exclusively.

A *valid operator tree* for a query $Q = (\mathcal{R}, \mathcal{P})$ is an operator tree that uses each relation $r_i \in \mathcal{R}$ exactly once and each join predicate in $\mathcal{P}$ at least once. We do not allow node sharing in operator trees, even though a tree with shared nodes may result to better performance.

The size of an operator tree node is defined by the following recursive equations:

$$\begin{array}{lcl} \text{size}(r_i) & = & \text{the cardinality of } r_i \\ \text{size}(P_1 \bowtie_{pred} P_2) & = & \text{size}(P_1) \times \text{size}(P_2) \times \text{selectivity}(pred) \end{array}$$

That is, the size of a tree node is the size of the intermediate result produced by the operation of this node.

The *capacity* of an operator tree is the sum of size($P_1 \bowtie_{pred} P_2$) for every node $P_1 \bowtie_{pred} P_2$ in the tree:

$$\begin{array}{lcl} \text{capacity}(r_i) & = & 0 \\ \text{capacity}(P_1 \bowtie_{pred} P_2) & = & \text{capacity}(P_1) + \text{capacity}(P_2) + \text{size}(P_1 \bowtie_{pred} P_2) \end{array}$$

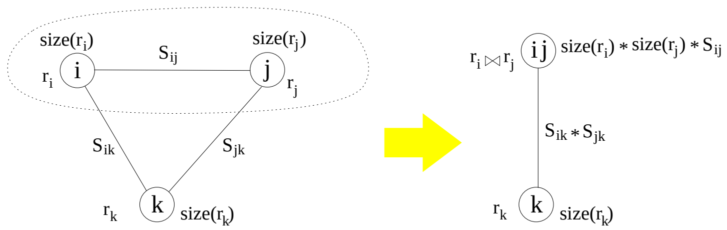
In database terminology, this is the total size of the intermediate results. Note that trees that are equal modulo commutativity have equal capacities.

A *tree of minimum capacity* for a query Q is a valid operator tree for Q that has the least capacity among all valid operator trees for Q. Our goal is to construct an operator tree for Q that has very low capacity, preferably a minimum capacity.

2.2 The GOO Algorithm

Our join ordering algorithm is called *GOO*, which stands for a Greedy Operator Ordering algorithm. It is a greedy bottom-up algorithm, similar to Kruskal's minimum spanning tree algorithm. The main data structure that GOO uses is the *query graph* of a query Q. In the query graph of Q, each relation r_i corresponds to a node i and each join predicate P_{ij} between i and j corresponds to a graph edge between i and j. Each node i is labeled by the size of i and each edge between i and j is labeled by S_{ij} (the selectivity of P_{ij}). Edges of $S_{ij} = 1$ are not displayed in the graph. For example, Figure 1.A shows a query graph with 9 relations.

The GOO algorithm constructs the optimal operator tree by merging pairs of graph nodes, as it is shown in the following figure:



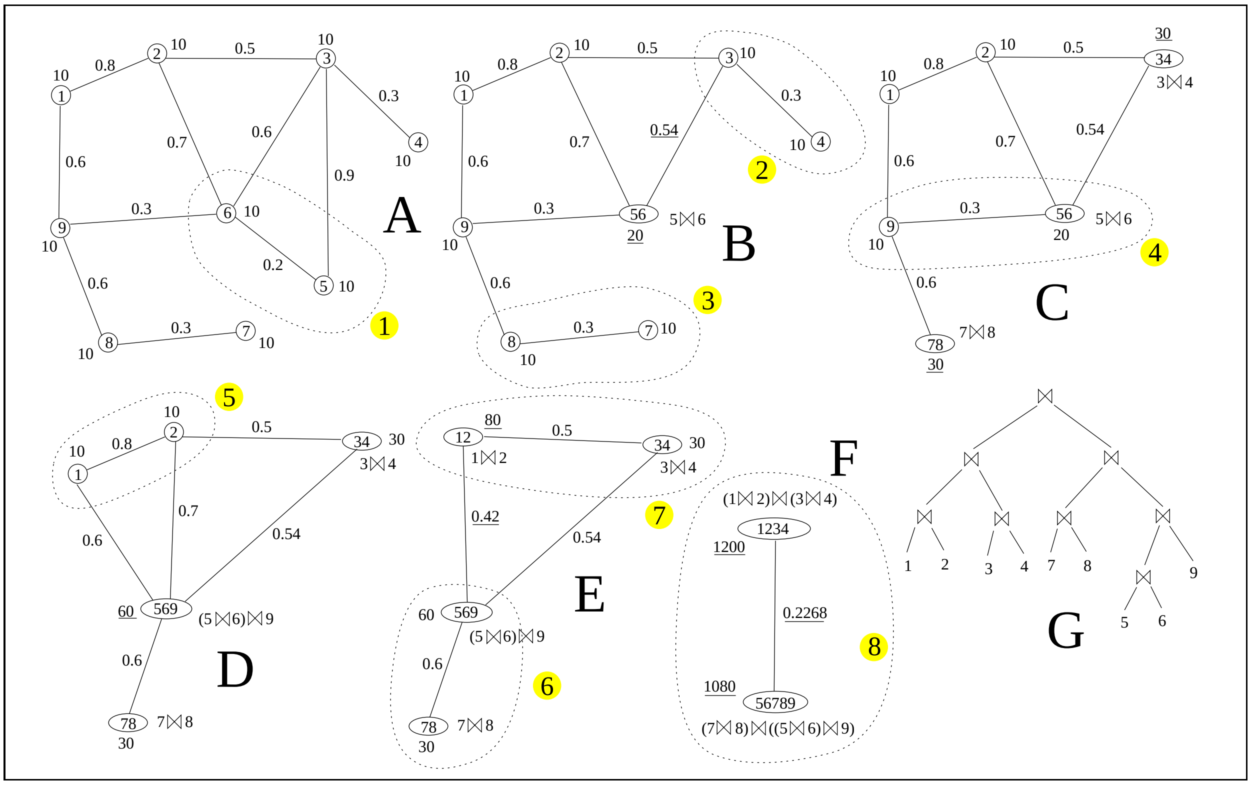


Figure 1: Example of Join Ordering

At each step of the algorithm, we select two nodes i and j that have a minimum value of size$(r_i) \times$size$(r_j) \times S_{ij}$. Then we merge these two nodes into a new node ij. The size of ij is size$(r_i) \times$ size$(r_j) \times S_{ij}$ and its operator tree is $r_i \bowtie r_j$. In addition, for each node k connected to both i and j, the weight of the edge between k and ij becomes $S_{ik} \times S_{jk}$.

For example, in Figure 1.A the nodes i and j that have the minimum value size$(r_i) \times$ size$(r_j) \times S_{ij}$ (equal to $10 \times 10 \times 0.2 = 20$) are 5 and 6. Therefore, in Figure 1.B we merge 5 and 6 into the node 56, which has size 20 and operator tree $5 \bowtie 6$ (a shorthand for $r_5 \bowtie_{P_{56}} r_6$). At the same time, the selectivity of the edge between 3 and 56 becomes $0.6 \times 0.9 = 0.54$. Next, in Figure 1.B, we group together the nodes 3 and 4 and then the nodes 7 and 8, yielding the graph in Figure 1.C. We continue this process until we are left with one node only. The operator tree of the final node is the resulting operator tree (shown in Figure 1.G).

The pseudo code of the GOO algorithm is given in Figure 2. The semantic brackets $[\![\ldots]\!]$ are used to construct operator trees and predicates. Information about graph nodes is given in a form of one-dimensional vectors while information about edges is given in a form of adjacency matrices. The algorithm starts with a query graph of N nodes, where N is the number of relations and, at each iteration of the loop (1) through (13), the graph shrinks by one node. Line (2) finds two nodes i and j with minimum value size$[i] \times$ size$[j] \times S[i,j]$. Lines (3) through (7) merge nodes i and j and replace node i with the new merged node. Lines (8) through (12) copy the last node n to node j, so that the graph shrinks by one node.

The complexity of this algorithm is $\mathcal{O}(N^3)$. In contrast to Kruskal's minimum spanning tree algorithm, we cannot use a heap to speed up the extraction of the minimum in line (2) because the weights of the edges

Algorithm 1 (GOO)
Input:

N	*number of relations*
predicate$[i,j]$	*the predicate, P_{ij}, between r_i and r_j (default is true)*
$S[i,j]$	*the selectivity, S_{ij}, of* predicate$[i,j]$ *(default is 1)*
tree$[i]$	*the operator tree for r_i (initially r_i)*
size$[i]$	*the size of r_i*

Output: *A valid operator tree with low capacity.*

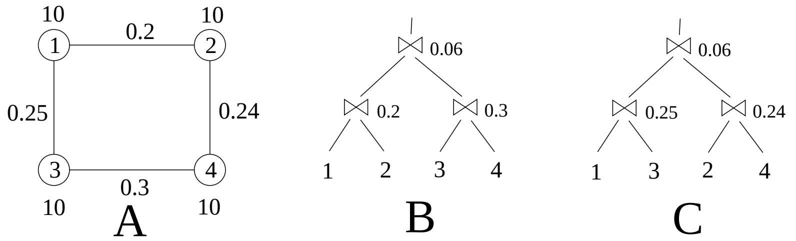
Algorithm:

(1) **for** $n \leftarrow N \ldots 2$ **do**
(2)   **find** $i,\ j \in [1 \ldots n],\ i \neq j$: size$[i] \times$ size$[j] \times S[i,j]$ *is minimum*
(3)   tree$[i] \leftarrow [\![$tree$[i] \bowtie_{\text{predicate}[i,j]}$ tree$[j]]\!]$
(4)   size$[i] \leftarrow$ size$[i] \times$ size$[j] \times S[i,j]$
(5)   **for** $k \leftarrow 1 \ldots n,\ k \neq i$ **do**
(6) $S[i,k] \leftarrow S[k,i] \leftarrow S[i,k] \times S[j,k]$
(7)   predicate$[i,k] \leftarrow$ predicate$[k,i] \leftarrow [\![$predicate$[i,k] \ \wedge$ predicate$[j,k]]\!]$
(8)   **for** $k \leftarrow 1 \ldots n,\ k \neq j$ **do**
(9) $S[j,k] \leftarrow S[k,j] \leftarrow S[n,k]$
(10)   predicate$[j,k] \leftarrow$ predicate$[k,j] \leftarrow$ predicate$[n,k]$
(11)   tree$[j] \leftarrow$ tree$[n]$
(12)   size$[j] \leftarrow$ size$[n]$
(13) **return** tree$[1]$

Figure 2: The GOO Algorithm

that are connected to both i and j nodes are modified at each iteration. Nevertheless, the GOO algorithm can derive a join order for 100 joins in 32 milliseconds (more performance results are given in Section 4).

The GOO algorithm is not optimal. The following example illustrates a case where our algorithm derives a suboptimal join tree:



The query graph is given in Figure A above and the join tree derived by GOO is given in B. The capacity of this tree is $20 + 30 + 36 = 86$. The optimal tree is given in C. It has capacity $25 + 24 + 36 = 85$.

As an alternative, we can use a real cost function in line (2) instead of the intermediate result size. In that case we would need a value cost$[i]$ to store the cost of a node i. Line (2) would need to be modified to find the minimum get_cost(size$[i]$, size$[j]$, $S[i,j]$) + cost$[i]$ + cost$[j]$, where get_cost is the cost function.

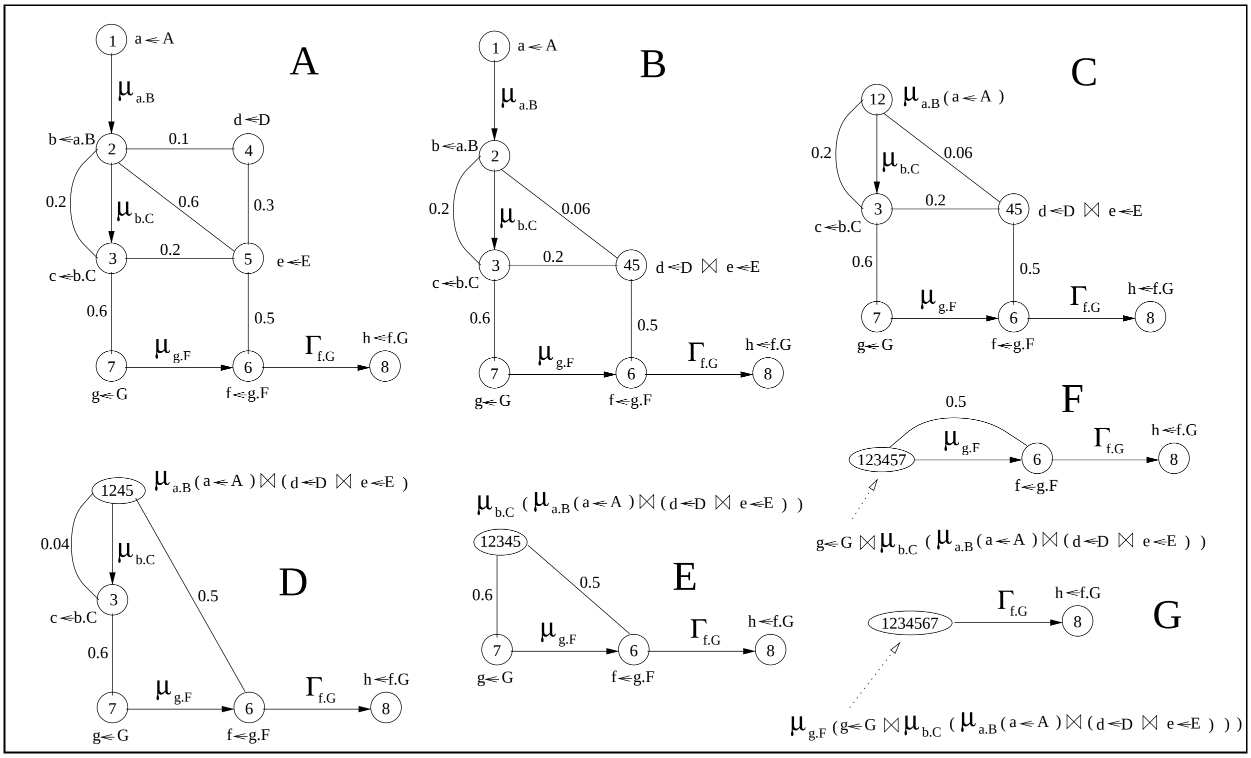


Figure 3: Deriving an OODB Operator Order

3 Ordering OODB Operations

The join ordering problem becomes more critical for OODBs, mostly because path expressions are usually translated into pointer joins, resulting to a large number of joins. In addition, the unnesting and nesting operators commute with joins and with each other, thus, increasing the number of possible permutations. This section generalizes the GOO algorithm to handle OODB operators along with joins.

The unnest operator $\mu^v_{A_i.path}$, where $A_i.path$ is a path expression of type set(t) and v is a variable, accepts a stream of tuples of type $< A_1 : t_1, \ldots, A_n : t_n >$ and constructs a stream of tuples of type $< A_1 : t_1, \ldots, A_n : t_n, v : t >$ connecting each tuple x of the input stream with every member of the set $x.A_i.path$. The unnest operator may use an optional predicate to restrict the output tuples. The nest operator $\Gamma^v_{A_i}$ groups the input by the range variable A_i, constructing a set of all tuples that are associated with the value of A_i. This set becomes the value of the range variable v. That is, this nest operator reads a stream of tuples of type $< A_1 : t_1, \ldots, A_n : t_n >$ and constructs a stream of tuples of type $< A_i : t_i, v : \text{set}(< A_1 : t_1, \ldots, A_n : t_n >) >$ by grouping together all tuples with the same value A_i.

The GOO algorithm can be extended to handle the nest and unnest operators along with joins. For an OODB query, the nodes of the query graph are the range variables in the query, associated not only with extents, but with the nest and unnest operators as well. A dependency between two range variables is denoted by an arrow in the query graph. For example, the following OQL query:

```
group x in ( select f
                from  a in A, b in a.B, c in b.C, d in D, e in E, g in G, f in g.F
               where  ... )
   by  ( h: x.G )
```

corresponds to the query graph in Figure 3.A (with some additional predicates). For example, the range variable a that ranges over the extent A corresponds to node 1, while b that ranges over a.B depends on a and, therefore, node 2 of b must depend on node 1.

A range variable is associated with an average cardinality if it ranges over a nested collection domain. For example, if the FROM-clause of an OQL query is a **in** A, b **in** a.B, c **in** b.C, then the cardinality of the range variable a is the cardinality of A, the cardinality of b is the average cardinality of all a.B, and the cardinality of c is the average cardinality of all b.C, for every b in a.B. Similarly, the nest operator has a cardinality between zero and one, which is the inverse average cardinality of the produced inner collection.

The GOO-OO algorithm that sorts OODB operators is a simple extension of the GOO algorithm. It is presented in Figure 4. It uses the additional vectors depends_on and dependency to handle range variable dependencies (represented by arrows in the query graph). The algorithm groups together two nodes if these two nodes do not depend on any other node (but one may depend on the other). If one node depends on the other, then we generate the appropriate OODB operation (nest or unnest); otherwise we generate a join.

Figure 3 displays the step for deriving the OODB operator tree for our example query (extent sizes are omitted). The complexity of GOO-OO is still $\mathcal{O}(n^3)$, where n is the number of range variables.

4 Performance Evaluation

In this section we evaluate the performance of the GOO algorithm. The tests reported here were executed on an Ultra Sparc 166MHz with 64MB memory running Solaris 2.5.1. The program that produced these results is available at: `http://www-cse.uta.edu/~fegaras/order/`.

The query graphs used in our experiments varied in two parameters: the number of relations and the maximum allowable number of edges attached to a node (called *fan-out*). We require that each node be attached to at least one edge and the query graph be connected (i.e., there is a path between every pair of nodes). The last restriction was set to reflect real queries and to avoid large cost values that may result due to cartesian products. The size of a relation associated with a node has a uniform distribution between 1 and 100 blocks and the value $-\log s$ of the selectivity s associated with an edge has a uniform distribution between 0 and 5.

We compared four different algorithms in our performance evaluation:

1. **GOO**: the GOO algorithm (applied to relational queries only);
2. **II**: the iterative improvement method that applies the downhill improvement phase to various random joins (as it will be described in detail below);
3. **GOOI**: the GOO algorithm followed by a downhill improvement phase;
4. **ES**: exhaustive search (generate-and-test).

Algorithm 2 (GOO-OO)
Input:

N	*number of relations*
predicate$[i,j]$	*the predicate, P_{ij}, between r_i and r_j (default is true)*
$S[i,j]$	*the selectivity, S_{ij}, of* predicate$[i,j]$ *(default is 1)*
tree$[i]$	*the operator tree for r_i (initially r_i)*
size$[i]$	*the size of r_i*
depends_on$[i]$	*equal to j, if r_i depends on r_j, otherwise it is 0*
dependency$[i]$	*the type of dependency (μ_p or Γ_p) if* depends_on$[i] \neq 0$

Output: *A valid OODB operator tree with low capacity.*

Algorithm:

for $n \leftarrow N \ldots 2$ **do**
    find $i,\, j \in [1 \ldots n] : i \neq j \;\wedge\; (\text{depends\_on}[i] = 0 \;\vee\; \text{depends\_on}[i] = j)$
                        $\wedge\; \text{depends\_on}[j] = 0 \;\wedge\; (\text{size}[i] \times \text{size}[j] \times S[i,j])$ is minimum
    if depends_on$[i] = j$
      then tree$[i] \leftarrow [\![\text{dependency}[i]_{\text{predicate}[i,j]}(\text{tree}[j])]\!]$
      else tree$[i] \leftarrow [\![\text{tree}[i] \bowtie_{\text{predicate}[i,j]} \text{tree}[j]]\!]$
    size$[i] \leftarrow$ size$[i] \times$ size$[j] \times S[i,j]$
    for $k \leftarrow 1 \ldots n,\; k \neq i$ **do**
        $S[i,k] \leftarrow S[k,i] \leftarrow S[i,k] \times S[j,k]$
        predicate$[i,k] \leftarrow$ predicate$[k,i] \leftarrow [\![\text{predicate}[i,k] \;\wedge\; \text{predicate}[j,k]]\!]$
    for $k \leftarrow 1 \ldots n,\; k \neq j$ **do**
        $S[j,k] \leftarrow S[k,j] \leftarrow S[n,k]$
        predicate$[j,k] \leftarrow$ predicate$[k,j] \leftarrow$ predicate$[n,k]$
    tree$[j] \leftarrow$ tree$[n]$
    depends_on$[j] \leftarrow$ depends_on$[n]$
    dependency$[j] \leftarrow$ dependency$[n]$
    size$[j] \leftarrow$ size$[n]$
return tree$[1]$

Figure 4: The GOO-OO Algorithm

The downhill improvement phase used in II and GOOI improves joins using the following rules:

$$
\begin{array}{lrcl}
1 & A \bowtie (B \bowtie C) & \rightarrow & (A \bowtie B) \bowtie C \\
2 & (A \bowtie B) \bowtie C & \rightarrow & A \bowtie (B \bowtie C) \\
3 & A \bowtie (B \bowtie C) & \rightarrow & B \bowtie (A \bowtie C) \\
4 & (A \bowtie B) \bowtie C & \rightarrow & (A \bowtie C) \bowtie B
\end{array}
$$

A rule is applied to a join tree node only if it improves the cost. Instead of randomly selecting a rule from these rules and applying it to a random node in the join tree, our downhill improvement phase improves the join tree in a bottom-up fashion (from leaves to root) as follows:

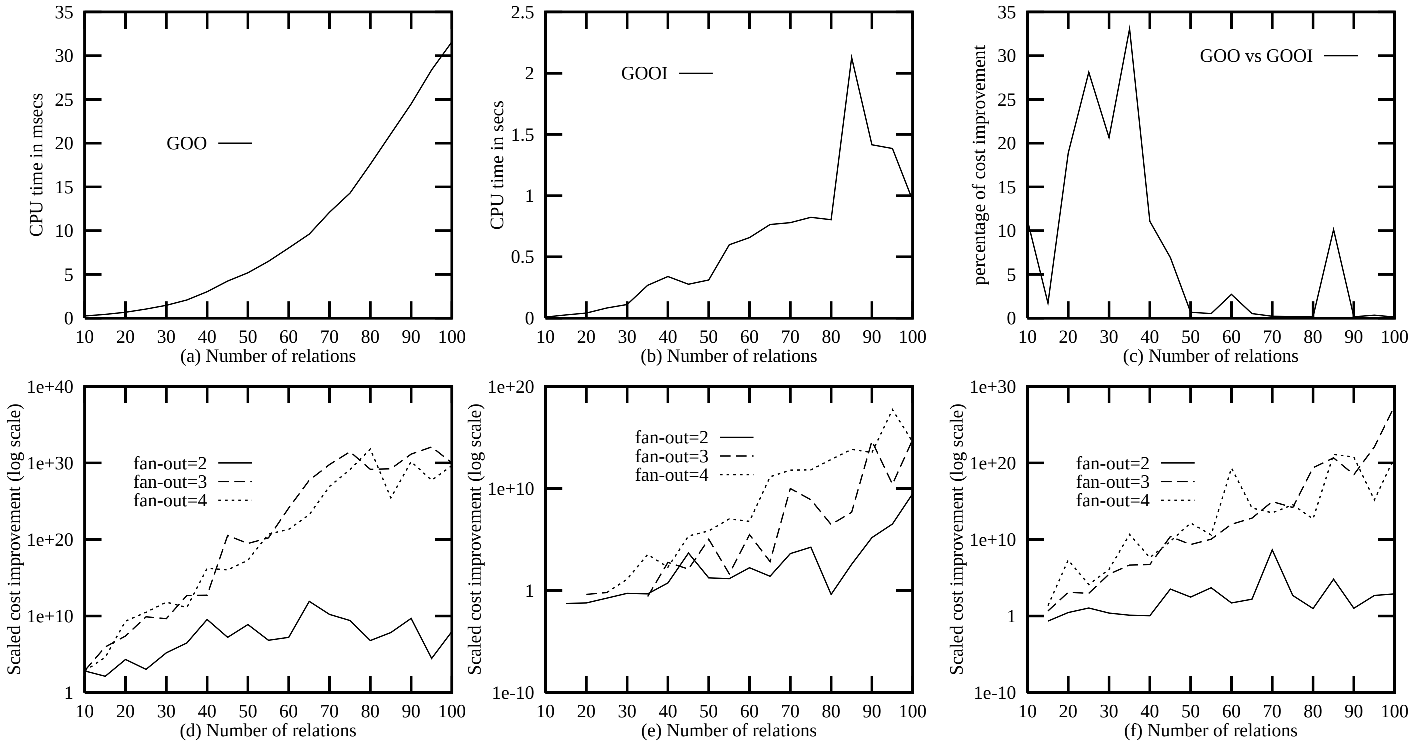


Figure 5: Performance Results

```
improve ( plan x )
    if x has the form join(left,right) then
    { new_left ← improve(left);
      new_right ← improve(right);
      create new plans by applying all applicable rewrite rules (Rules 1 through 4);
      if any of these rules constructs a plan with a lower cost than join(new_left,new_right)
          then return improve(p), where p is the plan with the lowest cost;
      else return join(new_left,new_right); };
```

According to our experiments, this version of downhill improvement results to a better performance for both II and GOOI.

Figure 5.a gives the running time performance (in milliseconds) of the GOO algorithm for query graphs of $10 \leq n \leq 100$ relations. The construction of the join from a query graph is also included in the time computation. Each point on a curve represents the average time of ten experiments (over different random graphs). Figure 5.b gives the running time performance (in seconds) of the GOOI algorithm for query graphs of $10 \leq n \leq 100$.

In our experiments, we used only two types of joins: nested-loops joins and merge joins. Even though the cost model we used to evaluate performance is very simple, it still gives a good approximation of the I/O cost of the evaluation algorithms found in commercial databases. Since costing is very important when comparing optimization techniques, we present here the cost model in full detail.

Let $||plan||$ be the output size of the physical plan $plan$ (in number of blocks) and $\mathcal{C}_{plan}$ be the cost of executing the plan (in seconds). For a table T we have:

$$\begin{aligned} ||T|| &= \text{number of blocks in } T \\ \mathcal{C}_T &= ||T|| \times cb \end{aligned}$$

where cb is the time needed to access one disk block. For a join $R \bowtie_p S$, where R and S are plans and p is a join predicate, we have:

$$\begin{aligned} ||R \bowtie_p S|| &= ||R|| \times ||S|| \times \rho \\ \mathcal{C}_{R \bowtie_p S} &= \mathcal{C}_R + \mathcal{C}_S + ||R \bowtie_p S|| \times cb + \min(NLJ_1, NLJ_2, INL_1, INL_2, MJ) \times cb \end{aligned}$$

where ρ is the selectivity of the predicate p. The first two terms of the cost represent the costs of executing plans R and S. The third term is the cost of writing the result to an intermediate relation (we assume that all intermediate results are materialized; i.e., no pipelining is used). The last term comes from five different join algorithms: nested loops where the outer relation is R, nested loops where the outer relation is S, indexed nested loops where the outer relation is R, indexed nested loops where the outer relation is S, and merge join:

$$\begin{aligned} NLJ_1 &= ||R|| + \lceil \tfrac{||R||}{M-1} \rceil \times ||S|| \\ NLJ_2 &= ||S|| + \lceil \tfrac{||S||}{M-1} \rceil \times ||R|| \\ INL_1 &= ||R|| + B_R \times ||R|| \times \lceil \log ||S|| \rceil \\ INL_2 &= ||S|| + B_S \times ||S|| \times \lceil \log ||R|| \rceil \\ MJ &= (||R|| + ||S||) + (||R|| + 2 \times ||R|| \times \lceil \log \lceil \tfrac{||R||}{M} \rceil \rceil) + (||S|| + 2 \times ||S|| \times \lceil \log \lceil \tfrac{||S||}{M} \rceil \rceil) \end{aligned}$$

The nested-loops algorithm of NLJ_1 (and similarly of NLJ_2) reads the outer relation R in clusters of $M-1$ blocks, where M is the size of the available memory in blocks, and for each cluster it scans the entire inner relation S one block at a time. The indexed nested-loops algorithm of INL_1 (resp. INL_2) is used only when S (resp. R) is a table. (We assume that table S has an index over each of its join attributes.) Here B_S is the blocking factor of S, thus $B_S \times ||S||$ is the cardinality of S. The first term of MJ represents the I/O cost (in number of blocks) of merging R with S after these tables are sorted. The second term represents the number of blocks for sorting R: $\lceil \frac{||R||}{M} \rceil$ initial runs are created, which are merged in pairs. The creation of the initial runs requires $2 \times ||R||$ blocks and there are $\lceil \log \lceil \frac{||R||}{M} \rceil \rceil$ levels of merging and each level involves $||R||$ blocks, which are read and written once. If R is a table, then we do not take into account the cost of sorting R: that is, we assume that table R has an index over each of its join attributes. This simplifies the cost computation and justifies the use of merge join. Note that if R is another merge join, then, in our model, R in MJ is always sorted, which may be unnecessary if both merge joins are over the same attributes. This assumption is also made to simplify the cost calculation. The third term of MJ is for sorting S. Like the second term, if S is a table, then the third term is not used.

As a first experiment, we compared the quality of the plans produced by the GOO algorithm with the quality of the best plans found using exhaustive search (ES):

n	plans	2	3	4
4	15	0.01	0.69	0.35
5	105	0.02	1.08	5.77
6	945	0.01	14.16	11.69
7	10395	0.02	1.78	10.53

n	2	3	4
4	0	0.57	0.05
5	0	0.17	0.1
6	0.01	0.03	0.05
7	0.01	0.14	10.26

The left table compares GOO with ES while the right table compares GOOI with ES. The first column of the first table is the number of relations, while the second column is the number of different plans (they are different modulo commutativity) that were tested during the exhaustive search. The other columns are associated with different fan-outs. We tested random graphs of $4 \leq n \leq 7$ relations only, because for $n > 7$ exhaustive search becomes infeasible. Each table entry is the percentage of the scaled difference between the two costs: if t_1 is the cost of the plan produced by GOO (GOO or GOOI) and t_2 is the cost of the best plan, then the table entry is $(t_1 - t_2) * 100/t_2$. Like before, each table entry is the average of ten experiments. When generating a join tree during exhaustive search (as well as during iterative improvement), we push selectivities to the appropriate joins (i.e., we perform selections as early as possible). From these tables, we can see that GOOI generates plans that are as good as the best plans.

As another experiment, we compared GOO with GOOI (Figure 5.c). The values in Figure 5.c represent percentage of cost improvement: if t_1 is the cost of the plan produced by GOOI and t_2 is the cost of the plan produced by GOO, then the table entry is $(t_2 - t_1) * 100/t_1$. Like before, each value is the average value of ten experiments. We can see that GOOI results to at most 30% cost improvement but it requires at most 100 times more CPU time (if we compare Figures 5.a and 5.b).

As another demonstration of the effectiveness of our algorithm, we compared the quality of the produced plans with the quality of the plans produced by the iterative improvement method (II), when the latter is allowed to run ten times longer than GOO (Figure 5.d). The II algorithm works as follows: a random join is selected and it is improved by the downhill improvement phase. When no improvement can be made to any node, the iterative improvement algorithm chooses another random plan and performs the same process. This task is repeated until the total running time of II exceeds the specified time limit (ten times the running time of GOO). At the end, the best plan found during all iterations is returned. The values in Figure 5.d represent scaled costs: if t_1 is the cost of the plan produced by GOO and t_2 is the cost of the plan produced by II, then the table entry is $(t_2 - t_1)/t_1$. Like before, each value is the average value of ten experiments. We can see that GOO clearly beats iterative improvement despite the time advantage given to II.

Figure 5.e compares GOOI with II, when II is allowed to run ten times longer than GOOI. The values in the figure represent scaled costs: if t_1 is the cost of the plan produced by GOOI and t_2 is the cost of the same plan improved by the improvement phase of the iterative improvement algorithm, then the value is $(t_1 - t_2)/t_2$. Like before, each value is the average value of ten experiments. The parts of the curve between 10 and 25 relations that are not shown correspond to negative values between 0 and -0.5. We can see that here too GOOI clearly beats II. The reason that the values in Figure 5.e are less than the values in Figure 5.d is that GOOI takes about 100 times more time than GOO so II is allowed to run longer with GOOI than

with GOO.

Figure 5.f is the result of allowing II to run one hundred times longer than GOO. The difference from Figure 5.d is not substantial. The parts of the curve for 10 relations that are not shown correspond to negative values between 0 and -0.5. This indicates that when II is bad is really bad (GOO plans often have many orders of magnitude lower cost than II plans) and when II is good is not substantially better than GOO (II plans are at most 50% better than GOO plans). Consequently, the GOO algorithm is more consistent and stable in optimizing large queries.

5 Related Work

Many heuristics have been proposed in the literature on join ordering for large relational queries, even though most of them deal with acyclic query graphs and generate deep-left trees only [10, 21]. A comparison between these heuristics as well as between various randomized algorithms can be found in [18]. It has been shown that if there is no restriction on the query graphs, then the problem of generating the optimal plan is NP-hard [16]. Most commercial relational optimizers are based on the dynamic programming approach of System R [17] and they cannot handle queries of more than ten tables. A notable exception is Starburst [14], which uses various polynomial algorithms based on the query graph shape (such as, for linear queries) and is able to optimize queries containing up to 110 tables. For other shapes, their proposed dynamic algorithm becomes exponential ($\mathcal{O}(2^n)$).

The only heuristic join ordering algorithm for general (cyclic) graphs that we are aware of is the KBZ algorithm by Krishnamurthy, Boral, and Zaniolo [10], which in turn is based on the IK algorithm of Ibaraki and Kameda [7]. The IK algorithm finds the optimal deep-left tree of an acyclic graph under certain cost functions in polynomial time by assigning ranks to relations and ordering the relations according to rank. The KBZ algorithm is an $\mathcal{O}(n^2)$ heuristic that finds the minimum spanning tree of a cyclic graph and applies the IK algorithm to the resulting tree. The minimum spanning tree used in this algorithm considers the minimum product of edge weights (selectivities) rather than the minimum sum. We believe that our join ordering algorithm is superior to the KBZ algorithm because it does not pose any restrictions to cost functions, it generates bushy trees, and it considers sizes of intermediate results instead of selectivities only. Recently, the KBZ algorithm has been extended by Scheufele and Moerkotte to include expensive predicates [15].

In a previous work [5], we have presented a top-down heuristic for ordering joins. It applies the min-cut algorithm recursively to split the query graph into two query subgraphs each time so that the product of selectivities between the relations of these two subgraphs is maximum. For each such cut, the algorithm generates a join. That way, selective joins are done as early as possible while non-selective joins (such as cartesian products) are left for the end. Our new algorithm is superior to this algorithm, because it considers cardinalities. In addition, our new algorithm is more efficient than the old one.

Ioannidis and Kang [9, 8] proposed various combinatorial optimization techniques for large relational queries based on iterative improvement and simulated annealing. Iterative improvement consists of a number of local optimizations, where each local optimization starts at a random state and performs downhill moves until it reaches a local minimum. Simulated annealing starts at a random state and proceeds by random moves, but if the move is uphill it is accepted by a certain probability, which decreases over time. The iterative improvement method does not consider states that are promising as starting points, while the simulated annealing method does not address the problem of selecting a random uphill state at each state transition. For both algorithms, the quality of the plan found at each iteration depends in a great extent on the quality of the initial state. In addition, there is no guarantee that a random initial state will not fall in the neighborhood of a previously examined state, thus resulting to the same local minimum. To be effective, a large number of initial states should be tried as starting points so that the computed local minima would include a fair number of deep minima. Alternatively, a good quality starting point generated by a heuristic algorithm can be used that promises a deep local minimum. The work by Swami and Gupta [20, 19] compares various heuristics and combinatorial algorithms. Their experiments suggest that iterative improvement is the best combinatorial optimization technique. Morzy, Matysiak, and Salza [12] improved the iterative improvement algorithm by maintaining a list of constraints between iterations (called a tabu list) to avoid the repetition of non profitable moves.

6 Conclusion and Future Work

We have presented a polynomial-time heuristic algorithm that generates good quality plans for OODB queries. The algorithm has been tested on various random relational query graphs. Our preliminary results show that our method is clearly superior to the iterative improvement method. As a future work, we are planning to incorporate this algorithm to the experimental query optimizer for OQL being developed at the University of Texas at Arlington [4]. We are also planning to combine this algorithm with a local search technique, which improves small fragments of the query plan by performing a real cost analysis.

Acknowledgements: This work is supported in part by the National Science Foundation under grant IRI-9509955.

References

[1] R. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1994.

[2] S. Cluet and C. Delobel. A General Framework for the Optimization of Object-Oriented Queries. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Diego, California*, pp 383–392, June 1992.

[3] S. Cluet and G. Moerkotte. Nested Queries in Object Bases. In *Fifth International Workshop on Database Programming Languages, Gubbio, Italy*, September 1995.

[4] L. Fegaras. An Experimental Optimizer for OQL. University of Texas at Arlington Technical Report TR-CSE-97-007. Available at `http://www-cse.uta.edu/ ~fegaras/oqlopt.ps.gz`, May 1997.

[5] L. Fegaras. Optimizing Large OODB Queries. In *Fifth International Conference on Deductive and Object-Oriented Databases, Montreux, Switzerland*, December 1997. Available at `http://www-cse.uta.edu/ ~fegaras/dood97.ps.gz`.

[6] L. Fegaras. Query Unnesting in Object-Oriented Databases. *ACM SIGMOD International Conference on Management of Data, Seattle, Washington*, June 1998.

[7] T. Ibaraki and T. Kameda. Optimal Nesting for Computing N-Relational Joins. *ACM Transactions on Database Systems*, 9(3):482–502, September 1984.

[8] Y. Ioannidis and Y. Kang. Randomized Algorithms for Optimizing Large Join Queries. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ*, pp 312–321, May 1990.

[9] Y. Ioannidis and Y. Kang. Left-deep vs. Bushy Trees: An Analysis of Strategy Spaces and its Implications for Query Optimization. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Denver, Colorado*, 20(2):168–177, May 1991.

[10] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of Nonrecursive Queries. In Y. Kambayashi, editor, *Proceedings of the Twelfth International Conference on Very Large Databases, Kyoto, Japan*, pp 128–137, August 1986.

[11] R. Lanzelotte, P. Valduriez, and J. Ziane, M. Cheiney. Optimization of Nonrecursive Queries in OODBs. *Deductive and Object-Oriented Databases, Munich, Germany*, pp 1–21, 1991.

[12] T. Morzy, M. Matysiak, and S. Salza. Tabu Search Optimization of Large Join Queries. In M. Jarke, J. Bubenko, and K. Jeffery, editors, *Advances in Database Technology – EDBT '94*, pp 309–322. Springer-Verlag, 1994. LNCS 779.

[13] M. Muralikrishna. Improved Unnesting Algorithms for Join Aggregate SQL Queries. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 91, Vancouver, BC, Canada, August 1992.

[14] K. Ono and G. Lohman. Measuring the Complexity of Join Enumeration in Relational Query Optimization. In *Proceedings of the 16th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Brisbane*, August 1990.

[15] W. Scheufele and G. Moerkotte. Efficient Dynamic Programming Algorithms for Ordering Expensive Joins and Selections. *International Conference on Extending Data Base Technology, Valencia, Spain*, March 1997.

[16] W. Scheufele and G. Moerkotte. On the Complexity of Generating Optimal Plans with Cross Products. *Proceedings of the 16th ACM Symposium on Principles of Database Systems, Tucson, Arizona*, pp 238–248, May 1997.

[17] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Boston, Massachusetts*, pp 23–34, May 1979.

[18] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. *VLDB Journal*, 6(3):191–208, 1997.

[19] A. Swami. Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon*, pp 367–376, May 1989.

[20] A. Swami and A. Gupta. Optimization of Large Join Queries. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, Illinois*, 13(3):8–17, September 1988.

[21] K. Youssefi and E. Wong. Query Processing in a Relational Database Management System. In *Proceedings of the Fifth International Conference on Very Large Databases*, pp 409–417, 1979.