Optimization of Nonrecursive Queries

Ravi Krishnamurthy *Haran Boral* *Carlo Zaniolo*

MCC, 9430 Research Blvd, Austin, TX, 78759

Abstract

State-of-the-art optimization approaches for relational database systems, e.g., those used in systems such as OBE, SQL/DS, and commercial INGRES, when used for queries in non-traditional database applications, suffer from two problems. First, the time complexity of their optimization algorithms, being combinatoric, is exponential in the number of relations to be joined in the query. Their cost is therefore prohibitive in situations such as deductive databases and logic oriented languages for knowledge bases, where hundreds of joins may be required. The second problem with the traditional approaches is that, albeit effective in their specific domain, it is not clear whether they can be generalized to different scenarios (e.g. parallel evaluation) since they lack a formal model to define the assumptions and critical factors on which their validity depends. This paper proposes a solution to these problems by presenting (i) a formal model and a precise statement of the optimization problem that delineates the assumptions and limitations of the previous approaches, and (ii) a quadratic-time algorithm that determines the optimum join order for acyclic queries. The approach proposed is robust; in particular, it is shown that it remains heuristically effective for cyclic queries as well.

1. Introduction:

In traditional database applications, queries requiring more than 10 joins are considered improbable. However, deductive databases [Kellog 81, Gallaire 84], or logic based languages for knowledge applications [Ullman 85, Tsur 85], typically contain hundreds of rules. Translated into relational algebra, these correspond to expressions (similar to database queries) with hundreds (if not thousands) of joins [Zaniolo 85, Kellog 85]; these numbers underscore the need for efficient optimization algorithm.

Optimization of database queries has been extensively investigated. Many heuristics have been proposed and are still being proposed. For instance, pushing selects, preprocessing relations (e.g. sorting), avoiding duplication of work due to common subexpressions, composing a sequence of operations on a single relation into one operation, etc. are a few that are well known. These heuristics, though invaluable in the proper context, rarely take into account the global picture. That is, they do not individually present a framework in which the choices are made; e.g. in what order the heuristics are to be used, under what circumstances are they useful, how are the conflicting heuristics resolved, how to compare them etc. A method, that is both practical as well as comprehensive, was proposed in [Sellinger 79] in which all possible join orders are, in effect, compared to find the optimal execution. The proposed algorithm is functionally equivalent to an exhaustive search, in which, the different alternatives are compared for each element of the search space. In our opinion, an important property of this approach is that the search space is defined independent of the set of heuristics used in the optimization algorithm. As a consequence, this approach provides a systematic framework in which the choices for the above heuristics (and for any more that may be found to be interesting for optimization) can be made in a uniform way.

Although this approach has been proven effective in many existing systems (e.g. SQL/DS, Commercial INGRES, OBE, etc.), there are many questions that are unanswered. The algorithm includes a multitude of parameters (e.g. cost functions, approximations, etc.) whose relevance/effects on the result are very difficult to comprehend. The ramifications of the assumptions are not easily isolatable. Further, the combinatoric behavior of the algorithm is alarming. For a given query on N relations, the worst case time complexity of the algorithm is $O(N!)$ (or $O(2^N)$ if the space requirement is increased exponentially [Sell 79]). This may be acceptable for less than 10 relations, but to optimize a query with 100 relations this approach is prohibitively expensive.

In this paper we present a model and a formal statement of the problem of optimization. In so doing, we delineate the assumptions and ramifications on the results of the optimizer. The model enables us to isolate the various parameters and understand their utility in an optimization algorithm and further indicate the important areas of research to be pursued. We import the polynomial time optimization algorithm for tree queries, proposed by [Ibaraki 84] for a restricted case of query processing, to the more general case discussed in this paper. We also improve this algorithm to present an $O(N^2)$ solution, where N is the number of relations in the query. This, unlike the exponential algorithm presented in [Sellinger 79], is capable of handling queries with large number of relations. Finally we extend this algorithm to include cyclic queries.

Before we discuss the outline of this paper, we would like to emphasize a practical difficulty in designing/implementing query optimizers. The spectrum of execu-

tions that are possible for a given query is vast and the worst execution may be many orders of magnitude worse than a reasonable, (not necessarily the best) execution. Further, the input to such optimizers (e.g. estimates of the selectivities, cardinalities, etc.) are not very accurate for reasons of efficient maintenance of these values. Lastly, even when supplied with accurate input, the problem of optimization is quite difficult (in fact known to be NP-hard), so an approximate algorithm may be the only feasible alternative. Therefore, it is generally accepted by most designers of optimizers, that while the goal of an optimizer is to obtain the "best" execution, it is even more important to avoid the worst cases. Thus, we propound the following maxim: **For an optimizer, it is more important to avoid the worst executions than to obtain the best execution.** The obvious drawback of this maxim is that it defines what not to do but fails to state what an optimizer should do. We shall still seek to obtain the best execution at all times in an optimizer; but whenever we are not sure of optimality, we shall at least ensure the avoidance of the worst cases.

In Section 2 we set up the terminology. The model is described in Section 3. We also define a notion of rooted join tree for a query, which is used to develop the strategy. Intuitively, the root is the first relation to be joined in any resulting execution. In Section 4, we import/improve on the polynomial time algorithm presented in [Ibaraki 84]. First, the algorithm is presented that optimizes a rooted join tree. Then, all possible choices for root are investigated to compute the best execution for tree queries. The specific assumptions needed in this section are clearly listed in the beginning of this section. All of these assumptions are relaxed in Section 5. One important assumption that is used till Section 5, is that the *database is memory resident* – i.e. there is no paging to disk during the execution. Most knowledge bases of today satisfy this property and we do not consider this to be a restrictive assumption. But, as discussed in Section 5, this method is also applicable to the case of disk resident databases. Although the contributions of the model are put forth along with the development of the model, they are not consolidated in any one section. In conclusion, we briefly summarize these contributions and propose some interesting directions for future research.

2. Terminology:

Let $R_1, R_2, \ldots, R_n$ be the *relations* defining the *database DB*, and without loss of generality, let us assume that all of these relations are referenced in a given query Q. A *query* Q = (QL,TL) on the database DB, is a non-procedural request for extracting information from the DB:

$$Q(DB) = \pi_{TL}[\sigma_{QL}(R_1 \times R_2 \times R_3 \times \cdots \times R_n)]$$

where σ, π, and $\times$ are selection, projection and cross product operators respectively; QL is the conjunct of predicates that include both join and selection predicates (i.e. $R_i.a = R_j.b$ and $R_i.a$ = constant), and TL is the list of attributes required in the answer. Certain features of relational languages like aggregation, set difference, etc. are omitted. This is traditionally justified on the basis that the join operation is the most expensive operation [Sellinger 79]. As a matter of fact we go one step further and omit the selection predicates which are not joins (i.e. of the type, $R_i.a$ = constant), and restrict our attention to the query with only joins. In Section 5, we relax this simplification. We make no use of the projection attributes in the optimization. This aspect of the problem, even though quite interesting, has been an open problem in the literature. In summary, we view a query to be a set of join operations (both equality and inequality) on the relations.

Corresponding to a given query, we define a graph representing all the joins implied by the query. This *join graph* is an undirected graph on the set of relations (as nodes) and an edge between R_i and R_j represents a join (e.g. predicate of the type $R_i.a = R_j.b$). A query is called a *tree query* if its join graph is acyclic; the corresponding graph is called *join tree* – to emphasize the tree property.

A *rooted join tree* is a join tree in which a relation is a priori chosen to be the root of the join tree. Although the join tree is undirected, any choice of a root for the join tree can be viewed as the process of converting the tree into a directed tree, the direction being defined from the parent to the child. Consequently, a rooted join tree defines a partial order or in other words, defines the set of total orderings of the joins. In both the partial ordering as well as in each of the total orderings, the root relation precedes all other relations; i.e., root relation must be the first relation to join.

The *selectivity* s_{ij}, with respect to the join of R_i and R_j, is defined to be the expected fraction of tuple pairs from R_i and R_j that will join, quantified as,

$$s_{ij} = \frac{\text{expected no. of tuples in the result of joining } R_i \text{ and } R_j}{\text{number of tuples in } R_i \ * \ \text{number of tuples in } R_j}$$

We make the usual assumptions here regarding the uniformity of the distribution of values and the independence with respect to each other. These assumptions are only approximations to the reality and have been traditionally adopted for convenience [Sellinger 79, Whang 83, Ibaraki 84]. As a consequence of these assumptions, number of tuples satisfying two joins (say R_1 joins with R_2 and R_2 joins with R_3) is $n_1 n_2 n_3 s_{12} s_{23}$; i.e. the product of all the cardinalities and selectivities. This can be extended in the obvious way for k joins. This property will be used in deriving a cost equation for an execution.

For the sake of symmetry, we associate a unique selectivity for each relation in a given rooted join tree. Obviously, a relation may join with many relations and therefore has many selectivities associated with it. *But for a given rooted join tree, each relation R_i has a unique parent and the selectivity of the join with the parent is defined to be s_i.* As for the root, the selectivity is defined to be unity. This definition is only to provide a notational convenience, without any loss of generality. This observation for a rooted join tree also has one more interesting implication. That is, in any total ordering for a rooted join, a non-root relation joins first with its parent.

3. Model:

In any system the query optimizer has to come up with a processing strategy in some form. A strategy, modelled as *'processing tree'*, includes decisions regarding the operations to be performed, the sequence of the operations, and the intermediate relations to be materialized. Since the most expensive operation is join, it is further necessary to prescribe "how" each join must be performed. Thus, we first classify the methods of performing joins into three categories and then give a definition for the processing tree which models an execution. This model also defines the execution space over which the optimization problem is defined. Using this definition, we formulate a precise statement for the optimization problem.

3.1. Join Methods:

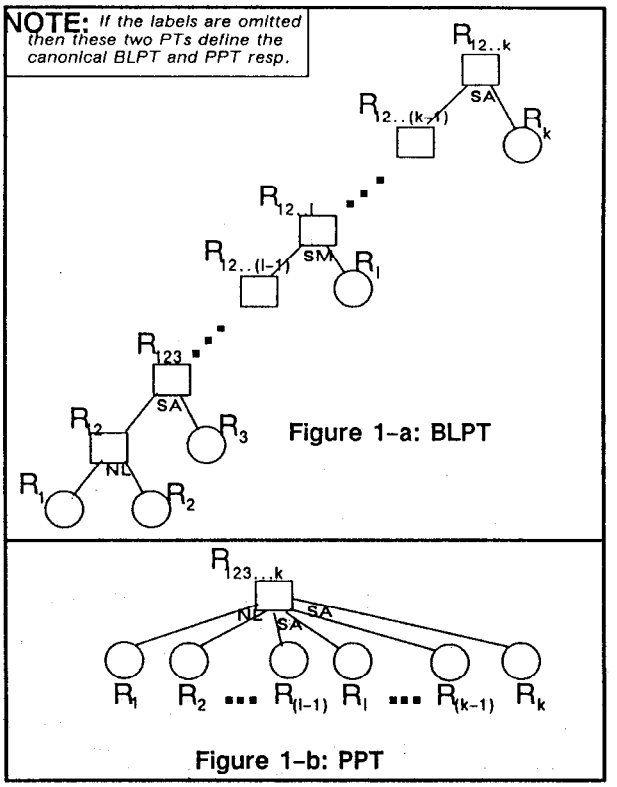
Let n_i be the cardinality of R_i. Binary join methods can be categorized as follows:

1) *Nested Loop (NL):* Relations R_1, and R_2 are scanned in a nested fashion to find tuples in the cross product that satisfy the join predicate. The estimated cost is $n_1 * n_2$, which is invariant even if we commute the two relations. This is true *only* because of the database memory resident assumption.
2) *Selective Access (SA):* Several methods have been proposed including indexed methods, link based methods [Blasgen 76] and hash based methods, having respective cost estimates of $n_1 * \log(n_2)$, n_1, and $n_1 * h$, where h is the average chain length. Note that the cost estimates for these accesses are not invariant over commutation. Thus, one relation is treated as the "outer" relation and the other is the relation that is selectively accessed.
3) *Sort-Merge (SM):* Here, R_1 and R_2 are sorted on join column values, and then the sorted relations are merged to obtain the result. The cost can be estimated by $n_1 + n_2 + n_1 * \log(n_1) + n_2 * \log(n_2)$.

In order to model an execution in which the decisions regarding the join methods are made "independently" of the decisions on other parameters, we make the following assumption. We observe that a generic cost function for most of the above cases is $n_1 * g(n_2)$, where $g(n_2)$ is the differential cost incurred per tuple of the outer relation and is based on the method used. Obviously, the sort-merge technique does not fit this category; neither is the case when the outer relation is selectively accessed (i.e. $g(n_1)$). Initially, we limit our discussion to the class of methods whose cost estimates are $n_1 * g(n_2)$. In Section 5, we relax this assumption to include these other techniques. Note that the accuracy of the proposed cost formulae is debatable. As our intention is to use these formulae in deriving the cost formulae for an execution, the particular formulae used (as long as it is of the above type) is an orthogonal issue and therefore, the accuracy is not discussed here. We will discuss this issue in the context of sort-merge in Section 5.

3.2 Processing Tree:

In most cases, there are many ways to execute a query. An execution of Q is represented by a *processing tree PT(Q)*. PT(Q) is a labelled tree where leaf nodes are relations in Q and each non-leaf node (represented by a square node) is an intermediate relation (i.e. a temporary relation) resulting from the join of all its children; the label specifies the join method. For example, Figure 1a shows a processing tree representing an execution of



a query on m relations. R_1 and R_2 are joined to obtain R_{12}; .....; $R_{12\cdots i}$ and $R_{12\cdots(i+1)}$ are joined to obtain $R_{12\cdots(i+2)}$; ...;and $R_{12\cdots(k-1)}$ and R_k are joined to obtain the answer to the query. The particular join method used for each join operation is a label for the resulting non-leaf node in PT(Q); e.g. R_{12} in Figure 1a is the result of joining R_1 and R_2 using nested loop method.

Processing trees can be classified according to the nature of the tree. For example, the tree in Figure 1-a has a linear structure - so it is termed a *Linear Processing Tree (LPT)*. It has the special property that no more than one temporary relation is used as input to any join operation. A processing tree will be a binary tree *(BPT)* if all the joins performed are binary in nature, leading to an obvious definition of a *Binary LPT (BLPT)*. Another type of LPT, called *Pipelined Processing Tree, PPT,* is an LPT whose height is unity. An example of a canonical PPT is shown in Figure 1-b. That is, the answer is generated directly from the relations without creating any temporary relations. Traditionally, this is known as the nested loop n-ary join method. Note that the pipelining (and therefore, the joining) takes place in the order from left to right - which defines the order of nesting and a join method is specified for each level of nesting. A particular variation of this method was assumed in [Ibaraki 84]. It is possible to classify various other types of processing trees. For our subsequent discussion, LPTs (and in particular, binary LPTs and PPTs) are sufficient.

A class of PTs defines an *execution space* over which the optimization is defined. For example, LPT execution space is the set of all executions whose processing trees are LPTs, which is the search space assumed by many optimizers [Sellinger 79, Whang 85]. We use the terms

PTs, LPTs, BLPTs, PPTs to refer to the execution space as well as the class of processing trees.

A processing tree also represents the partial order of the join operations; i.e., the lower level joins must be performed before the joins above them. In the case of LPTs this ordering is a total order, called the *LPT sequence*. Consequently, a given total order has a unique binary LPT (or PPT). Further, an LPT is said to *correspond* to a rooted join tree in which the root is the first element of the LPT sequence.

Even though all processing trees will produce the same answer, some processing trees can be discarded a priori (e.g. those trees that require the result of a cross product). We formalize this notion here. For a given LPT, the LPT sequence is a traversal of the corresponding rooted join tree (i.e. a directed tree) if and only if the following two properties are satisfied: (1) the LPT sequence is consistent with the partial order defined by the corresponding rooted join tree; (2) that a join operation (that is not a cross product) is performed at each of the non-leaf node of the LPT. Thus we define an LPT to be *consistent* if the LPT sequence is consistent with the partial order implied by the rooted join tree that corresponds to the LPT. For example, executions allowed by SQL/DS, OBE are limited to consistent LPTs.

3.3. Cost Equation:

Given a processing tree, based on the cost of the individual joins, we can estimate the cost of the corresponding execution - this is defined to be the cost of the processing tree. In this subsection, we compute the cost of a given PPT and a given BLPT. In Section 5, we handle the case of any general LPT. Note that any reference to selectivity s_i, for the relation R_i, is the selectivity of the join of R_i with its parent, based on the rooted join tree corresponding to the LPT.

In order to compute the cost of joining a relation with the result of the join of all the previous relations, - either in the form of a materialized temporary relation, or if done in a pipelined fashion - we need to compute the cardinality of the result of a series of joins. It is easy to see that the cardinality of the result of joining R_1 and R_2 (i.e. cardinality of R_{12} in Figure 1-a) is

$$n_{12} = s_2 * (n_2 * n_1) = (s_2 * s_1) * (n_2 * n_1) \quad (1)$$

Note that the "dummy" selectivity s_1 (i.e. the selectivity associated with the root of the join tree) was defined to be unity. Using n_{12} we can compute n_{123} as follows:

$$n_{123} = s_3 * n_{12} * n_3 = (s_3 * s_2 * s_1) * (n_3 * n_2 * n_1) \quad (2)$$

In general,
$$n_{123\ldots j} = \prod_{i=1}^{j} (s_i * n_i) \quad (3)$$

Note that $n_{123\ldots j}$ is the size of $R_{123\ldots j}$ in a BLPT whereas $n_{123\ldots j}$ is the number of times the pipelined strategy goes past the j-th relation in a PPT. The above expression is valid for both PPTs and BLPTs, whose canonical executions are given in Figure 1, where the labels are omitted.

The cost for a PPT(Q) is computed as the sum of the individual join costs. Intuitively, the cost is measured in terms of the number of comparisons made.

$$\text{Cost of PPT(Q)} = \sum_{j=2}^{k} \left[(n_{123\ldots(j-1)}) * g_j(n_j) \right]$$
$$= \sum_{j=2}^{k} \left[\left(\prod_{i=1}^{j-1} [s_i * n_i] \right) * g_j(n_j) \right] \quad (4)$$

where $g_j(n_j)$ is dependent on the join method used.

The cost of a given BLPT differs from that of a PPT *only* because the BLPT execution has the extra overhead of storing temporary relations. In most systems, insert operations would be considerably more expensive than a simple retrieve. So the equation has an extra component that estimates the cost of materialization. For each temporary relation created, (say $R_{12\ldots j}$), a cost of $c * n_{12\ldots j}$, is incurred where c is the constant that relates the cost of comparison and cost of insert (assuming the cost of comparison is unity). Thus the cost of BLPT(Q) is

$$\text{Cost of BLPT(Q)}$$
$$= \sum_{j=2}^{k} \left[(n_{123\ldots(j-1)}) * g_j(n_j) \right] + \sum_{j=2}^{k} (c * n_{123\ldots j}) \quad (5)$$
$$= \sum_{j=2}^{k} \left[(n_{12\ldots(j-1)}) * g_j(n_j) \right] + \sum_{j=2}^{k} \left[(n_{12\ldots(j-1)}) * c * n_j * s_j \right]$$
$$= \sum_{j=2}^{k} \left[\left(\prod_{i=1}^{j-1} [s_i * n_i] \right) * (g_j(n_j) + c * n_j * s_j) \right] \quad (6)$$

Note that the equations (6) and (4) are identical if we redefine the 'g' function. This leads us to an interesting observation that the cost of materializing the temporary (which is a function of the size of the temporary relation) does not change the structure of the cost equation. Also note that the above cost does include the cost of constructing the answer, whereas equation (4) does not. As this is a fixed cost per query, neither omitting this cost nor including it affect the result of the minimization algorithm.

3.4. Optimization problem:

We define a query optimization strategy as an algorithm used to choose an "optimal" processing tree (including the necessary join methods) for a given query. This can be formally stated as follows: **Given a query Q, and an execution space E, find an execution in E that is of minimum cost.** As the execution space can be abstractly viewed as a set of processing trees, the above problem can be restated to search for the minimum cost processing tree. As mentioned before, not all possible executions are allowed by a given system. For example the execution space allowed by QBE, System-R and OBE are all a subset of LPTs. All these systems choose executions that have no more than one temporary relation used in any join operation, at any time during the execution. Further, the execution space of the optimization presented in [Ibaraki 84] is a subset of PPTs and PPTs $\subset$ LPTs. Thus we restrict our problem as follows:

LPT Query Optimization Problem:
Given a query Q, find a processing tree pt in LPT that is of minimum cost; i.e.

$$\underset{pt \in LPT}{MIN} \; [\text{cost of } pt(Q)]$$

Note that if Q is a tree query then the cost of pt(Q) is estimated by equation (4) or (6) as long as the class is limited to either BLPT or PPT. As every pt in BLPT and PPT uniquely correspond to an LPT sequence and vice versa, this problem in effect defines the search space to be the n! sequences, out of which the optimal sequence is chosen. In [Sellinger 79] they suggest searching this combinatoric space, with the restriction that only consistent sequences are checked. Obviously, checking consistent sequences does not change the worst case size of the search space. Although this approach is effective in the database domain, unfortunately, in knowledge base systems this is not feasible. In the next Section we discuss the problem of LPT query optimization where we import the polynomial time solution for tree queries from [Ibaraki 84] to this general model of optimization. Further, we also improve on their solution.

4. Strategy for Tree Queries:

In this subsection we first develop a strategy for a rooted tree query and then find the optimal execution over all possible choices for the root. Before we present the strategy, we first reiterate the list of assumptions, we restate the cost equation of the previous section and observe some properties.

4.1. Assumptions:

We restate the list of significant assumptions here, all of which will be relaxed in the next section.

1) The query is restricted to contain only join predicates; i.e. no selection predicates.

2) The general cost formula (i.e. $n_1 * g_2(n_2)$) is applicable for all the join methods.

3) The execution space is restricted to be either PPT or BLPT in the proposed optimization algorithm.

4) Database is assumed to be memory resident. Therefore, the cost functions are based on the processing costs (e.g. comparisons, insert operations) instead of number of disk accesses.

5) Query is assumed to be a tree query

The above list is only a partial list of assumptions – i.e. those that we consider significant. There are other assumptions in the definition of the query, definition of the selectivity, etc. that are made by most, if not all, previous solutions to this problem [Sellinger 79, Ibaraki 84, Whang 85].

4.2. Cost Model:

The cost of processing a query for a given sequence (i.e. a BLPT or PPT), which was formulated in equations (4) and (6) of the previous section, can be recursively defined as follows:

$C(\Lambda) = 0$ for the null sequence Λ.
$C(R_j) = g_j(n_j)$ R_j, non-root relation
$C(R_j) = 0$ R_j is the root of join tree
$C(S_1S_2) = C(S_1) + T(S_1) * C(S_2)$ any subsequences S_1 and S_2.

where $T(*)$ is given by
$T(\Lambda) = 1$ for the null sequence Λ.
$T(S) = \prod_{R_k \in S} (s_k * n_k)$ for any sequence S.

It is easy to show that the above recursive definition correctly computes the cost of a consistent sequence as per equations (4) and (6). Note that the 'g' functions depend on the join method specified in the chosen PPT or BLPT. For any consistent LPT, corresponding to a given rooted join tree, the join of any relation is with its parent. Therefore, the best join method for each relation can be determined independent of the choice for LPT. Note that the independence is a direct consequence of the two assumptions: 1) tree query; 2) differential cost formulae for joins. Interestingly, Ibaraki and Kameda [Ibaraki 84] started with a model that computes the number of page fetches (i.e. the database is disk resident) and derived an identical cost equation where $g_j(n_j)$ is computed only for the restricted form of the nested loop. More discussion on this topic is relegated to the next section when we attempt to relax the assumptions. Consequence of this similarity is that we can import their polynomial time solution to the above general statement of the problem and also present an improved version of their solution.

An interesting property of the above equation (as observed by [Ibaraki 84]) is that it satisfies the adjacent sequence interchange property (ASI property for short, [Monma 79]). Even though it is straightforward, for the sake of completeness, we reiterate the following lemma and proof from [Ibaraki 84]. This lemma identifies the ASI property.

Lemma 1: Given arbitrary sequences A, B and nonnull sequences U and V, such that AUVB and AVUB are consistent with the given rooted join tree, then $C(AUVB) \leq C(AVUB)$ if and only if $rank(U) \leq rank(V)$, where the rank is defined for any nonnull sequence as $rank(S) = (T(S)-1)/C(S)$.

Proof: Using the recursive definition of the cost function, we have

$$C(AUVB) = C(A)+T(A)C(U)+ T(A)T(U)C(V)+T(A)T(U)T(V)C(B).$$

Thus, we can derive,

$$C(AUVB) - C(AVUB)$$
$$= T(A)\left[C(V)[T(U) - 1] - C(U)[T(V) - 1]\right]$$
$$= T(A)C(U)C(V)\left[rank(U) - rank(V)\right]$$

The lemma follows directly from the above equation. ■

A corrollary of the above lemma can be stated as follows: *purely based on the properties of the subsequences U and V, their ordering can be decided irrespective of the rest of the sequence (i.e. A and B).* Thus a cost function is said to satisfy the ASI property if there exists a rank function as defined in the above lemma [Monma 79].

Let us consider an example of Figure 2 in which the relation R_1 joins with R_2 and R_6 and the root is assumed to be R_1. The question of finding the total order is same

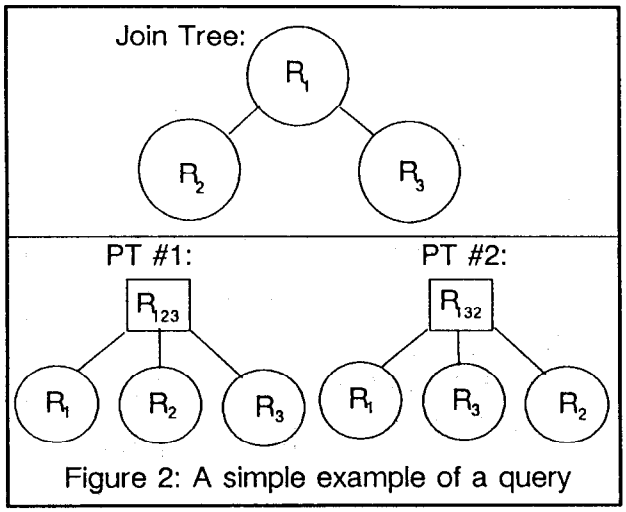


Figure 2: A simple example of a query

as determining which of the two relations, R_2 or R_3, is to be joined first. The two PPT cases $R_1 R_2 R_3$ and $R_1 R_3 R_2$ are shown in PT#1 and PT#2 respectively. The rank function dictates that if rank(R_2) < rank(R_3) then R_2 should join before R_3. To understand the rank function intuitively, let us analyze the case when PT#1 is better than PT#2. Obviously, from equation (4), we can argue that PT#1 is better if the following holds.

$$n_1 * g_2(n_2) + n_{12} * g_3(n_3) < n_1 * g_3(n_3) + n_{13} * g_2(n_2)$$

or,
$$\frac{n_{12} - n_1}{g_2(n_2)} < \frac{n_{13} - n_1}{g_3(n_3)}$$

Intuitively, the increase in the 'intermediate result', normalized by the cost of doing the join is to be minimized for optimality. Replacing n_{12} and n_{13} using equation (3) of the previous section, we get,

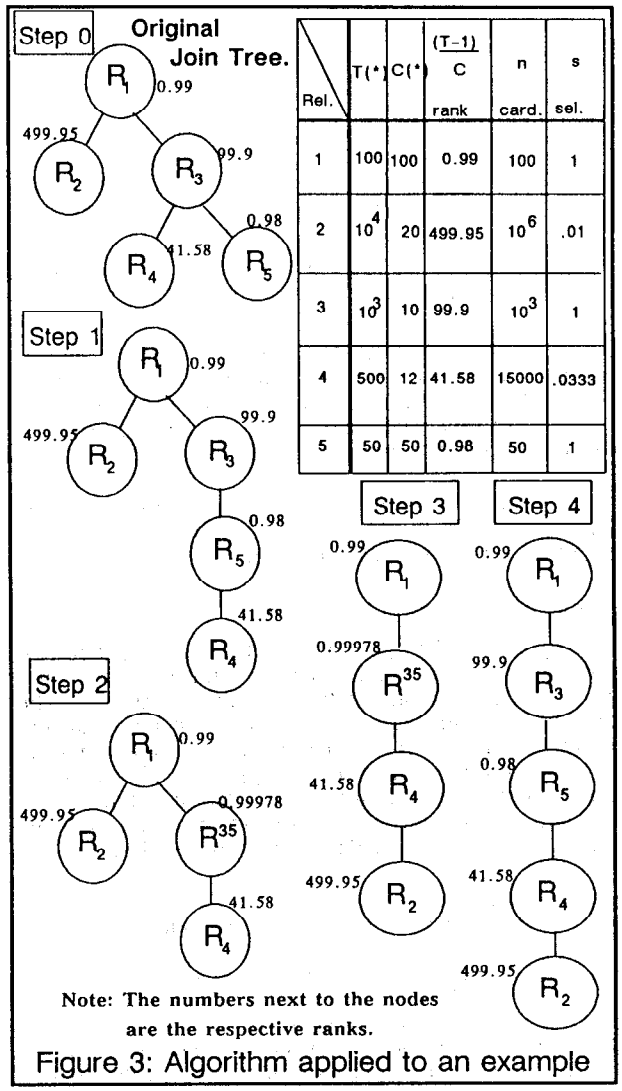
$$\frac{n_2 * s_2 - 1}{g_2(n_2)} < \frac{n_3 * s_3 - 1}{g_3(n_3)}, \quad or\ Rank(R_2) < Rank(R_3)$$

Intuitively, **the rank measures the increase in the intermediate result per unit differential cost of doing the join.**

It is known that an optimal sequence based on such a cost function can be obtained in O(N log N) time for the case of series-parallel order constraints, where N is the number of elements to be sequenced [Lawler 78, Monma 79, Abdel-Waheb 80]. Having observed that a rooted join tree is a special case of the series-parallel constraints, Ibaraki and Kameda [Ibaraki 84] imported the solution to find the optimal order of the joins for a rooted join tree in O(N log N) time. As there are N choices for the root, they conclude that the time taken to compute the optimal sequence for the query is $O(N^2 \log N)$. We use the same algorithm to compute the order for a rooted tree. Subsequently, we present an $O(N^2)$ algorithm that finds the optimal sequence for the query.

4.3. An Example:

Let us first show the use of the ASI property on an example given in Figure 3. The query consists of 5 relations that are joined as given by the join tree. This join tree is assumed to be rooted at R_1. In the adjoining table the values for functions T and C are given for each individual relation. Also shown for each relation is the computed value for the rank function. The following algorithm uses a bottom-up approach. Succinctly, the algorithm uses the ranks of the individual nodes to order the lowest level subtrees, converting each subtree into a *chain* (i.e. a total order). Then the subtrees of the next higher level are converted to chains by merging the sibling chains based on the ranks..... and so on... until the tree under the root is converted into a chain.



Rel.	T(*)	C(*)	(T-1)/C rank	n card.	s sel.
1	100	100	0.99	100	1
2	10^4	20	499.95	10^6	.01
3	10^3	10	99.9	10^3	1
4	500	12	41.58	15000	.0333
5	50	50	0.98	50	1

Figure 3: Algorithm applied to an example

First, the subtree rooted at R_3 is converted into a chain using the rank function. As rank(R_5) < rank(R_4), the total order for this subtree is $R_3 R_5 R_4$, which is shown as step 1. The intuition behind this transformation is that the relations R_4 and R_5 are not constrained to be ordered in any way. Therefore, we can order them based on their ranks. This ordering, by the lemma of the previous section, minimizes the cost. But R_3 has to precede R_4 and R_5 due to the partial ordering defined by the rooted join tree.

Next the two subtrees of R_1 are combined. Note that these two subtrees are chains and therefore we can combine the two chains in any order that preserves the individual order of each chain. We do this by merging the two chains based on the ranks. In order to merge them we must ensure that each chain is ordered by rank to begin with. Note that the chain under R_3 is not ordered

by rank, but R_3 has to be the first element in the chain due to the constraints in the rooted join tree. Note that *only* the root of the chain may violate the order.

To transform this chain into a chain where the nodes are ordered by rank, we apply a normalization step in which we combine the nodes R_3 and R_5 into one node and the rank is computed for this sequence of nodes as follows:

$$\text{rank}(R^{35}) = \frac{T(R^{35}) - 1}{C(R^{35})} = \frac{T(R_3)\,T(R_5) - 1}{C(R_3) + T(R_3)C(R_5)} = 0.99978$$

We can combine these nodes because it can be shown, in general, that no other nodes can be placed between R_3 and R_5 if the cost is to be minimized. The result is shown in step 2. This normalization step is repeated until the chain containing the merged nodes is ordered by rank.

In step 3 the two chains are merged and in step 4 the final total order for the query is shown where the merged nodes have been expanded.

4.4. Algorithm for rooted tree queries:

Restating the BLPT (or PPT) optimization probolem: given a rooted join tree for a query, the goal is to find the total order of relations that minimizes the cost. The sequence is constrained by the fact that it be consistent with the partial order defined by the rooted join tree. The algorithm uses a bottom-up approach of transforming each subtree (all of whose children are chains) into one chain. Thus the result of transforming the join tree produces one chain that determines the total order.

Algorithm OPT:
Input: rooted tree query Q, including the values for the functions T and C.
Output: Total Order for the rooted tree query.

1. If the tree is a single chain then stop.
2. Find a subtree (say rooted at *r)* all of whose children are chains.
3. Merge the chains based on the ranks such that the resulting single chain is nondecreasing on the rank.
4. Normalize the root *r* of the subtree (i.e. a single chain now) as follows:
   a. While the rank of the root is greater than its immediate child c do;
      Replace root and c by a new node representing the subchain r followed by c
5. Go to 1.

Intuitively, the algorithm works bottom up, whereby, the lowest subtree is converted into a chain; then all the sibling chains are converted to form a chain under their parent, ... etc. Thus, the algorithm terminates after creating the chain under the root. The total order for the query is obtained by decomposing all the composite nodes created by the normalization step above.

For a formal proof of correctness the reader is referred to [Monma 79] wherein this is proved in a more general context. We present an intuitive argument here. If the normalization step is never executed then it is obvious from the lemma of subsection 4.2 that the correct total order is obtained. In order to show that the creation of the composite nodes does not prohibit any interesting order we can show the following: if the rank(c) < rank(r)

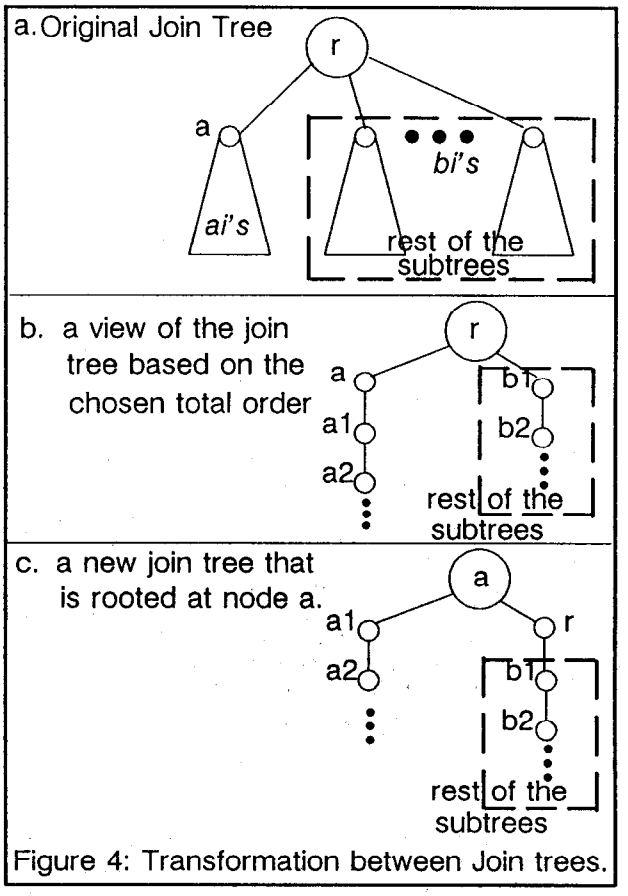


Figure 4: Transformation between Join trees.

then any node from the subtrees rooted at the siblings of r has to be placed either above r or below c.

This algorithm can be implemented to run in O(N log N) time [Lawler 78]. This may not be an interesting bound for our case because we will spend $O(N^2)$ time to find the optimal rooted tree for which the cost is minimized.

4.5. Algorithm for Tree queries:

As mentioned before, one approach to find the rooted join tree with the optimal cost (termed the optimal rooted join tree) is to compute the cost for each choice of the root and choose the one with the minimum cost. We present in this section, a more efficient method to find the optimal rooted join tree. This approach uses the fact that the computation corresponding to two choices for the root have a lot in common - especially if the roots are adjacent in the join tree.

In Figure 4-a, a join tree rooted at r (say T_1) is shown with its subtrees, one of which is rooted at, say, a. Let us compute the optimal order for the join tree rooted at a, given the optimal total order for T_1.

We first transform T_1 into another tree (as shown in Figure 4-b) based on the optimal total order for T_1. All the nodes a1, a2, ..., etc. (b1, b2, ..., etc.) are nodes from the subtree rooted at a (from the rest of the subtrees). Without loss of generality, let the order of *ai's* and *bi's* be the order in the respective subsequences of the optimal order for T_1. It is straight forward to show that the

transformation to the tree shown in Figure 4-b can be done in O(N) time, given the optimal order for T_1 .

To compute the optimal order for the join tree rooted at a, we have to do the following: (1) Compute the rank of r using a as its parent; (2) Merge the two chains (shown in Figure 4-c) under a based on their ranks. Obviously, these two steps can also be done in O(N) time. Therefore, given the optimal order for a join tree rooted at *r*, the optimal order for a join tree that is rooted at an adjacent node to r can be computed in O(N) time. As there are (N-1) such transformations to be done to exhaust all possible choices for the root, the total time taken is $O(N^2)$.

The correctness of this algorithm can be shown from the following observation. If the parent of a node is same in two join trees, then the respective ranks remain unchanged and therefore their respective ordering also remains unchanged. In the proposed transformation, all the *ai's*, and the *bi's* have the same parent. Only a and r have different parents. As a is the root of the new join tree, it must be the first in the optimal order irrespective of its rank. As for r, we compute the new rank. Based on these observations, it is straightforward to prove the correctness of the above algorithm.

5. Relaxing the Assumptions:

In this section we relax the major assumptions we made in presenting the strategy of the previous section. First, we emphasize a practical difficulty regarding the estimation of selectivities. Being an estimation, the accuracy is an issue. Accurate estimation is quite expensive; therefore, in most systems, accuracy is sacrificed for the sake of efficient implementation. Lack of accuracy in the estimation of selectivities is assumed in this paper. The consequence of this assumption is that the accuracy of computations using the selectivities need only be comparably accurate.

5.1 Pushing the select:

Pre-selecting a relation is known to be advantageous in many cases. The main drawback of pre-selecting is that the existing indices on the original relation cannot be used on the selected version of the relation. If pre-selection is chosen, then either an index has to be created on the selected version or use the selected version without any index. Ideally, an optimizer should weigh these alternatives based on the expected savings. It is easy to see that the choice of pre-selecting or not can be made in conjunction with the choice of the join method for a given rooted join tree.

5.2. Accommodating other Join Methods:

We omitted two join methods from our discussion so far. These are: i) sort-merge technique, and ii) the join method in which an index is created on the temporary relation and then the other relation is joined by looking up the created index. Both these resulted in a cost formula that did not correspond to the general formula, namely $n_1 * g(n_2)$. Here we argue that we can find an approximation to the cost of these join methods and use the proposed strategy using these approximate cost estimates. Note that if all the relations are in memory, which is at least the case in most knowledge base systems of today, then sort-merge is never an useful join method [see Krishnamurthy]. But if database is disk resident then sort-merge is an useful technique.

The main reason for any method to fit the general cost formula is that the "differential " cost per tuple for the join of any relation must be computed only on the information from that relation. Mathematically speaking, the partial differentiation of the join cost with respect to the n_1 (i.e. number of tuples in the outer relation) is independent of n_1. Obviously, if the cost has a nonlinear term on n_1 (e.g. $n_1 \log (n_1)$) then the differential cost will not be independent of n_1. We argue here that if we can estimate that cost to a reasonable accuracy, then we can use that approximation to be the value for g(n) and expect to get a reasonably correct result from the optimization algorithm. This argument underscores two fundamental maxim/assumption. First, the optimizer should avoid the worst cases and attempt to get a reasonably good execution. Second, the selectivities are themselves estimations with considerable inaccuracies. In short, we feel that a simple minded estimations should prove to be sufficient to accommodate these join methods.

5.3. LPT Query Optimization:

Until now we have restricted our attention to either BLPT or PPT query optimization problem. Interestingly, we can view BLPT and PPT as the two ends of the spectrum of processing trees in LPT. The trees in between have some temporary relations materialized and others are computing the joins using the pipelined execution approach. So the important question to be answered is: under what circumstances, does a temporary relation have to be materialized? Obviously, if all join methods are of the nested loop or selective access then the pipelined strategy is always better - this is a direct consequence of the cost equations in (4) and (6). As a matter of fact, when all relations are in memory, then PPT is the best choice in all but few cases. This is an obvious consequence of the uselessness of the sort-merge technique. On the other hand if sort-merge technique is used, then the temporary relation has to be sorted, for which it must be materialized. Thus, *we need to materialize the temporary relation, if and only if the subsequent join operation requires it.* Below we extend the strategy to find an optimal LPT for a given query.

If we use the approximations of the previous subsections for the join methods such as sort-merge, then we can find the total order of the execution assuming the processing tree to be a PPT. Note the cost of writing the temporaries should be included into the cost formula for these join methods. We observed in the formulation of the cost equation for BLPT that such an inclusion is feasible. On obtaining the total order, any relation whose join method requires a materialized temporary relation is identified and the processing tree is modified to reflect the change from the pipelined mode to LPT mode. Thus an optimal LPT can be obtained.

5.4. Disk Resident Databases:

One of the important claims we make in this paper is that the problem of optimization is not complicated by the fact that the database is disk resident. The structure of the equation in Section 4.2 is solely dependent on the fact that the optimization is limited to the LPT execution class. The fact that the data is disk resident is reflected

in formulating the 'g' functions. This is also confirmed by the fact that in [Ibaraki 84] they start with a disk based model (i.e. computing the number of I/Os) and the cost equation developed by them is structurally identical. Thus, we claim that extending this proposal to a disk based model is a straightforward task.

The change to disk based model may have some ramifications indirectly. For instance, sort-merge technique may become more important and a good approximation may be desired. Further, the cost estimates and discussions in the previous subsections will have to be argued on the basis of block accesses and not tuples. Nevertheless, it is our contention that the **structure** of the cost equation will remain intact and the strategy will be applicable.

5.5. Optimizing cyclic queries:

Extending the solution to allow the queries to be cyclic is difficult because this problem can be shown to be NP-hard. In [Ibaraki 84] they have shown that this problem for the restricted case of optimizing in the PPT execution space is NP-hard. The reduction from this problem to the more general problem is straightforward. Here we present a heuristic to handle cyclic queries in the context of the LPT query optimization problem. As the proposal is based on heuristics, it is not necessarily an optimal solution, but we argue that the resulting order is likely to be reasonably good.

The basis of this heuristic depends on the following two observations:

Observation No. 1: The answer to a query, corresponding to any subgraph of the join graph, is a superset of the answer to the original query.

Intuitively, by disregarding a join predicate we get more tuples in the answer. Therefore, if we compute the query corresponding to any spanning tree for the graph and then check each tuple in the answer for the satisfiability of the join predicates (i.e. edges) not in the spanning tree, the resulting set will be the answer to the original query. In fact, we can improve on this approach by observing that we need the tree property only for computing the 'optimal' order. The subsequent computation of the query may use all the join predicates and compute the final answer directly. The relevant question to be answered is - how good is the chosen order that is computed based on the spanning tree for the original cyclic query. The following observation gives a clue that leads us to the proposed heuristic.

Observation No. 2: A join that has a good selectivity is more likely to be influential in choosing the order than a join that has poor selectivity.

Let us take the limiting case of a join that has the worst selectivity (i.e. unity), which is the case of a cross product. Note that in a join graph, the lack of an edge between two relations is by default a cross product. As we mentioned in the motivation of the definition of 'consistency', any join is implicitly favored over a cross product. This is because, the cross product, in most cases, is better if done as late as possible. Therefore, a cross product is of no use in dictating the order of the joins - as mentioned before, this has been assumed in the approach taken by all the optimizer of known systems as well as the previous research [Sellinger 79, Ibaraki 84, Whang 85]. The above observation is a straightforward extension of this argument.

Combining the two observations, we conclude that a spanning tree containing the joins with good selectivities is possibly a good choice. Thus the following strategy:

Choose the minimum cost spanning tree from the join graph, where the selectivities are the weights for the edges. Then use this spanning tree to compute the total order, which is used to compute the original query.

Here, the total cost of the spanning tree is defined as the product of all the selectivities and not the summation as it is commonly stated for the minimum cost spanning tree problem. Intuitively, the proposed heuristic finds a spanning tree such that the cardinality of the answer to the query (corresponding to the spanning tree) is minimized. Thus, the reduction from this intermediate answer to the final answer is also minimized. Consequently, the order chosen for the spanning tree, even if it is not optimal, is likely to assure the maxim outlined in the introduction.

6. Conclusion:

We have presented a viable strategy for optimizing knowledge base queries. In so doing, we have proposed a model for the general problem of optimization of queries that captures many approaches (e.g. BLPT, PPT optimizations). In this model we have imported and improved the previously proposed polynomial time algorithm for ordering the joins. We have extended this to include heuristics such as pushing selects, preprocessing of relations, allowing other join methods, etc. By these examples, we have demonstrated the capability and the flexibility of incorporating various heuristics into the optimization strategy without changing the structure of the algorithm. In summary, we have presented an algorithm for optimization of queries with large number of joins that is adaptable in the new scenario.

We consider the model to be a by-product of this research. By presenting a formal model for the well-known optimization problem, the strengths and weaknesses of the traditional approach are made apparent. We briefly summarize them here. One of the advantage of the traditional approach, as evident from the model, is the separation of the abstract search space and the set of heuristics. This provides the flexibility to extend the algorithm to incorporate new heuristics. On the other side, although the traditional algorithm is an exhaustive search, it is still limited to the LPT execution space. Therefore, the approach is not necessarily optimal, especially if intra-query parallelism is to be exploited. Further, pruning effect due to consistent executions or branch-and-bound techniques are not necessarily successful in many cases (e.g. a relation joining with 99 other relations in a star-like join graph).

An interesting observation that can be made from the proposed algorithm is the importance of the notion of differential cost of a join method. By estimating the join cost independent of the join order, the optimization algorithm is drastically simplified. This is, in our opinion,

the major deviation from the traditional approach that provides the handle for efficient optimization.

Based on the model, it is clear that if sorting of temporary results is not required by the processing tree, then PPT is the optimal strategy. An observation confirmed by the results from implemented systems (e.g. SQL/DS, OBE). This leads us to the important conclusion that pipelined strategy is optimal when database is memory resident, because the sort-merge technique is useless. As a consequence of this observation, we make an important observation in the arena of expert systems. A commonly pondered question: is the tuple-at-a-time modus operandi a source of poor performance for expert systems? Stated otherwise, can the performance be improved by taking the approach of a set-at-a-time processing? This model answers both these questions in the negative as long as the database is in memory, which is true in most knowledge base systems. As a matter of fact, based on the above cost model, the conclusion is that PPT executions may be better to the extent that they do not materialize the temporaries. On the other hand if database is in disk there is some advantage to materializing temporaries.

Finally, this model clearly puts forth areas of research that has not been investigated. First, can we optimize over a larger class of executions, namely the entire PT. This will allow us to create more than one temporary result and thereby allow more parallelism if there are resources (e.g. DB machine) to support it. Second, is it possible to develop cost functions for other heuristics, such as, union/intersection operations, duplicate elimination, aggregation etc., such that the recursive cost structure is retained. Yet another more difficult problem is to extend this approach to optimization of recursive queries. Last, can this method be validated? These are some of the areas of an on going research by the authors.

Acknowledgments: We are grateful to Francois Bancilhon, George Copeland, Setrag Khoshafian, Won Kim, and Patrick Valduriez, who provided a forum for discussion at various stages of development of this paper. We would like to thank Guy Lohman and Kyu-Young Whang for their help in disproving some earlier conjectures and to David Maier for focusing our attention to the quadratic-time solution.

References:

[Abdel-W. 80] Abdel-Wahab, H.M., and Kameda, T. On strictly optimal schedules for the cumulative cost-optimal scheduling problem. *Computing 24* (1980), 61-86

[Blasgen 76] Blasgen, M.W., and Eswaran, K.P. Storage and Access in relational databases, *IBM System J. 16, 4* (1977), 363-377

[Gallaire 84] Gallaire, H., Minker, J. and Nicholas, J. Logic and Databases; A deductive approach, Vol. 16, No. 2, June 1984, 153-185

[Ibaraki 84] Ibaraki, T., and Kameda, T. Optimal nesting for Computing N-relational Joins, *TODS 9, 3* (1984), 482-502

[Lawler 78] Lawler, E.L., Sequencing jobs to minimize total weighted completion time subject to precedence constraints, *Ann. Discrete Math. 2* (1978) 75-90

[Kellog 81] Kellog, C., and Travis, L. Reasoning with data in a deductively augmented database system, in *Advances in Database Theory: Vol 1*, H.Gallaire, J. Minker, and J. Nicholas eds., Plenum Press, New York, 1981, pp 261-298.

[Kellog 85] Kellog, C., O'Hare, T., and Travis, L. Optimizing the rule-data interface in a KMS, Submitted for publication.

[Krishnamur.] Krishnamurthy, R., Navathe, S., and Morgan, S.P. A Pragmatic approach to Query Processing , in preparation.

[Monma 79] Monma, C.L., and Sidney, J.B. Sequencing with series-parallel precedence constraints. *Math. Oper. Res. 4* (1979), 215-224

[Sellinger 79] Sellinger, P.G. et. al. Access Path Selection in a Relational Database Management System. In Proc. of ACM-SIGMOD Intl. Conf. on Mgt. of Data, (1979), 23-34

[Tsur 85] Tsur, S., and Zaniolo, C. LDL: A logic-based data language, submitted for publication.

[Ullman 85] Ullman, J. D. Implementation of logical query languages for databases, *TODS*, 10, 3, 1985, pp 289-321

[Whang 85] Whang, K. Y. Query Optimization in Office-by-Example, IBM Research report, RC 11571, (1985).

[Zaniolo 85] Zaniolo, C. The representation and deductive retrieval of complex objects, Proc. of 11th VLDB, pp 458-469, 1985