Optimization of Queries with User-defined Predicates

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Kyuseok Shim*
IBM Almaden Research Center
shim@bell-labs.com

Abstract

Relational databases provide the ability to store user-defined functions and predicates which can be invoked in SQL queries. When evaluation of a user-defined predicate is relatively expensive, the traditional methods of evaluating predicates as early as possible is no longer a sound heuristic. There are two previous approaches for optimizing such queries. However, none of these approaches is able to guarantee the optimal plan over the desired execution space. We present an efficient technique that is able to guarantee the choice of an optimal plan over the desired execution space. The optimization algorithm that we present has the desirable properties that (a) it is an extension of the algorithm used by commercial optimizers and never requires exhaustive enumeration of join ordering, (b) the complexity of the algorithm is bounded by a polynomial in the number of user-defined functions and (c) requires no special assumptions on the cost formulas for join. We also propose a *conservative local heuristic* that is even simpler but produces nearly optimal plans. We have implemented the algorithms by extending a System-R style optimizer.

*Currently at Bell Laboratories, Murray Hill.

Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996

1 Introduction

In order to efficiently execute complex database applications, many major relational database vendors provide the ability to define and store user-defined functions. Such functions can be invoked in SQL queries and make it easier for developers to implement their applications. However, such extensions make the task of the execution engine and optimizer more challenging. In particular, when user-defined functions are used in the Where clause of SQL, such predicates cannot be treated as SQL built-in predicates. If the evaluation of such a predicate involves a substantial CPU and I/O cost, then the traditional heuristic of evaluating a predicate as early as possible may result in a significantly suboptimal plan. We will refer to such predicates as *user-defined* (or, *expensive*) predicates.

Consider the problem of identifying potential customers for a mail-order distribution. The mail-order company wants to ensure that the customer has a high credit rating, is in the age-group 30 to 40, resides in the San Francisco bay area, and has purchased at least $1,000 worth of goods in the last year. Such a query involves a join between the Person and the Sales relation and has two user-defined functions zone and high_credit_rating.

```
Select name, street_address, zip
From   Person, Sales
Where  high_credit_rating(ss_no)
       and age In [30,40]
       and zone(zip) = "bay area"
       and Person.name = Sales.buyer_name
Group By name, street_address, zip
Having Sum(sales.amount) > 1000
```

Let us assume that the predicate high_credit_rating is expensive. In such a case, we may evaluate the predicate after the join so that fewer tuples invoke the above expensive predicate. However, if the predicate is very selective, then it may still be better to execute high_credit_rating so that the cost of the join is reduced. Such queries involving user-defined predicates occur in many applications, e.g., GIS and multi-media.

This paper shows how commercial optimizers, many of which are based on system R style dynamic programming algorithm [SAC+79], can be extended easily to be able to optimize queries with user-defined predicates. We propose an easy extension of the traditional optimizer that is efficient and that guarantees the optimal. We associate a *per tuple cost* of evaluation and a *selectivity* with every user-defined predicate (as in [HS93]). While the task of optimizing queries with user-defined predicates is important, there are other interesting directions of research in user-defined predicates, e.g., use of semantic knowledge, e.g., [PHH92, CS93].

As pointed out earlier, the traditional heuristic of evaluating predicates as early as possible is inappropriate in the context of queries with user-defined predicates. There are two known approaches to optimizing queries that treat user-defined predicates in a special way. The first technique, used in LDL [CGK89] is exponential in the number of expensive predicates and it fails to consider the class of traditional plans where user-defined predicates are evaluated as early as possible. The second technique, known as Predicate Migration [HS93] is polynomial in the number of expensive predicates and takes into consideration the traditional execution space as well. However, this algorithm *cannot* guarantee finding the optimal plan. Moreover, in the worst case, it may need to exhaustively enumerate the space of joins ($O(n!)$ in the number of joins n in the query).

Our algorithm finds the optimal plan without ever requiring to do an exhaustive enumeration of the space of join orderings. The complexity of the algorithm is polynomial in the number of user-defined functions[1]. Our approach does not require any special assumptions about the execution engine and the cost model. In designing this optimization algorithm, we discovered a powerful pruning technique (*pushdown rule*) that has broader implication in other optimization problems as well [CS96].

Although the optimization algorithm that guarantees the optimal has satisfactory performance for a large class of queries, its complexity grows with the increasing query size. Therefore, we wanted to investigate if simpler heuristics can be used as an alternative. The *conservative local* heuristic that we present *guarantees* optimality in several cases and experimental results show that it chooses an execution plan very close to the optimal while being computationally inexpensive. Thus, this heuristic serves as an excellent alternative where query size or complexity of the optimization algorithm is a concern.

We have implemented the optimization algorithm as well as the heuristic by extending a System-R style optimizer. We present experimental results that illustrate the characteristics of the optimization algorithms proposed in this paper.

The rest of the paper is organized as follows. In the next section, we review the System R optimization algorithm [SAC+79] which is the basis of many commercial optimizers. Next, we describe the desired execution space and review the past work on optimizing queries with user-defined predicates. Sections 4 and 5 describe the optimization algorithm and the conservative local heuristic respectively. The performance results and implementation details are given in Section 6.

2 System R Dynamic Programming Algorithm

Many commercial database management systems have adopted the framework of the System R optimizer [SAC+79] which uses a dynamic programming algorithm. The execution of a query is represented syntactically as an *annotated join tree* where the internal node is a join operation and each leaf node is a base relation. The annotations provide the details such as selection predicates, the choice of access paths, join algorithms and projection attributes of the result relation. The set of all annotated join trees for a query that is considered by the optimizer will be called the *execution space* of the query. A cost function is used to determine the cost of a plan in the execution space and the task of the optimizer is to choose a plan of minimal cost from the execution space. Most optimizers of commercial database systems restrict search to only a subset of the space of join ordering. Most optimizers of commercial database systems restrict search to only a subset of the space of join ordering. In many optimizers, the execution space is restricted to have only *linear* join trees, whose internal nodes have at least one of its two child nodes as a leaf (base relation). In other words, a join with N relations is considered as a linear sequence of 2-way joins. For each intermediate relation, the cardinality of the result size and other statistical parameters are estimated.

Figure 1 (adopted from [GHK92]) illustrates the System R dynamic programming algorithm that finds an optimal plan in the space of linear (left-deep) join trees [SAC+79]. The input for this algorithm is a select-project-join (SPJ) query on relations $R_1,...,R_n$. The

[1]The complexity is exponential in the number of joins. This is not unexpected since the traditional join optimization problem itself is NP-hard.

```
procedure DP_Algorithm:
for i := 2 to $n$ do {
    for all $S \subseteq \{R_1, ..., R_n\}$ s.t. $||S|| =$ i do {
        bestPlan := a dummy plan with infinite cost
        for all $R_j, S_j$ s.t. $S = \{R_j\} \cup S_j$ do {
            p := joinPlan(optPlan($S_j$), $R_j$)
            if cost(p) $\leq$ cost(bestPlan)
                bestPlan := p
        }
        optPlan($S$) := bestPlan
    }
}
return($optPlan(\{R_1, ..., R_n\})$)
```

Figure 1: System R Algorithm for Linear Join Trees

function joinPlan(p,R) extends the plan p into another plan that is the result of p being joined with the base relation R in the best possible way. The function cost(p) returns the cost of the plan p. Optimal plans for subsets are stored in the optPlan() array and are reused rather than recomputed.

The above algorithm does not expose two important details of the System R optimization algorithm. First, the algorithm uses *heuristics* to restrict the search space. In particular, all selection conditions and secondary join predicates are evaluated as early as possible. Therefore, all selections on relations are evaluated before any join is evaluated. Next, the algorithm also considers *interesting orders*. Consider a plan P for $R_1 \bowtie R_2$ that uses sort-merge join and costs more than another plan P' that uses hash-join. Nonetheless, P may still be the optimal plan if the sort-order used in P can be reused in a subsequent join. Thus, the System R algorithm saves not a single plan, but multiple optimal plans for every subset S in the Figure, one for each distinct such order, termed *interesting order* [SAC+79]. Thus, a generous upper bound on the number of plans that must be optimized for a query with joins among n tables is $O(2^n)$ (the number of subsets of n tables) times the number of interesting orders.

3 Execution Space and Previous Approaches

As mentioned earlier, for traditional SPJ queries, many optimizers find an optimal from the space of linear join orderings only. When user-defined predicates are present, the natural extension to this execution space consists of considering linear sequence of joins, and allowing an expensive predicate to be placed following any number of (including zero) joins. Thus, an expensive selection condition can be placed either immediately following the scan of the relation on which it applies, or after any number of joins following the scan. Likewise, an expensive secondary join predicate can be placed either immediately after it becomes evaluable (following the necessary joins), or after any number of subsequent joins. In other words, this execution space restricts the join ordering to be linear but allows expensive predicates to be *freely* interleaved wherever they are evaluable. We refer to this execution space as *unconstrained linear join trees*. This is the same execution space that is studied in [HS93, Hel94]. In this section, we discuss two approaches that have been studied in the past for optimizing queries with user-defined predicates.

3.1 LDL Approach

In this approach, an expensive predicate is treated as a relation from the point of view of optimization. This approach was first used in the LDL project at MCC [CGK89] and subsequently at the Papyrus project at HP Laboratories [CS93]. Viewing expensive predicates as relations has the advantage that the System-R style dynamic programming algorithm can be used for enumerating joins as well as expensive predicates. Thus, if e is an expensive predicate and R_1 and R_2 are two relations, then the extended join enumeration algorithm will treat the optimization problem as that of ordering R_1, R_2 and e using the dynamic programming algorithm.

Shortcoming of the Approach:

This approach suffers from two drawbacks both of which stem from the problem of over-generalizing and viewing an expensive predicate as a relation. First, the optimization algorithm is exponential not only in the number of relations but also in the number of expensive predicates. Let us consider the case where only linear join trees are considered for execution. Thus, in order to optimize a query that consists of a join of n relations and k expensive predicates, the dynamic programming algorithm will need to construct $O(2^{n+k})$ optimal subplans. In other words, the cost of optimizing a relation with n relations and k expensive predicates will be as high as that of optimizing (n+k) relations. Another important drawback of this approach is that if we restrict ourselves to search only linear join trees, then the algorithm cannot be used to consider all plans in the space of unconstrained linear trees. In particular, the algorithm fails to consider plans that evaluate expensive predicates on both operands of a join prior to taking the join [Hel94]. For example, assume that R_1 and R_2 are two relations with expensive relations

e_1 and e_2 defined on them. Since the LDL algorithm treats expensive predicates and relations alike, it will only consider linear join sequences of joins and selections. However, the plan which applies e_1 on R_1 and e_2 on R_2 and then takes the join between the relations R_1 and R_2, is not a linear sequence of selections and joins. Thus, this algorithm may produce plans that are significantly worse than plans produced by even the traditional optimization algorithm.

3.2 Predicate Migration

Predicate Migration algorithm improves on the LDL approach in two important ways. First, it considers the space of unconstrained linear trees for finding a plan, i.e., considers pushing down selections on both operands of a join. Next, the algorithm is polynomial in the number of user defined predicates. However, the algorithm takes a step backwards from the LDL approach in other respects. This will be discussed later in this section.

We will discuss two aspects of this approach. First, we will discuss the *predicate migration* algorithm, which given a join tree, chooses a way of interleaving the join and the selection predicates. Next, we will describe how predicate migration may be integrated with a System R style optimizer [HS93, Hel94].

The predicate migration algorithm takes as input a join tree, annotated with a given join method for each join node and access method for every scan node, and a set of expensive predicates. The algorithm places the expensive predicates in their "optimal" (see discussion about the shortcomings) position relative to the join nodes. The algorithm *assumes* that join costs are linear in the sizes of the operands. This allows them to assign a *rank* for each of the join predicate in addition to assigning ranks for expensive predicates. The notion of rank has been studied previously in [MS79, KBZ86]. Having assigned ranks, the algorithm iterates over each *stream*, where a stream is a path from a leaf to a root in the execution tree. Every iteration potentially rearranges the placement of the expensive selections. The iteration continues over the streams until the modified operator tree changes no more. It is shown in [HS93] that convergence occurs in a polynomial number of steps in the number of joins and expensive predicates.

The next part of this optimization technique concerns integration with the System R style optimizer. The steps of the dynamic programming algorithm are followed and the optimal plan for each subexpression is generated with the following change. At each join step, the option of evaluating predicates (if applicable) is considered: Let P be the optimal plan of $\sigma_e(R \bowtie S)$ and P' be the optimal plan for $\sigma_e(R) \bowtie S$. If $cost(P) < cost(P')$, then the algorithm prunes the plan P' without compromising the optimal. However, if the plan for P' is cheaper, then dynamic programming *cannot* be used to extend the plan P' Rather, the plan P' is marked as *unprunable*. Subsequently, when constructing larger subplans, the algorithm ignores the unprunable plans. After the dynamic programming algorithm terminates, each such unprunable plans needs to be extended through *exhaustive enumeration*, i.e., all possible ways of extending each unprunable plan are considered.

Shortcomings of the Approach:

This approach to optimization has three serious drawbacks that limit its applicability. First, the algorithm requires that cost formulas of join to be linear in the sizes of the inputs. Next, the algorithm *cannot guarantee* an optimal plan even if a linear cost model is used. This is because the use of predicate migration algorithm may force estimations to be inaccurate. In a nutshell, predicate migration requires a join predicate to be assigned a *rank*, which depends on the cost of the join and the latter is a function of the input sizes of the relations. Unfortunately, the input sizes for the join depends on whether the expensive predicates have been evaluated! This cyclic dependency forces predicate migration to make an ad-hoc choice in calculating the rank. During this step, the algorithm potentially underestimates the join cost by assuming that all expensive predicates have been pushed down. This ad-hoc assumption sacrifices the guarantee of the optimality (See Section 5.2 of [Hel94] for a detailed discussion). Finally, the global nature of predicate migration hinders integration with a System R style dynamic programming algorithm. The algorithm may degenerate into exhaustive enumeration. Let us consider a query that has n relations and a *single* designated expensive predicate e on the relation R_1. Let us assume that for the given database, the traditional plan where the predicate e is evaluated prior to any join, is the optimal plan. In such a case, plans for $\sigma_e(R_1) \bowtie R_i$ $(i \neq 1)$ will be marked as unprunable. For each of these plans, there are $(n-2)!$ distinct join orderings and for each of these join orderings, there can be a number of join methods. Thus, in the worst case, the optimization process requires *exhaustive enumeration* of the join space.

4 Dynamic Programming Based Optimization Algorithms

Our discussion of the previous section shows that none of the known approaches are guaranteed to find an optimal plan over the space of unconstrained linear join trees. In this section, we present our optimization algorithm which is *guaranteed* to produce an optimal plan over the above execution space. To the best of our knowledge, this is the first algorithm that provides such a guarantee of optimality. The techniques presented in this section are readily adaptable for other join execution spaces as well (e.g., bushy join trees) [CS96].

Our algorithm has the following important properties as well: (1) it is remarkably *robust*. It is free from special restrictions on cost model or requirements for caching (2) The algorithm integrates well with dynamic programming based algorithm used in commercial optimizers, and *never* requires exhaustive enumeration. (3) The algorithm is *polynomial* in the number of user-defined predicates. We provide a succinct characterization of what makes this optimization problem polynomial and the parameters that determine its complexity of optimization.

Thus, our algorithm successfully addresses the shortcomings of the Predicate migration algorithm without sacrificing the benefit of considering the execution space of unconstrained linear join trees and ensuring that the complexity of optimization grows only polynomially with the increasing number of user-defined functions.

For notational convenience, we will indicate ordering of the operators in a plan by nested algebraic expressions. For example, $(\sigma_e(R_1) \bowtie R_2) \bowtie \sigma_{e'}(R_3)$ designates a plan where we first apply selection e on relation R_1, then join that relation with R_2 before joining it with the relation R_3, which has been reduced by application of a selection condition e'. In describing the rest of this section, we make the following two assumptions: (a) all user-defined predicates are *selections*. This assumption is to simplify the presentation. Our algorithms accommodate user-defined join predicates as well and preserves the guarantee of optimality as well as properties (1)-(3) above [CS96] (b) no traditional interesting orders are present. This assumption is for ease of exposition only.

We begin by presenting the "naive" optimization algorithm that guarantees optimality and has properties (1) through (3) above. Next, we present two powerful pruning techniques that significantly enhance the efficiency of the optimization algorithm, as will be shown later in the experimental section.

4.1 Naive Optimization Algorithm

The enumeration technique of our algorithm relies on clever use of the following two key observations:

Equivalent Plan Pruning Rule: The strength of the traditional join enumeration lies in being able to compare the costs of different plans that represent the same subexpression but evaluated in different orders. Since selection and join operations may be commuted, we can extend the same technique to compare and prune plans for queries that have the same expensive predicates and joins, i.e., if P and P' are two plans that represent the same select-project-join queries with the same physical properties, and if $Cost(P') < Cost(P)$, then P may be pruned. For example, we can compare the costs of the plans P and P' where P is the plan $(\sigma_e(R_1) \bowtie R_2) \bowtie \sigma_{e'}(R_3)$ and P' is the plan $(R_2 \bowtie \sigma_{e'} R_3) \bowtie \sigma_e(R_1)$.

Selection Ordering: Let us consider conjunction of a set of expensive selection predicates applied on a relation. The problem of ordering the evaluation of these predicates is the *selection ordering* problem. The complexity of selection ordering is *very different* from that of ordering joins among a set of relations. It is well-known that for traditional cost models, the latter problem is NP-hard. On the other hand, the selection ordering problem can be solved in *polynomial time*. Furthermore, the ordering of the selections does *not* depend on the size of the relation on which they apply. The problem of selection ordering was addressed in [HS93] (cf. [KBZ86, MS79, WK90]). It utilizes the notion of a rank. The *rank* of a predicate is the ratio $c/(1-s)$ where c is its cost per tuple and s is its selectivity.

Theorem 4.1: *Consider the query $\sigma_e(R)$ where $e = e_1 \wedge .. \wedge e_n$. The optimal ordering of the predicates in e is in the order of ascending ranks and is independent of the size of R.*

For example, consider two predicates e and e' with selectivities .2 and .6 and costs 100 and 25. Although the predicate e is more selective, its rank is 125 and the rank of e' is 62.5. Therefore evaluation of e' should precede that of e. The above technique of selection ordering can be extended to broader classes of boolean expressions [KMS92].

Ensuring Complete Enumeration Efficiently

Equivalent plan pruning rule allows us to compare two plans that represent the same expression. This observation will help us integrate well with the System R algorithm and avoid exhaustive enumeration (unlike predicate migration). On the other hand, selection ordering tells us that (in contrast to the LDL algorithm), we can treat selections unlike relations to make enumeration efficient. Indeed, this observation is what makes our algorithm polynomial in the number of user-defined predicates. Therefore, the challenge is to treat selections differently from joins while enumerating but to be still able to compare costs of two plans when they represent the same expression. In order to achieve this goal, we exploit the well-known idea of *interesting orders* [SAC+79] in a novel way.

We keep multiple plans that represent the join of the same set of relations but differ in the sets of predicates that have been evaluated. In other words, with every join plan, an additional "tag" is placed, which records the set of predicates that have been evaluated in the plan. Thus, a tag acts very much like an interesting order from the point of view of join enumeration. This is a useful way of thinking about enumerating the space of execution plans since the selection ordering rule ensures that we need "a few" tags. Notice that whenever two plans represent the join of same set of relations and agree on the tags, they can be compared and pruned.

Figure 2 illustrates the execution plans (and subplans) that need to be considered when there are three relations and two expensive selection predicates e_1 and e_2 on R_1. P_1, P_2 and P_3 are possible plans for $R_1 \bowtie R_2$ (each with differing tags). The plans from P_5 to P_{13} are for $R_1 \bowtie R_2 \bowtie R_3$. We will distinguish between P_5 and P_6 since they will have different tags, but will keep a single plan among P_5, P_7, P_{10} and P_{13}. We now formalize the above idea.

Tags

Let us consider a join step where we join two relations R_1 and R_2. Let us assume that $(p_1, \ldots, p_m)$ are the predicates applicable on R_1, in the order of the increasing rank. From the selection ordering criterion, we conclude that if the predicate p_j is applied on R_1 prior to the join, then so must all the predicates $p_1.,, p_{j-1}$. In other words, there can be at most $(m+1)$ possibilities for pushing down selections on R_1 : (a) not applying any predicate at all (b) applying the first j predicates only where j is between 1 and m. Likewise, if the predicates applicable on R_2 are $(q_1, \ldots, q_s)$, then there are at most $(s+1)$ possibilities. Thus, altogether there can be $(m+1)(s+1)$ plans for the join between R_1 and R_2 that differ in the applications of selections prior to the join. We can denote these plans by $P_{0,0}, \ldots, P_{m,s}$ where $P_{r,t}$ designates the plan that results from evaluating $(p_1, \ldots, p_r)$ on R_1 and $(q_1, \ldots, q_t)$ on R_2 prior to the join. The selection ordering plays a crucial role in reducing the number of tags from exponential to a polynomial in the number of user defined predicates. Observe that if we cannot have a linear ordering among selections, then we have to consider cases where any subset of the selection predicates are chosen for evaluation prior to the join. In that case, in the above join between R_1 and R_2, the number of plans can be as many as $2^{m+1}.2^{s+1}$.

We generalize the above idea in a straight-forward fashion. For a subquery consisting of the join among relations $\{R_{i1}, \ldots, R_{il}\}$, there will be *at most* $(m_1 + 1)(m_2+1)..(m_{l-1}+1)(m_l+1)$ optimal plans that need to be kept where m_j represents the number of expensive predicates that apply on R_{ij}. We will associate a *distinct tag* with each of these plans over the same subquery. We now sketch how tags may be represented. We assign a number to each expensive predicate according to the ordering by rank. If an user-defined selection condition is present over w of the relations (say, $R_{u1}, \ldots, R_{uw}$) in the query, then with each plan, we associate a vector of width w. If $< a_1, ..a_w >$ is the tag vector with a plan P, then it designates that all expensive predicates of rank lower or equal to a_j on R_{uj} have been evaluated in P for all $1 \leq j \leq w$. We defer a full discussion of the scheme for tagging to the extended version of our forthcoming report [CS96], but illustrate the scheme with the following example.

Example 4.2: Consider a query that represents a join among four relations $R_1, .., R_4$ and nine selections where the selections are numbered by their relative increasing rank. The relation R_1 has three predicates numbered 2,5,6. Let R_2 have three predicates 1,3,4. Let R_3 have predicates 7,8,9. The relation R_4 has no predicates defined. The tag vector has three positions, where the ith position represents predicates on the relation R_i. There are altogether 16 plans for join over $\{R_1, R_2\}$, each with a distinct tag. Consider the plan for the tag vector $< 5, 4, 0 >$. This plan can be joined with the relation R_3. Depending on the selection predicates evaluated prior to the join, there will be altogether 8 plans with different tag vectors that extend the above plan. In particular, a plan will be generated with a tag vector $< 5, 4, 8 >$. This plan can be compared with the plan obtained by extending a plan for $\{R_2, R_3\}$ with the tag vector $< 0, 4, 8 >$ through a join with R_1 and evaluating predicates 2 and 5 on R_1 prior to the join. ∎

In the above example, we illustrated how we can prune plans with the same tag vector and over the same set of relations. This is unlike the approaches in [HS93, Hel94] where once a user-defined predicate has been "pushed-down", the plan is unprunable.

Algorithm: The extensions needed to the algorithm in Figure 1 for the naive optimization algorithm are straightforward. There is no longer a single optimal plan for S_j (in Figure 1), but there may be multiple plans, one for each tag vector. Thus, we will need to iterate over the set of possible tags. For each such optimal plan S_j^t with a tag t, we consider generating all

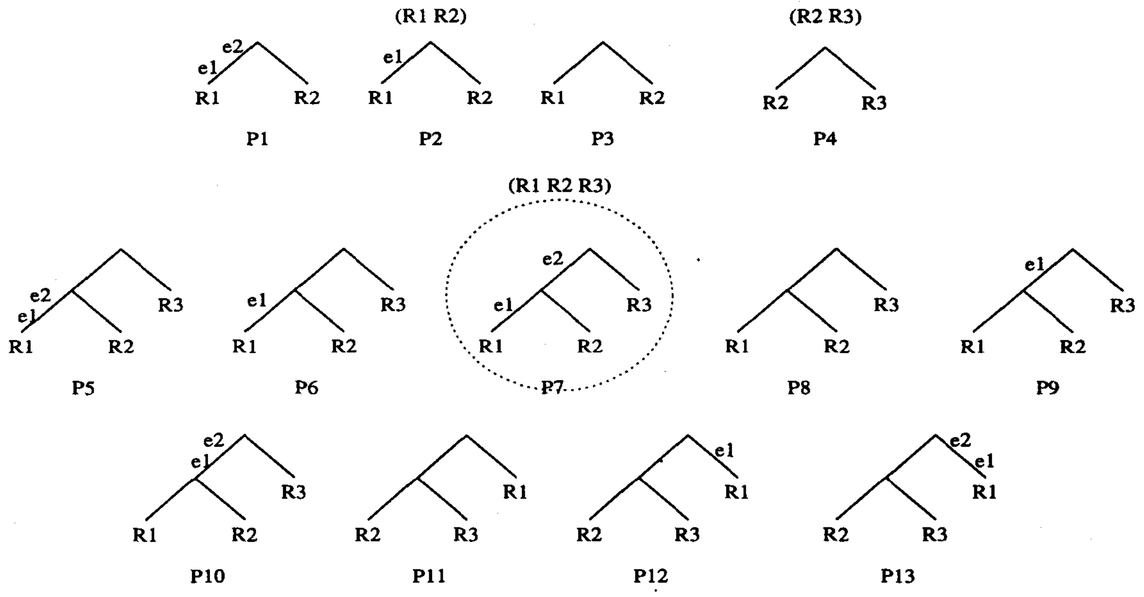


Figure 2: Search Space of Naive Optimization Algorithm

possible legal tags for S. For each such tag t', *joinPlan* needs to be invoked to extend the optimal plan S_j^t. We need to also ensure that we compare costs of plans that have the same tag.

4.2 Complexity

In the optimization algorithm that we presented, we exploited dynamic programming as well as selection ordering. The latter makes it possible for us to have an optimization algorithm which is polynomial in k whereas the former made it possible for us to retain the advantage of avoiding exhaustive enumeration of the join ordering. The efficiency of our algorithm is enhanced by the applications of pruning rules that will be described in the next section.

Let us consider a query that consists of a join among n relations and that has k user-defined predicates. Let us assume that only g of the n relations have one or more user-defined selection conditions. Furthermore, let w be the maximum number of expensive predicates that may apply on one relation. In such cases, the number of tags can be no more than $(1+w)^g$. Furthermore, we can show that the total number of subplans that need to be stored has a *generous* upper-bound of $2^n(1+w/2)^g$. Note that since n is the total number of relations and k is the total number of user-defined predicates, $g \leq n$ and $w \leq k$. Therefore, the above formula can be used to derive an upper-bound of $(2+k)^n$. Hence, for a given n, the upper-bound is a polynomial in k. The above is a very generous upper bound and a more detailed analysis will be presented in [CS96]. Observe that as in the case of traditional join enumeration, the complexity is exponential in n.

The analysis of our complexity shows that the complexity is sensitive to the *distribution* of predicates among relations as well as to the *number* of predicates that may apply to a single relation. In particular, if all user-defined predicates apply to the same relation, then the complexity is $O(2^n)(1+k/2)$, a *linear* function of k. The complexity of this algorithm grows with the number of relations over which user-defined predicates occur since they increase the number of tags exponentially. In the full paper, we study the effect of varying distributions of user-defined predicates on efficiency of the optimization algorithms [CS96].

It is important to recognize how we are able to avoid the worst cases that predicate migration algorithm encounters. Predicate migration algorithm has worst running time when user-defined predicates turn out to be relatively inexpensive (i.e., has low rank). It is so since in such cases, unprunable plans are generated (See Section 3). On the other hand, our optimization algorithm prepares for all possible sequences of predicate pushdown through the use of tags. Furthermore, since in many applications, we expect the number of expensive user-defined functions in the query to be a few and less than the number of joins, it is important to ensure that the cost of join enumeration does not increase sharply due to presence of a few user-defined predicates. However, as pointed out earlier, even with a single user-defined predicate over n joins, the worst-case complexity of predicate migration can be $O(n!)$. Our approach overcomes the above shortcoming of predicate migration effectively.

4.3 Efficient Pruning Strategies

The naive algorithm can compare plans that have the same tags only. In this section, we will augment our "naive" optimization algorithm with two pruning techniques. The pruning techniques that we propose here allow us to compare and prune plans that have *different* tags. These pruning techniques are sound, i.e., guaranteed not to compromise the optimality of the chosen plan.

Pushdown Rule

This rule says that if the cost of evaluating the selections (prior to the join) together with the cost of the join after the selections are applied, is less than the cost of the join without having applied the selections, then we should push down the selections [2]. For example, in Figure 2, if the cost of P_5 is less than the cost of P_8 we can prune P_8. In naive optimization algorithm, we had to keep both P_5 and P_8 since they had different tags, i.e., different numbers of expensive predicates were applied.

Lemma 4.3: *Let P' be a plan for the join $R \bowtie S$. Let P be a plan that applies an user-defined predicate e on the relation R before taking the join with S, (i.e., $\sigma_e(R) \bowtie S$). If $Cost(P) \leq Cost(P')$, then the plan P' may be pruned.*

We refer to the above lemma as the *pushdown rule*. The soundness of the above lemma follows from the observation that for SPJ queries with the same interesting order, the cost is a monotonic function of sizes of relations [CS96]. A consequence of this rule is that if P' is a plan that has a set S' of expensive predicates applied, then it can be pruned by another plan P over the same set of relations where (a) $Cost(P) \leq Cost(P')$ (b) P has a set S of expensive predicates applied where S is a superset of S' (therefore, $S \Rightarrow S'$). Given two plans over the same set of relations, we can easily check (b) by examining the tag vectors of P and P' [CS96]. If indeed (b) holds, then we say T *dominates* T', where T' and T are tags of P' and P. We can rephrase the above lemma to conclude the following:

Corollary 4.4: *If P and P' are two plans over the same set of relations with the tags T and T' such that T dominates T', and P is cheaper than P', then P' may be pruned.*

For a given plan P, the set of plans (e.g., P') that the above corollary allows us to prune will be denoted by *pushdown_expensive*(P).

[2]Strictly speaking, the lemma can be used to compare plans P and P' that have the same interesting order.

Example 4.5: Let us consider the previous example. For the plan that represents the join among $\{R_1, R_2, R_3\}$, there will be altogether 64 tags. However, if the cost of the plan with the tag $< 6, 4, 9 >$ is lower than that of $< 5, 4, 8 >$, we can use the pushdown rule to prune the latter plan. ∎

Pullover Rule

This rule says that if locally deferring evaluation of a predicate leads to a cheaper plan than the plan that evaluates the user-defined predicate before the join, then we can defer the evaluation of the predicate without compromising the optimal. The soundness of this rule uses the dynamic programming nature of the System R algorithm and can be established by an inductive argument. For example, if the cost of the plan extending P_6 with evaluation of e_2 (i.e., $\sigma_{e_2}(\sigma_{e_1} R_1 \bowtie R_2)$) is less than the cost of P_5 in Figure 2, we can prune P_5. In naive optimization algorithm, we had to keep both P_5 and P_6 since they had different tags, i.e., different number of predicates were applied to each of the plans.

Lemma 4.6: *Let e be a user-defined predicate on a relation R. Let P and P' represent the optimal plans for $\sigma_e(R \bowtie S)$ and $\sigma_e(R) \bowtie S$ respectively. If $Cost(P) \leq Cost(P')$, then the plan P' may be pruned.*

We refer to the above as the *pullover rule* since the plan P in the lemma corresponds to the case where the predicate is pulled up. This rule can also be used in the context of predicate migration to reduce the number of unprunable plans generated (cf. [HS93]). We can use the pullover rule for pruning plans as follows. Let us consider plans P and P' over the same set of relations but with different tags T and T'. If the tag T dominates T', then all predicates that are evaluated in T' are also evaluated in T. Let $Diff(T, T')$ represent the set of predicates that are evaluated in T but not in T'. We can then use the Pullover rule to obtain the following corollary. Intuitively, the corollary says that we can compare *cost*(P), with that of *cost*$(P') + \delta$ where δ is the cost of evaluating predicates $Diff(T, T')$ after the join in P'.

Corollary 4.7: *Let P and P' be two plans with tags T and T' over the same set of relations and T dominates T'. Let P'' be the plan obtained by applying the predicates in $Diff(T, T')$ to P'. If $cost(P'') \leq cost(P)$, then P may be pruned.*

For a given P, we can construct a set of all such plans P' each of which may be used to prune P. We can refer to the above set as *pullover_cheaper*(P). The following example illustrates the corollary. For example, consider

Example 4.5 with the following change: the cost of the plan P with the tag $T =< 6,4,9 >$ is higher than the cost of the plan P' with the tag $T' =< 5,4,8 >$. Notice that the tag $< 6,4,9 >$ dominates the tag $< 5,4,8 >$. The set $Diff(T,T') = \{6,9\}$. In such a case, the above lemma allows us to prune the plan P if the cost of the plan P' with the added cost of evaluating the set of predicates $\{6,9\}$ after the join exceeds the cost of P.

4.4 Optimization Algorithm with Pruning

In this section, we augment the naive optimization algorithm with the pruning strategies. The extended algorithm is presented in Figure 3. The *Plantable* data structure stores all plans that need to be retained for future steps of the optimizer. For every subset of relations, the data structure stores potentially multiple plans. The different plans correspond to different tags. Storing such plans requires a simple extension of the data structure used to represent plans with interesting orders in the traditional optimizers.

In determining the access methods and choice of join methods, the algorithm behaves exactly like the traditional algorithm in Figure 1. However, when there are s applicable user-defined predicates on the operand S_j and r applicable predicates on the operand R_j, the algorithm iteratively considers all $(r+1)(s+1)$ possibilities which corresponds to applying the first u predicates and the first v predicates on S_j and R_j respectively where the predicates are ordered by ranks. This is the inner loop of the algorithm and is represented by *extjoinPlan*. It should be noted that S_j is an intermediate relation and so the first u predicates on S_j may include predicates on multiple relations that have been joined to form S_j.

The choices of u and v uniquely determine the tag for the plan p in Figure 3. The plan p will be compared against plans over the *same set of relations* that have already been stored. The plan p is pruned and the iteration steps to the next (u,v) combination if one of the following two conditions holds: (1) If p is more expensive than the plan in the *Plantable* with the same tag, if any. (2) If the set of plans *pullover_cheaper*(p) is empty, i.e., the pullover rule cannot be used to prune p.

Otherwise, the predicate *addtotable*(p) becomes true and the plan p is added to *Plantable*. Next, this new plan p is used to prune plans that are currently in *plantable*. In the algorithm, we have designated this set of pruned plans by *pruneset*(p). They may be: (1) The stored plan with the same tag, if it exists in the *Plantable* and is more expensive. (2) The set of plans in *pushdown_expensive*(p) , i.e., plans that may be pruned with p using the pushdown rule.

```
procedure Extended_DP-Algorithm:
for i := 2 to $n$ do {
    for all $S \subseteq \{R_1, ..., R_n\}$ s.t. $||S|| = $ i do {
        bestPlan := a dummy plan with infinite cost
        for all $R_j, S_j$ s.t. $S = \{R_j\} \cup S_j$ do {
            $s$ := Number of evaluable predicates on $S_j$
            $r$ := Number of evaluable predicates on $R_j$
            for all u := 0 to $s$ do
            for all v := 0 to $r$ do
            p := extjoinPlan(optPlan($S_j$), $R_j$,u,v)
            if addtotable(p) then {
                remove pruneset(p)
                add p to Plantable
            }
        }
    }
}
for all plan q of $\{R_1, ..., R_n\}$ do
    Final(q') = complete the plan q
            and estimate its cost
return (MinCost(Final))
```

Figure 3: The Optimization Algorithm with Pruning for Linear Join Trees

At the end of the final join, we consider all plans over the relations $\{R_1, ..R_n\}$. Some of these plans may need to be completed by adding the step to evaluate the remainder of the predicates. Finally, the cheapest among the set of completed plans is chosen.

5 Conservative Local Heuristic

Although the optimization algorithm with novel pruning techniques guarantees the optimal plan and is computationally efficient, the *conservative local heuristic* that we propose in this section has remarkable qualities that make it an attractive alternative for implementation. First, incorporating the heuristic in an existing System-R style optimizer is easier since tags do not need to be maintained. Next, incorporating the heuristic increases the number of subplans that need to be optimized for a query by no more than a factor of 2 compared to the traditional optimization, *independent* of the number of user-defined predicates in the query. Finally, there are a number of important cases where the algorithm guarantees generation of an optimal execution plan.

The simplest heuristics correspond to pushing all expensive predicates down or deferring evaluation of all expensive predicates until the last join. These heuristics do not take into account the costs and selectivities of the predicates and therefore generate plans of low quality. Recently, a new heuristic, *Pullrank*, was proposed but it was found that the heuristic fails to generate plans of acceptable quality [Hel94]. We begin by describing PullRank, characterizing its shortcomings and then presenting the conservative local heuristic.

Pullrank maintains *at most one* plan over the same set of relations. At each join step, for every choice of the set of predicates that are pushed down, the Pullrank algorithm estimates the sum of the costs (we will call it *completion cost*) of the following three components (i) Cost of evaluating expensive predicates that are pushed down at this step (ii) Cost of the join, taking into account the selectivities of expensive predicates that are applied (iii) Cost of evaluating the remainder of the user-defined functions that are evaluable before the join but were deferred past the join. Pullrank chooses the plan that has the minimum completion cost. Thus, the algorithm greedily pushes down predicates if the cost of deferring the evaluation of predicates past the join is more expensive, i.e., if Pullrank decides that evaluating a predicate u before a join j is cheaper than evaluating the predicate u immediately following j, then evaluation of u will precede j in the final plan, i.e., Pullrank will *not* consider any plans where u is evaluated after j. Thus, Pullrank fails to explore such plans where deferring evaluation of predicates past *more than one* joins is significantly better than choosing to greedily push down predicates based on local comparison of completion costs.

In order to address the above drawback of Pullrank, the conservative local heuristic picks *one additional* plan (in addition to the plan picked by Pullrank) at each join step based on sum of the costs of (i) and (ii) only. Let us refer to this cost metric as *pushdown-join* cost. This is the same as assuming that *deferred predicates* are evaluated for "free" (i.e., cost component (iii) is zero). In other words, the plans chosen using such a metric favor deferring predicates unless the evaluation of predicates helps reduce the cost of the current join. Thus, since conservative local heuristic picks two plans, one for completion cost and the other for pushdown-join cost, it is possible that the plan where the predicate u is deferred past j as well as the plan where u is pushed down prior to j (chosen by Pullrank), is considered towards the final plan. Thus, conservative local heuristic can find optimal plans that Pullrank and other global heuristics fail to find due to its greedy approach. This is illustrated by the following example.

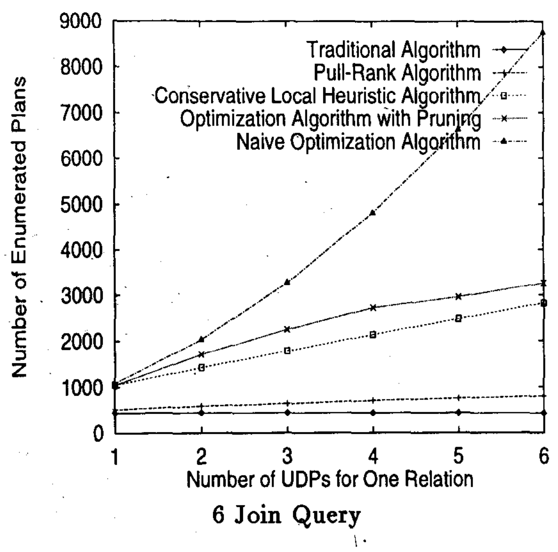
Example 5.1: Consider the query $Q = \sigma_e(R_1) \bowtie R_2 \bowtie R_3$. Let us assume that the plan $\sigma_e(R_1 \bowtie R_2) \bowtie R_3)$ is optimal. Note that none of the global heuristic that either pushes down or pulls up all the selections can find the optimal. If the plan for $\sigma_e(R_1) \bowtie R_2$ is cheaper than $\sigma_e(R_1 \bowtie R_2)$, then pullrank greedily pushes down P and fails to obtain the optimal. However, our algorithm uses the plan $R_1 \bowtie R_2$ in the next join step to obtain the optimal. This is an example where a pullup followed by a pushdown was optimal and therefore only our algorithm was able to find it. ▮

For join of every subset of relations, at most two plans are stored by conservative local heuristic. Therefore, we never need to consider optimizing more than $O(2^{n+1})$ plans. Thus, unlike the algorithm in Figure 3, the number of subplans that need to be optimized does not grow with the increasing number of user-defined predicates. In general, conservative local heuristic may miss an optimal plan. Intuitively, this is because in this algorithm, distinctions among the tags are not made. Nevertheless, the experimental results indicate that the quality of the plan is very close to the optimal plan [CS96]. Furthermore, as the following lemma states, the conservative local heuristic produces an optimal plan in several important special cases.

Lemma 5.2: *The conservative local heuristic produces an optimal execution plan if any one or more of the following conditions are true:(1) The query has a single join. (2) The query has a single user-defined predicate. (3) The optimal plan corresponds to the case where all the predicates are pushed down. (4) The optimal corresponds to the case where all the predicates are deferred until all the joins are completed.*

6 Performance Evaluation

We implemented the optimization algorithms proposed in this paper by extending a System R style optimizer. In this section, we present results of doing performance evaluations on our implementations. In particular, we establish:
(1) The pruning strategies that we proposed improve the performance of naive optimization algorithm significantly.
(2) The plans generated by the traditional optimization algorithm suffers from poor quality.
(3) The plans generated by PullRank algorithm are better (less expensive) than the plans generated by a traditional optimizer, but is still significantly worse than the optimal.
(4) The conservative local heuristic algorithm reduces the optimization overhead and it generates plans that are very close to the optimal.



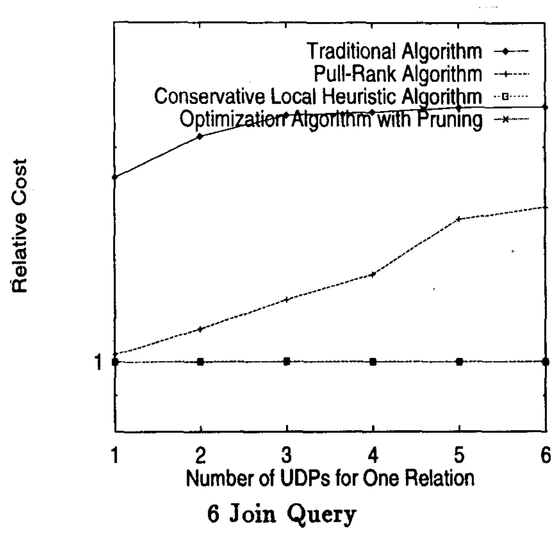


Figure 4: Performance on Varying Number of User-defined Predicates

Experimental Testbed

Experiments ware performed on an IBM RS/6000 workstation with 128 MB of main memory, and running AIX 3.2.5. We have used an experimental framework similar to that in [IK90, CS94]. The algorithms were run on queries consisting of equality joins only. The queries were tested with a randomly generated relation catalog where relation cardinalities ranged from 1000 to 100000 tuples, and the numbers of unique values in join columns varied from 10% to 100% of the corresponding relation cardinality. The selectivity of expensive predicates were randomly chosen from 0.0001 to 1.0 and the cost per tuple of expensive predicates was represented by the number of I/O (page) accesses and was selected randomly from 1 to 1000. Each query was generated to have two projection attributes. Each page of a relation was assumed to contain 32 tuples. Each relation had four attributes, and was clustered on one of them. If a relation was not physically sorted on the clustered attribute, there was a B^+-tree or hashing primary index on that attribute. These three alternatives were equally likely. For each of the other attributes, the probability that it had a secondary index was 1/2, and the choice between a B^+-tree and hashing secondary index were again uniformly random. We considered block nested-loops, merge-scan, and simple and hybrid hash joins. The interesting orders are considered for storing sub-plans. In our experiment, only the cost for number of I/O (page) accesses was accounted.

We performed two sets of experiments. In the first set, we varied the *number* of user-defined predicates that apply on one relation. In the second set, we varied the *distribution* of the user-defined predicates on *multiple* relations in the query. Due to lack of space, we present only the experiments where the number of user-defined selections that apply on a relation are varied. The results of the other experiments will be discussed in [CS96]. The second set of experiments shed light on how the distribution of the user-defined predicates among relations in the query influences the cost of optimization. The results also shows how our conservative local heuristic sharply reduces the overhead of optimization under varying distributions.

Effect of Number of User defined Predicates

Due to the lack of space, we will show the results for 6-join (i.e. join among 7 relations) queries only but similar results were obtained for other queries (e.g. 4-join and 10-join queries) as well. The detailed performance study with various queries will be presented in [CS96]. In this experiment, one relation in the query was chosen randomly and the number of expensive predicates applicable was varied from 1 to 6. The results presented here for each data point represents averages of 100 queries, generated randomly.

We experimented how the optimization algorithms behave as we increase the number of expensive predicates for the randomly selected relation in the queries. Figure 4 shows the *number of enumerated plans* and the *quality of plans* generated by each algorithm. A comparison of the performances of the naive optimization algorithm and optimization algorithm with pruning shows that our proposed pruning techniques are extremely effective. Note that both these algorithms are guaranteed to be optimal. Over all queries, the naive optimization algorithm enumerated about 3 times more plans than optimization algorithm with pruning.

The result on quality of plans shows the relative cost of plans generated by each algorithms. The cost of plan generated by optimization algorithm with pruning was scaled as 1.0. Since naive optimization algorithm and optimization algorithm with pruning always generate optimal plans, 1.0 represents the cost of both optimal plans. The figure illustrates that the quality of plan generated by traditional optimizer suffers significantly while the quality of plan generated by PullRank algorithm gets worse as the number of expensive predicates increases.

Conservative local heuristic chooses plans that are identical to or very close to the optimal[3]. This is illustrated by the fact that the graphs for the heuristic and the optimization algorithm are practically indistinguishable. Although in this experiment, conservative local heuristic doesn't reduce the number of enumerated plans significantly compared to the optimization algorithm with pruning, this observation does *not* extend in general, particularly when the user-defined selections are distributed among multiple relations In the latter cases, the conservative local heuristic proves to be the algorithm of choice, since it continues to choose plans close to the optimal plan with much less optimization overhead [CS96].

Acknowledgement: We are indebted to Joe Hellerstein for giving us detailed feedback on our draft in a short time. The anonymous referees provided us with insightful comments that helped improve the draft. Thanks are due to Umesh Dayal, Nita Goyal, Luis Gravano and Ravi Krishnamurthy for their help and comments. Without the support of Debjani Chaudhuri and Yesook Shim, it would have been impossible to complete this work.

References

[CGK89] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for LDL. In *Proceedings of the 15th International VLDB Conference*, pages 195–203, Amsterdam, The Netherlands, August 1989.

[CS93] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proceedings of the 19th International VLDB Conference*, Dublin, Ireland, August 1993.

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proceedings of the 20th International VLDB Conference*, Santiago, Chile, Sept 1994.

[CS96] S. Chaudhuri and K. Shim. Optimization with user-defined predicates. Technical report, 1996. In preparation.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proceedings of the 1992 ACM-SIGMOD Conference on the Management of Data*, pages 9–18, San Diego, CA, May 1992.

[Hel94] J. M. Hellerstein. Predicate migration placement. In *Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data*, pages 325–335, Minneapolis, MN, May 1994.

[HS93] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimization queries with expensive predicates. In *Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data*, pages 267–276, Washington, D.C., May 1993.

[IK90] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Proceedings of the 1990 ACM-SIGMOD Conference on the Manage ment of Data*, pages 312–321, Atlantic City, NJ, May 1990.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zanialo. Optimization of nonrecursive queries. In *Proceedings of International Conference on Very Large Data Bases*, pages 128–137, Kyoto, Japan, Aug 1986.

[KMS92] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object-bases. In *Proceedings of the 18th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Vancouver*, August 1992.

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

[PHH92] H. Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query optimization in starburst. In *Proceedings of the 1992 ACM-SIGMOD Conference on the Management of Data*, pages 39–48, San Diego, CA, May 1992.

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

[WK90] K-Y. Whang and R. Krishnamurthy. Query optimization in a memory-resident domain relational calculus database system. *ACM Transactions on Database Systems*, 15(1):67–95, March 1990.

[3]Note that the optimization algorithm with pruning becomes the same as conservative local heuristic algorithm when the number of expensive predicate is one. Thus, the number of enumerated plans for both algorithms are the same when we have only one expensive predicate.