Optimization of Large OODB 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 ordering of OODB algebraic operators. It is based on the min-cut algorithm, which cuts a graph into two subgraphs in such a way that the total weight of the edges being cut is minimum. This algorithm is applied recursively to a query graph, generating a join at each cut. It takes in account the dependencies between range variables and generates the proper nested relational operations (nest or unnest) to satisfy these dependencies.

Index Terms: query optimization, large queries, object-oriented databases, nested relational algebra, iterative improvement, combinatorial optimization.

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 ($\mathcal{O}(n!)$ for an n-way 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 [15]) 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, 9]. 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 [12]. 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 in 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 [8]. Even though there are many proposals for optimizers that consider different orders for OODB algebraic operators (see for example [9] and [5]), to our knowledge, none of these methods is actually non-exponential. 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 [7], can be used to extract a solution in a fixed amount of time. 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. It has been noted that "the solution space has a large number of local minima, with a small but significant fraction of them being deep local minima" [17]. As the result of that, for these algorithms 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. 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 algorithm that generates a "good quality" ordering of OODB algebraic operators. It can also be used to order the joins of a large relational query. An OODB query can be represented by a query graph whose nodes represent range variables and the edges represent predicates that relate range variables. The dependencies between the range variables, such as when a variable is bound to a set and another variable ranges over this set, are indicated by arrows in the query graph. Our operator ordering algorithm is based on the min-cut algorithm [16], which cuts a graph into two subgraphs in such a way that the total weight of the edges being cut is minimum. This algorithm is applied recursively to a query graph, generating one join at each cut. The cuts that do not satisfy the dependencies are rejected and the arrows in the graph are translated into unnest/nest operations. This algorithm can be easily extended with local search methods, which improve small clusters of operators of fixed size by using real cost functions and by considering different evaluation algorithms and access paths.

The paper is organized as follows. Section 2 presents our polynomial-time heuristic, based on the min-cut algorithm, that generates a good join order for a relational query graph. Section 3 presents various extensions to this algorithm to capture implied predicates and conjunctive queries. Section 4 extends the min-cut algorithm to handle OODB query graphs, which may include unnest and nest operators along with joins. Section 5 presents the detailed algorithm using pseudo-code. Section 6 compares the performance of our algorithm (applied to relational queries only) with that of the iterative improvement algorithm. Finally, Section 7 compares our work with previous work.

2 Finding a Good Join Order for a Relational Query

Let $join(R_1, \ldots, R_n, pred)$ be an n-way relational join, where R_i is a relation of cardinality c_i and the join predicate $pred$ is a conjunction of simple predicates of the form $f(R_i, R_j)$, that is, a predicate that relates R_i with R_j tuples. This join can be represented by a query graph, G, which allocates one node for each R_i and one edge between R_i and R_j for each simple predicate $f(R_i, R_j)$ in $pred$. (More complex predicates, such as $R_1.A = R_2.B + R_3.C$, require the use of hypergraphs, which add more complexity to the graph algorithms described in this paper.) We assume that local predicates (i.e., predicates that refer to one table only, such as, name='Smith') have been incorporated into the appropriate table scan operators. That way, when we are talking about table cardinality we will be actually talking about *effective* table cardinality, i.e., the cardinality of a table after all local predicates have been applied.

An edge between R_i and R_j is labeled by the selectivity factor [15] of this predicate (a number between 0 and 1), denoted by s_{ij}. The selectivity of a predicate is the fraction of tuples that satisfy this predicate. If two tables are not connected by some predicate, then the selectivity between them is 1 (a cartesian product). Edges of selectivity one are not displayed in a query graph. Figure 1.A shows an example of a query graph (the dashed lines will be explained later).

The size of a query graph G, size(G), is defined to be the cardinality of the output of the n-way join that corresponds to G. Since the selectivity of the predicate *pred* of the n-way join is the product of the selectivities of its constituent simple predicates, the size of G is $(\prod_{R_i \in G} c_i) \times (\prod_{R_i \in G, R_j \in G} s_{ij})$. The size estimation becomes a little bit more complicated in the presence of implied predicates, such as in the following WHERE-clause:

where x.A=y.B **and** y.B=z.C **and** x.A=z.C

Each of the above three predicates is implied by the other two. All three predicates should appear in the query graph as graph edges but only the uncorrelated selectivities should be used in the size and cost estimation, as it will be described in detail in Section 3.

Since we are interested in deriving binary joins from the query graph G, we would like to cut this graph into two subgraphs, G_1 and G_2, find a join tree for each subgraph using the same method recursively, and use a join to connect the two join trees. That is, we would like to use a divide-and-conquer algorithm. The predicate of the join that connects the two subgraphs G_1 and G_2 is the conjunction of all the simple predicates that correspond to the edges between G_1 and G_2 (i.e., the edges of the cut).

If we use a naive nested-loops evaluation technique for joins that does not make use of indexes, then the estimated cost of the derived join tree, cost(G), is:

$$\text{cost}(G) \quad = \quad \text{cost}(G_1) + \text{cost}(G_2) + \text{size}(G_1) \times \text{size}(G_2) + \text{size}(G)$$

(More realistic cost functions will be used in Section 6 when we evaluate the actual quality of the produced plans.) The above equation indicates that the cost of the join tree is the cost of the left subtree plus the cost of the right subtree plus the cost of the nested-loops join plus the cost of writing the output to an intermediate table. The last term does not depend on the way we cut the graph.

We would like to cut the graph in a way that cost(G) becomes minimum. The third term of cost(G) is:

$$\begin{aligned} \text{size}(G_1) \times \text{size}(G_2) &= \prod_{R_i \in G} c_i \times \prod_{R_i \in G_1, R_j \in G_1} s_{ij} \times \prod_{R_i \in G_2, R_j \in G_2} s_{ij} \\ &= \prod_{R_i \in G} c_i \times \prod_{R_i \in G, R_j \in G} s_{ij} / \prod_{R_i \in G_1, R_j \in G_2} s_{ij} \\ &= \text{size}(G) / \prod_{R_i \in G_1, R_j \in G_2} s_{ij} \end{aligned}$$

This term becomes minimum when the product of the selectivities of all the edges between G_1 and G_2, $\prod_{R_i \in G_1, R_j \in G_2} s_{ij}$, is maximum. When this product becomes maximum, the first two terms, cost(G_1) + cost(G_2), of cost(G) are very *likely* to become minimum, because, jointly, G_1 and G_2 would have a lower product of selectivities since we eliminated the weak (maximum) selectivities, thus leaving the strong selectivities to G_1 and G_2. That is, we perform selective joins as early as possible and we perform non-selective joins as late as possible, the latest being the cartesian products. This is very similar to System R's approach to generating join orders.

If we map the edge labels s_{ij} of G into the weights $s'_{ij} = -\log s_{ij}$, then the problem of cutting the graph G into two subgraphs G_1 and G_2 such that the $\prod_{R_i \in G_1, R_j \in G_2} s_{ij}$ is maximum becomes the minimum

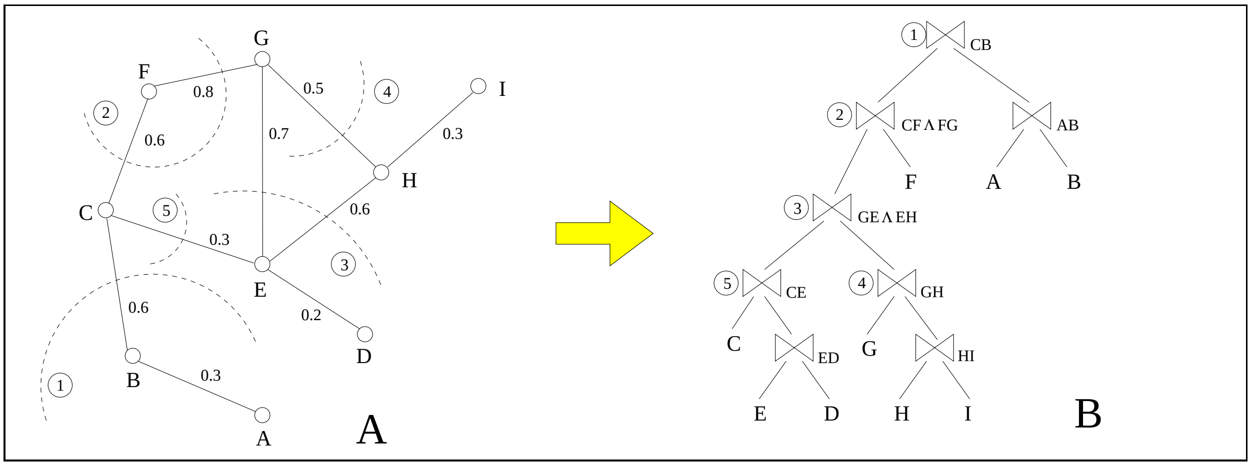

Figure 1: Deriving a Join Order Using the Min-Cut Algorithm

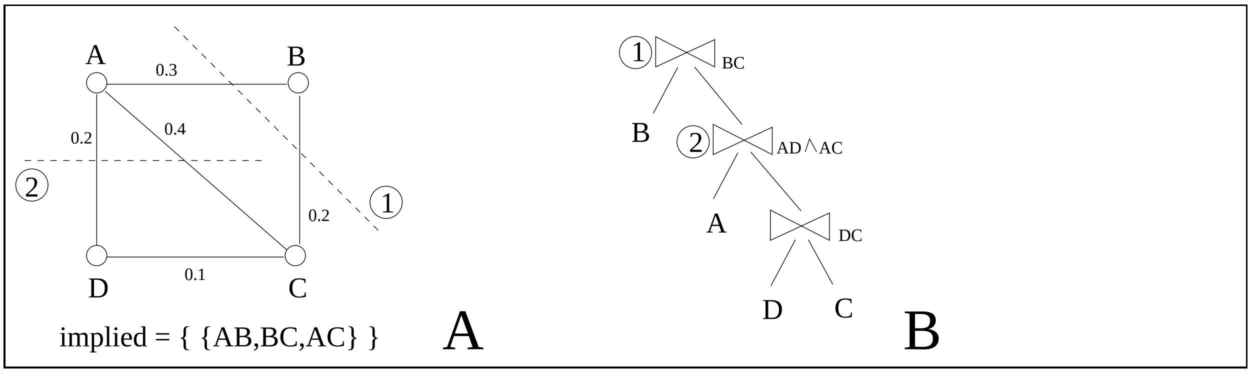

Figure 2: Optimizing a Query with Implied Predicates

cut problem where we cut the graph in a way that $\sum_{R_i \in G_1, R_j \in G_2} s'_{ij}$ is minimum. (Notice that edges of selectivity one, become edges of zero weight, which are ignored.) Fortunately, there is a polynomial algorithm for finding the minimum cut of a graph, called the *min-cut algorithm* [16]. This algorithm, described in detail in [10], runs in time $\mathcal{O}(nm + n^2 \log n)$, where n is the number of nodes and m is the number of edges. We use this algorithm to recursively cut a graph into two subgraphs until we derive a graph with one or two nodes. Therefore, the total running time of the join tree construction is $\mathcal{O}(n^2 m + n^3 \log n)$.

Figure 1 illustrates how the min-cut algorithm can be used to construct a join tree. The min-cut algorithm maps the query graph in Figure 1.A into the join tree in Figure 1.B. There are five cuts (denoted by dashed lines) labeled from one to five (the circled numbers in the figure). The first cut eliminates the edge with selectivity 0.6, which is the maximum selectivity; the second cut eliminates the edges with selectivities 0.6 and 0.8 (since the weight of the cut is $0.6 * 0.8 = 0.48$, which is next in order), etc. The join predicates correspond to the edges being cut off and are represented in Figure 1.B by the edge ends.

3 Various Extensions and Improvements

The min-cut algorithm needs to be slightly modified to cope with implied predicates, such as the first three predicates of the following WHERE-clause:

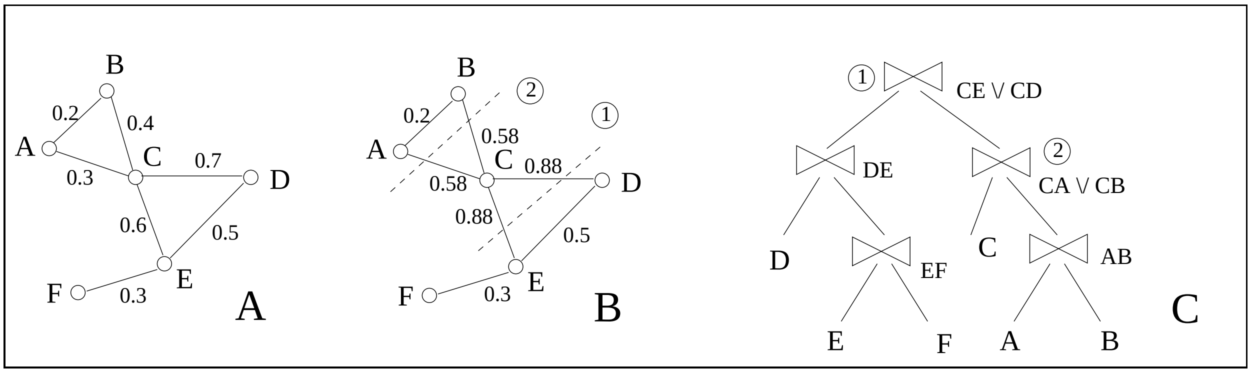

Figure 3: Optimizing a Conjunctive Query with a Predicate $AB \wedge (CA \vee CB) \wedge DE \wedge EF \wedge (CE \vee CD)$

where A.X = B.Y **and** B.Y = C.Z **and** A.X = C.Z **and** A.W = D.M **and** C.V = D.N

In this example, each of the first three predicates is implied by the other two. All implied predicates should appear in the graph with the proper selectivities since otherwise the involved joins would be handled as cartesian products. To handle this case properly, the min-cut algorithm maintains a list of all equivalent classes of the graph edges that denote implied predicates, called the *implied* list. For example, the previous WHERE-clause corresponds to the query graph and implied list in Figure 2.A. When a cut is considered by the min-cut algorithm, the implied list is consulted as follows: for each equivalent class in the implied list, the edge with the *maximum* selectivity among all edges that belong to both the cut and the equivalent class is not considered in the computation of $\prod_{R_i \in G_1, R_j \in G_2} s_{ij}$. When a particular cut is selected, all the equivalent classes referred to by the cut are removed from the implied list. For example, the best cut in the above graph is the cut with label one, since it has selectivity 0.2 (we do not take into account the selectivity 0.3 of AB). After the first cut is selected, the implied list becomes empty. Consequently, we take into account all the edges of the second cut. Figure 2.B displays the final join tree. Note that our approach of handling implied predicates is different than that of Swami and Schiefer [19], which always removes the edge of maximum selectivity (which in our example is the edge AC).

Surprisingly, the same algorithm that handles implied predicates can be used with minor modifications to handle disjunctive queries. The first step is to put the query predicate in conjunctive normal form, as it is done for the predicate $AB \wedge (CA \vee CB) \wedge DE \wedge EF \wedge (CE \vee CD)$ of the query graph in Figure 3.A. That is, the query predicate is put into the form: $\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k_i} p_{ij}$, where p_{ij} are simple predicates. For each disjunction $\bigvee_{j=1}^{k_i} p_{ij}$ with $k_i > 1$, we create a set of predicates $D_i = \{p_{i1}, \ldots, p_{ik_i}\}$, called a *disjunction set.* The weight of an edge in the query graph that corresponds to a predicate $p_{ij} \in D_i$ is $\sum_{j=1}^{k_i} c_{ij} - \prod_{j=1}^{k_i} c_{ij}$, where c_{ij} is the selectivity of the predicate p_{ij} (this formula is derived by handling selectivities as probabilities [15]). That is, all edges in D_i are assigned the same weight. For example, the graph in Figure 3.A is mapped into the graph of Figure 3.B because we have two disjunction sets: the set $D_1 = \{CA, CB\}$ with selectivity $0.6 + 0.7 - 0.6 \times 0.7 = 0.88$ and the set $D_2 = \{CE, CD\}$ with selectivity $0.3 + 0.4 - 0.3 \times 0.4 = 0.58$. If a cut of the min-cut algorithm cuts more than one edge from the same disjunction set, the selectivity is used only once. When a cut is selected, for each disjunction set that has at least one edge in the cut, the disjunction of all the predicates in the set is used in the join predicate of this cut and all the edges of the set are removed from the graph. For example, the first cut in Figure 3.B has the maximum weight, 0.88, since CA and CB belong to the same disjunction set D_1 and, thus, 0.88 counts only once. The predicate of the join that corresponds to this cut is $CA \vee CB$. After this cut is selected, all edges in D_1 are removed from the graph. The second cut does the same for D_2. The resulting join is assigned the predicate $CE \vee CD$.

There are many ways of improving the min-cut algorithm without making it exponential. One way is to consider the best two cuts each time (instead of using the best cut only) but only for a constant number of levels k from the join tree root. This adds a 2^k factor to the computational complexity. An alternative is to use local search to improve small clusters of joins of constant size using an optimizer that considers all permutations, and all available access paths and algorithms.

After a join tree is constructed by the min-cut algorithm, a transformation-based query optimizer can be used to transform this tree into a physical plan. Since there are many join algorithms and access paths, if we consider all of them at each tree node, the algorithm may become exponential. This can be avoided by using a bottom-up way of constructing the plans (from the leaves to the root of the join tree), where only the best plan is considered each time.

4 Finding a Good Evaluation Order for OODB Operators

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.

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 $< A_i : t_i, v : \mathrm{set}(< A_1 : t_1, \ldots, A_n : t_n >) >$ by grouping together all tuples with the same value A_i.

The min-cut algorithm described in Section 2 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 node that corresponds to a range variable over an extent is indicated by a small circle in the query graph, a node for an unnesting is a shaded circle, and a node for a nesting is a double circle.

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 )
  by  ( h: x.G )
```

corresponds to the query graph in Figure 4.B.

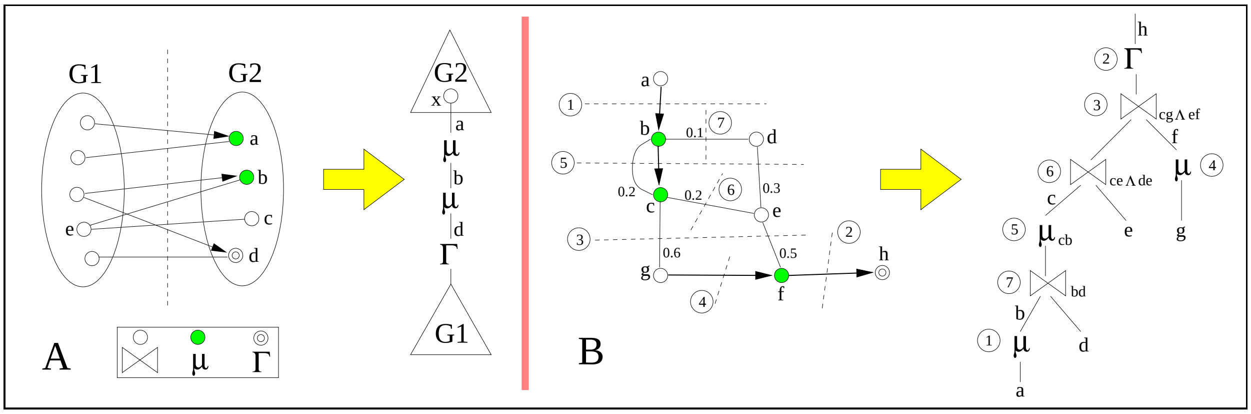

Figure 4: Deriving an OODB Operator Order Using the Min-Cut Algorithm

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. That way, the size of a graph, size(G), is still the product of cardinalities times the product of selectivities.

The min-cut algorithm can be modified to handle the dependencies between nodes. A cut of G into G_1 and G_2 is *valid* if one of the following conditions is true:

1. there are no arrows between the nodes of G_1 and the nodes of G_2;
2. all the arrows between the nodes of G_1 and the nodes of G_2 are of the same direction, e.g. from G_1 nodes to G_2 nodes; in that case, we say that the graph G_2 *depends* on the graph G_1.

All the other cuts are not considered at all. In the first case we generate a regular join, as we did for relational queries. Figure 4.A shows the second case: there are only arrows from G_1 to G_2. In that case, we generate the operator tree for G_1. Then for each node in G_2 that directly depends on a node in G_1 (i.e. when there is an arrow from G_1 to this node) we generate an unnest/nest operation (depending on the type of this node). For the graph G_2 in Figure 4.A, these nodes are a, b, and d, which correspond to unnest, unnest, and nest operations respectively. These operators are inserted on top of the tree generated for G_1. If there are edges between these nodes and the nodes in G_1, the edges become predicates attached to the inserted nest/unnest operators. These operators can be sorted so that the most selective operators (usually the nest operators) are executed before the less selective ones. The three nodes, a, b, and d, are merged into a new node x (of an extent type) in the graph G_2, which now holds the extended tree of G_1 as a value, instead of an extent name. That is, when x is used as a leaf in a later stage of the algorithm, the entire tree for G_1 (with all the inserted operators) will be used in place of x. If there is an edge between G_1 and G_2, such as ec in Figure 4.A, then this edge becomes an edge between c and x in G_2. Since the dependencies cannot form cycles, a query can always be reduced using the above two cases alone.

For example, Figure 4.B gives the operator tree of the example query graph. To simplify the figure, the path $A_i.path$ in $\mu^v_{A_i.path}$ and the attribute A_I in $\Gamma^v_{A_i}$ do not appear in the figure (they correspond to the source of the arrow associated with the operator). The first cut is between the nodes a and b (maximum selectivity). This cut falls into the second case: G_1 here is the node a alone, while G_2 contains the rest of the nodes. Therefore, b is replaced by $\mu(a)$ (circle 1 in the operator tree). The second cut (circle 2) is also of maximum selectivity and separates the nesting h from the rest of the graph. In that case, h is replaced by $\Gamma(\ldots)$, where $\ldots$ is the tree for the rest of the graph. The third cut (circle 3) is of the first type; thus, it produces a regular join. The process continues until all nodes are reduced. Notice that during the fifth cut, the predicate de was promoted to the sixth cut, because it became an edge between c and e.

5 The Algorithm

This section presents the detailed algorithm, called BuildTree, that constructs an OODB operator tree from a query graph. To make the algorithm simpler, we omitted the handling of implied predicates and disjunction. The input of this algorithm is a graph G associated with a set of nodes, nodes(G). A node $n \in$ nodes(G) is associated with a value, a set of edges, edges(n), and a set of arrows, arrows(n). If $(m, w) \in$ edges(n), where $m \in$ nodes(G), then there is an edge between m and n in the graph G of weight w (a real number). This weight is equal to $-\log s$, where s is the selectivity associated with the predicate that relates tables n and m. If $m \in$ arrows(n) then there is an arrow from n to m. (Recall that arrows denote dependencies in the query graph and do not have weights). Each arrow, $m \in$ arrows(n), has a type, *m.type*, that indicates whether the arrow corresponds to an unnest or nest operation. When a query graph is generated from an OODB, the n.value of every node n is the name (or number) that identifies the table associated with n. Before the min-cut algorithm starts, the query graph is lifted to an isomorphic graph with the same nodes and edges but different values; the *m.value* for the new node m, which corresponds to the old node n, becomes the singleton set $\{n\}$. Later the min-cut algorithm will proceed by merging nodes, that is by merging together the values of the new nodes until a graph with one node is produced.

Figure 5 presents the basic components of the algorithm, while Figure 6 presents the actual BuildTree algorithm. The min-cut algorithm is very similar to Prim's minimum spanning tree algorithm [16]. It works in phases: each phase is constructed by the MinCutPhase function.

A subset A of the graph nodes grows starting with an arbitrary single node, n, (line (1) in Figure 5) until A contains all the graph nodes. In each step, the node outside A most tightly connected with A is added to A (node m in line (2)). The $\log n$ factor in the computational complexity of this algorithm can be achieved by using a priority queue of edges to select the edge with the minimum weight each time. At the end of each phase, the two nodes added last (*best* and *last*) are merged, i.e., the value of *last* is extended with the value of *best* (line (3)) and any edges/arrows to *best* are replaced by edges/arrows to *last*. This is achieved with the help of the function substitute(G, *nset*, x), which replaces all nodes in *nset* with x in every edge/arrow in G.

The cut of G that separates *best* from *last* is called the *cut-of-the-phase*. Function MinCut selects the lightest of these cuts. More specifically, line (4) lifts the query graph G_0 into a graph G in which the node values are actually sets of nodes from graph G_0. The lift function is included in Figure 5. It uses a global

```
MinCutPhase ( G )
  begin
    choose a node n ∈ nodes(G);
    A ← {n};                                                              (1)
    while ||A|| < ||G|| do
      begin
        choose m ∈ nodes(G) such that m ∉ A and cost = Σ_{r∈A∧(r,v)∈edges(m)} v is maximum;   (2)
        A ← A ∪ {m};
        last ← best;
        best ← m
      end;
    last.value ← last.value ∪ best.value;                                 (3)
    substitute(G,{last},best);
    return ( best.value, cost )
  end

MinCut ( G_0 )
  begin
    Opened ← ∅; G ← ∅; min ← 0;
    for each m ∈ nodes(G_0) do insert lift(m) to G;                       (4)
    while ||G|| > 2 do
      begin
        (phase,cost) ← MinCutPhase(G);                                    (5)
        left_arrows ← { m | n ∈ phase, m ∈ nodes(G_0) − phase, m ∈ arrows(n) };   (6)
        right_arrows ← { n | n ∈ phase, m ∈ nodes(G_0) − phase, n ∈ arrows(m) };
        if cost < min and (left_arrows= ∅ or right_arrows= ∅) then        (7)
          begin min ← cost; min_phase ← phase;
                min_left ← left_arrows; min_right ← right_arrows end
      end;
    return ( min_phase, min_left, min_right )
  end

lift ( n )
  begin
    if exists (n, m) ∈ Opened then return m;
    S ← A ← ∅;
    for each (m, w) ∈ edges(n) do S ← S ∪ {(lift(m), w)};
    for each m ∈ arrows(n) do A ← A ∪ {lift(m)};
    create a new node new with value {n}, edges S, and arrows A;
    Opened ← Opened ∪{(n, new)};
    return new
  end

substitute ( G, nset, x )
  begin
    for each n ∈ nodes(G) do
      begin
        for each (m, w) ∈ edges(n) do if m ∈ nset then m ← x;
        for each m ∈ arrows(n) do if m ∈ nset then m ← x
      end
  end
```

Figure 5: The Min-Cut Algorithm

BuildTree (G_0)
begin
if nodes(G_0) = $\{n\}$ **then return** n.value; (1)
(min_phase, left_arrows, right_arrows) $\leftarrow$ MinCut(G_0); (2)
$G_1 \leftarrow$ subGraph(G_0, min_phase); (3)
$G_2 \leftarrow$ subGraph(G_0, nodes(G_0) $-$ min_phase);
if left_arrows = right_arrows = $\emptyset$ **then return** BuildTree(G_1) $\bowtie$ BuildTree(G_2) (4)
else begin assuming left_arrows $\neq \emptyset$:
$z \leftarrow$ BuildTree(G_1); (5)
for each $m \in$ left_arrows **do** $z \leftarrow \mu_v(z)$ or $z \leftarrow \Gamma_v(z)$, depending on $m.type$; (6)
create a new node x with x.value=$\{z\}$;
$G_2 \leftarrow$ substitute(G_2,left_arrows,x);
return BuildTree(G_2) (7)
end
end

Figure 6: The Operator Tree Construction

set of visited nodes, called Opened, to avoid lifting the same node twice. After G is populated in line (4), it is passed to MinCutPhase to get one cut-of-the-phase (line (5)). The MinCutPhase function returns a set of nodes, *phase*, that belong to the same side of the cut (say the left side), and the cost of the cut. The set of nodes left_arrows are the nodes in the left side of the cut (i.e. in phase) that have arrows to the nodes in the right side of the cut. The same is done for right_arrows but in the opposite direction. A cut-of-the-phase is ignored if both left_arrows and right_arrows are not empty (i.e. when there are vectors from left to right and from right to left). The cut with the minimum cost is selected in line (7).

Finally, the OODB operator tree is constructed by the function BuildTree in Figure 6. This is a divide-and-conquer algorithm. If the graph has just one node, then the value of this node is returned (line (1)). (Recall that a node value can be an OODB operator tree.) Otherwise, the MinCut function is called to cut the graph G (line (2)). The two cuts, G_1 and G_2, are constructed from the phase nodes (line (3)) (function subGraph(G, s) builds a subgraph of G from a set of nodes s). If there are no arrows between G_1 and G_2, then we generate a join, $\bowtie$, and we call the BuildTree function for G_1 and G_2 (line (4)). The predicate of the join is the conjunction of all the predicates that correspond to the edges in the cut-of-the-phase. Otherwise, we assume with no loss of generality that left_arrows $\neq \emptyset$ and right_arrows $= \emptyset$. Then, according to Figure 4.A, we create a new node x that contains the OODB operator tree for G_1 (line (5)) encapsulated with μ_v and Γ_v operators that correspond to the m.type of every $m \in$ left_arrows (line (6)). These operators can be sorted, as we have discussed in Section 4. Finally, G_2, which now contains a new node x, is translated into a tree by the BuildTree function.

6 Performance Evaluation

In this section we evaluate the performance of the min-cut algorithm related to the optimization of large relational queries (joins). All the tests reported here were executed on an Ultra Sparc 166MHz with 64MB

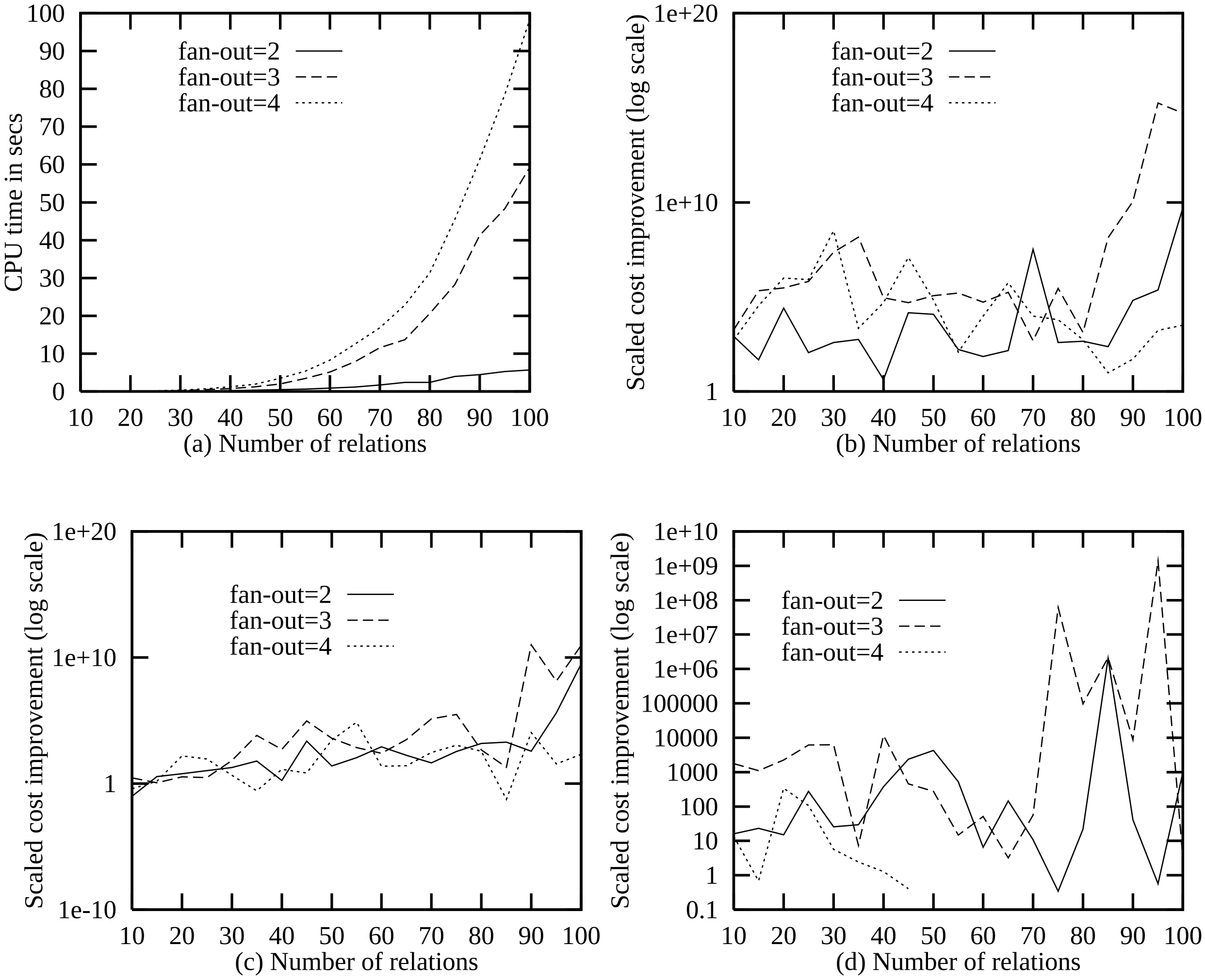

Figure 7: Performance Results

memory running Solaris 2.5.1. The program that produced these results is available at:
`http://www-cse.uta.edu/~fegaras/mincut/`.

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.

Figure 7.a gives the running time performance (in seconds) of the min-cut algorithm for query graphs of $10 \leq n \leq 100$ relations. We tested random graphs with fan-out two (linear queries), three, and four. 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).

We used compared four different algorithms in our performance evaluation:

1. **MC**: the min cut 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. **IMC**: the min-cut algorithm followed by the downhill improvement phase;

4. **ES**: exhaustive search (generate-and-test).

The downhill improvement phase used in II and IMC improves a join 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:

```
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 IMC.

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

n	plans	2	3	4
4	15	3.40	1.94	1.73
5	105	13.30	10.42	11.53
6	945	9.09	26.82	19.46
7	10395	7.75	87.28	51.64

n	2	3	4
4	0	0	0
5	0	0	0
6	0.02	0	0.02
7	0.02	0.01	0.01

The left table compares MC with ES while the right table compares IMC 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 min-cut (MC or IMC) 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 IMC generates plans that are as good as the best plans.

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 three different join algorithms: nested loops where the outer relation is R, 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 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 50% longer than MC (Figure 7.b). 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 (1.5 times the running time of MC). At the end, the best plan found during all iterations is returned. The values in Figure 7.b represent scaled costs: if t_1 is the cost of the plan produced by MC 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 min-cut clearly beats iterative improvement despite the time advantage given to II.

Figure 7.c compares IMC with II, when II is allowed to run 50% longer than IMC. The values in the figure represent scaled costs: if t_1 is the cost of the plan produced by IMC 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. We can see that in that case IMC has about 100% advantage against II. Of course, if we let II run for sufficiently long time, it will eventually find a better plan than MC. Improving MC plans would only result in wasting more cpu time while the plan quality would be comparable with that of improving random plans for the same time period.

Figure 7.d is the result of allowing II to run three times longer than MC. The difference from Figure 7.b is not substantial. The parts of the curve for fan-out=4 that are not shown correspond to negative values between 0 and -0.5. This indicates that when II is bad is really bad (MC plans often have many orders of magnitude lower cost than II plans) and when II is good is not substantially better than MC (II plans are at most 50% better than MC plans). Consequently, the min-cut algorithm is more consistent and stable in optimizing large queries.

7 Related Work

To our knowledge, the heuristic algorithm presented in this paper is the first one that deals with operator ordering in the context of an OODB. 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 [8, 20]. 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 [14]. Most commercial relational optimizers are based on the dynamic programming approach of System R [15] and they cannot handle queries of more than ten tables. A notable exception is Starburst [13], 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 ($\mathcal{O}(2^n)$).

Ioannidis and Kang [7, 6] 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 work by Swami and Gupta [18, 17] compares various heuristics and combinatorial algorithms. Their experiments suggest that iterative improvement is the best combinatorial optimization technique. Morzy, Matysiak, and Salza [11] 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.

8 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: The author is grateful to Yannis Tollis for pointers into the literature on min-cut algorithms and to Ramez Elmasri for helpful comments on the paper. 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, D. Maier, and T. Sheard. Specifying Rule-based Query Optimizers in a Reflective Framework. *Deductive and Object-Oriented Databases, Phoenix, Arizona*, pp 146–168, December 1993. Springer-Verlag, LNCS 461.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] K. Mehlhorn and C. Uhrig. The Minimum Cut Algorithm of Stoer and Wagner. Unpublished manuscript, Max Planck Institute for Computer Science. Available at `http://www.mpi-sb.mpg.de/guide/staff/uhrig/ledapub/reports/mincut.ps.Z`, June 1995.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] M. Stoer and F. Wagner. A Simple Min Cut Algorithm. *Jan van Leeuwen (Ed.): Algorithms - ESA '94, Second Annual European Symposium, Utrecht, The Netherlands*, pp 141–147, September 1994. LNCS 855. Available at `file://fubinf.inf.fu-berlin.de/pub/reports/tr-b-94-12.ps.gz`.

[17] 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.

[18] 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.

[19] A. Swami and B. Schiefer. On the Estimation of Join Result Sizes. In M. Jarke, J. Bubenko, and K. Jeffery, editors, *Advances in Database Technology – EDBT '94*, pp 287–300. Springer-Verlag, 1994. LNCS 779.

[20] 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.