Optimization of Queries with User-Defined Predicates

SURAJIT CHAUDHURI
Microsoft Research
and
KYUSEOK SHIM
Bell Laboratories

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 method of evaluating predicates as early as possible is no longer a sound heuristic. There are two previous approaches for optimizing such queries. However, neither is able to guarantee the optimal plan over the desired execution space. We present efficient techniques that are able to guarantee the choice of an optimal plan over the desired execution space. The *naive optimization algorithm* is very general, and therefore is most widely applicable. The *optimization algorithm with complete rank-ordering* improves upon the naive optimization algorithm by exploiting the nature of the cost formulas for join methods and is polynomial in the number of user-defined predicates (for a given number of relations). We also propose *pruning rules* that significantly reduce the cost of searching the execution space for both the naive algorithm as well as for the optimization algorithm with complete rank-ordering, without compromising optimality. We also propose a *conservative local heuristic* that is simpler and has low optimization overhead. Although it is not always guaranteed to find the optimal plans, it produces close to optimal plans in most cases. We discuss how, depending on application requirements, to determine the algorithm of choice. It should be emphasized that our optimization algorithms handle user-defined selections as well as user-defined join predicates uniformly. We present complexity analysis and experimental comparison of the algorithms.

Categories and Subject Descriptors: H.2 [**Information Systems**]: Database Management

General Terms: Algorithms, Management, Performance

Additional Key Words and Phrases: Query Optimization, Dynamic Programming, User-defined Predicates

Authors' addresses: S. Chaudhuri, Microsoft Research, One Microsoft Way, Redmond, WA 98052; K. Shim, Bell Laboratories, 600 Mountain, Murray Hill, NJ 07974.

1. INTRODUCTION

In order to efficiently support complex database applications, many major relational database vendors support user-defined functions. Such functions can be invoked in SQL queries, making it easier for developers to implement their applications with significantly greater efficiency. However, such extensions also make the task of the execution engine and the optimizer more challenging. In particular, when user-defined functions are used in the `Where` clause of SQL, such predicates, henceforth called *user-defined* (or *expensive*) predicates cannot be treated as SQL built-in predicates. Unlike evaluation of built-in SQL predicates, the evaluation of such predicates may involve substantial CPU and I/O cost. In such cases, the traditional heuristic of evaluating predicates as early as possible may result in significantly suboptimal plans, as the following example demonstrates.

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 from 30 to 40, resides in the San Francisco bay area, and has purchased at least $1,000 worth of goods in the last year. This query has a join between the Person and the Sales relation and 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 user-defined predicate `high_credit_rating` is expensive. In such a case, we may evaluate the predicate after the join so that fewer tuples invoke the expensive predicate. However, if the predicate is very selective, then it is better to execute `high credit rating` first so that the cost of the join is reduced.

As the above example illustrates, 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. Given a SPJ query, the first technique, used in LDL [Chimenti et al. 1989], treats a user-defined predicate much like a relation (see Section 2) and is exponential in the number of user-defined predicates. This technique 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 [Hellerstein and Stonebraker 1993] is polynomial in the number of user-defined predicates in a SPJ query and takes into consideration the traditional execution space as well. However, this algorithm *cannot* guarantee finding the optimal plan, and in some situations may need to exhaustively enumerate the space of join ordering.

This paper shows how commercial optimizers, many of which are based on a system R-style dynamic programming algorithm [Selinger et al. 1979], can be extended easily to optimize queries with user-defined predicates and guarantee optimality of the plan. First, we present an algorithm that can guarantee the optimal evaluation of queries with user-defined predicates with no assumptions on the cost model except that it follows the principle of optimality [Cormen et al. 1990]. However, the complexity of the algorithm is exponential in the number of user-defined predicates. Next, we show that under broad assumptions about the cost formulas for join methods, satisfied by common join methods, we can reduce the complexity of the first algorithm to be polynomial in the number of user-defined predicates.[1] This required proving the nontrivial result that, for the given assumptions on the cost model, placement of the predicates in an execution tree must follow the *rank ordering* (see Section 4.1) of user-defined predicates. Finally, we discovered powerful pruning techniques that further reduce the search space of the optimizer without compromising the optimal.

Although the above optimization algorithms guarantee the optimality of plans and have satisfactory performance for a large class of queries, their complexity grows monotonically with increasing query size. Therefore, we explore if computationally inexpensive heuristics can be used as viable alternatives. The *conservative local* heuristic that we present has very little overhead above traditional optimizers. It is able to guarantee the optimality of the chosen plan in several cases, and experimental results show that it typically chooses an execution plan very close to the optimal. This heuristic serves as an excellent alternative where query size or complexity of the optimization algorithm is a concern.

Our techniques provide optimization algorithms for queries containing user-defined selections as well join predicates. In the rest of this paper, we use the term user-defined predicate (or expensive predicate) to generically refer to any user-defined predicate whether it occurs as a selection or a join predicate. In contrast, we reserve the term “join predicate” to refer to traditional join predicates in queries.

We implemented the optimization algorithms by extending a System R style optimizer. We present worst-case complexity analysis and 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 present the cost model and execution space in traditional framework and the extensions needed to handle user-defined predicates (selections as well

[1]The complexity is exponential in the number of joins. This is to be expected, since the traditional join optimization problem itself is NP-hard.

as join predicates) in query optimization. In Section 3, we review the System R optimization algorithm [Selinger et al. 1979], which is the basis of many commercial optimizers. Next, we describe the desired execution space and review past work on optimizing queries with user-defined predicates. Sections 4 and 5 present the two optimization algorithms that find the optimal plan. In Section 6, we provide the complexity analysis for these algorithms. The pruning techniques and extensions of the optimization algorithms to incorporate these techniques are presented in Section 7. Section 8 presents the conservative local heuristic that is simpler but produces a nearly optimal plan. The performance results and implementation details are given in Section 9.

2. OPTIMIZATION FRAMEWORK

In this paper, we consider Select-Project-Join queries containing user-defined predicates. We assume that the `Where` clause of the query consists of a *conjunction* of built-in and user-defined predicates. Our techniques also extend to queries that have a more general form. Many commercial database management systems have adopted the framework of the System R optimizer [Selinger et al. 1979] for optimization. We use the same framework for studying the problem of optimization of queries with user-defined predicates. First, we describe the traditional optimization framework (cost model as well as execution space) and then explain how we need to extend the framework when user-defined predicates are present.

2.1 Traditional Framework

We describe the traditional optimization framework for SPJ queries. An *execution plan* of such a query is represented syntactically as an *annotated join tree* where the internal node is a join operation and each leaf node is a scan of a base relation. The edges of the tree indicate the flow of data. The annotations provide details such as selection predicates, the choice of access paths, join methods, and projection attributes of the result relation. The set of all execution plans for a query that is considered by the optimizer is the *execution space* of the query. A cost function is used to determine the *cost* of an execution plan in the execution space. The task of the optimizer is to choose a plan of minimal cost from the execution space. Optimizers of commercial database systems often restrict search for a plan to only a subset of the space of all annotated join trees. For example, the execution space may be restricted to have only *linear* join trees. A linear join tree represents an execution that is a linear sequence of joins. Thus, each internal join node has at least one of its two child nodes as a leaf (base relation). A join among multiple relations is represented as a linear sequence of 2-way joins. In contrast, in a *bushy* join tree, both the operands of one or more join nodes may be an intermediate (computed) relation.

The traditional optimization framework only concerned itself with handling conditions in the `Where` clause that used inexpensive built-in predicates. It was assumed that the cost of evaluating built-in predicates is zero, and so conditions are evaluated in an eager fashion. A *selectivity* is associated with each condition to estimate the effect of applying the predicate on the relation. Another assumption that is commonly made in traditional query optimization is that of *independence* of conditions, i.e., if $p1$ and $p2$ are two predicates, then $\mathit{Selectivity}(p1 \wedge p2) = \mathit{Selectivity}(p1) \times \mathit{Selectivity}(p2)$.

It is assumed that the problem of query optimization satisfies the *principle of optimality* [Cormen et al. 1990]. The principle of optimality forms the basis of dynamic programming-based query optimization algorithms that are used in commercial optimizers. In other words, for linear join trees, the optimal plan for join of $(n + 1)$ relations $S_{n+1} = \{R_1, \ldots R_{n+1}\}$ must be obtained by extending some optimal plan S over a subset (of size n) of the relations in S_{n+1}, i.e., $S \subseteq S_{n+1}$ and the cardinality of S is n. It should be noted that when subplans that represent the join of the same set of relations but differ in their physical properties (e.g., sort-order), they cannot be compared, i.e., optimal plans for each of the physical properties need to be extended to guarantee that the optimality of the final plan is not compromised.

Finally, optimization algorithms based on dynamic programming need to estimate the sizes of the intermediate relations to choose among alternative execution plans. The optimizer estimates the size of the relation represented by a node in the execution tree by using the sizes of the children nodes in the execution tree. For example, they compute the sizes of intermediate relations for $\sigma_e(E)$ from the estimated sizes for E and that of $E \bowtie R$ from the estimated sizes of E and R. As in all past optimization work, we need to assume that the cost model is *consistent*, i.e., the sizes of resulting relations of two execution trees that would result in the same relation must also be estimated to be the same.

2.2 Extensions for User-Defined Predicates

Unlike built-in predicates, we associate a *per tuple* cost for evaluation with every user-defined predicate. In other words, if a relation has r tuples then the cost of evaluating a user-defined predicate p on the relation is rc_p where c_p is the per tuple cost of evaluating the predicate p. As in the case of built-in predicates, we assume that every user-defined predicate p has a selectivity (a user-defined real number between 0 and 1) s_p associated with it. Applying the user-defined predicate is assumed to reduce the size of the relation to rs_p. The above cost model is used uniformly for user-defined selections and join predicates. Thus, a user-defined predicate is characterized by selectivity as well as by a cost per tuple parameter. As in the case of built-in predicates, we make the assumption that each user-defined predicate is independent of every other condition in the `Where` clause. In this paper, we do not discuss how the cost per tuple parameter can be estimated (see discussion in Hellerstein [1995]). Naturally, for a user-defined predicate to be evaluated, the intermediate relation must contain the columns

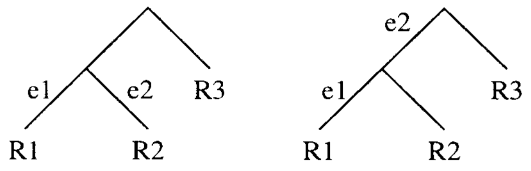


Fig. 1. Examples of unconstrained linear join trees.

over which the predicate is defined. For example, if the user-defined predicate is of the form $P(R1.c1, R2.c2)$, then the above predicate may not be evaluated until the join between $R1$ and $R2$ is taken.

Since the conditions using built-in predicates are assumed to have zero cost, all such predicates are evaluated at the earliest. However, since a user-defined predicate has an associated cost, we need to consider placement of user-defined predicates on the execution tree in a cost-based way. Consider any choice of execution space of join ordering. For any such choice, its natural extension when user-defined predicates are present is the execution space where we consider placing a user-defined predicate following any number of (including zero) joins, as long as the placed user-defined predicate is evaluable at the point of its placement. For example, a user-defined 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, a user-defined 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, we consider unconstrained placement of user-defined predicates. However, as in traditional optimization, we can continue to restrict the join ordering to be linear. In such a case, we refer to the execution space of *unconstrained linear join trees*. This is the same execution space that is studied in Hellerstein and Stonebraker [1993] and in Hellerstein [1994] and is formally defined below.

Definition 2.1 An *unconstrained linear join tree* is an execution tree where the sequence of operators form a linear sequence (tree) and a user-defined predicate may be placed anywhere in the sequence, provided it is evaluable at the point of evaluation.

Alternatively, we can consider a join execution space as consisting of bushy trees, and extend it in a similar way when user-defined predicates are present. In such a case, we refer to the execution space as *unconstrained bushy join trees* or *unconstrained join trees*.

Example 2.2 Figure 1 shows examples of unconstrained linear join trees for the execution plans of a query in which there are three relations and two user-defined selection predicates e_1 applicable on R_1 and e_2 applicable on R_2.

```
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. $S = \{R_j\} \cup S_j$ and $\{R_j\} \cap S_j = \emptyset$ do {
            p := joinPlan(optPlan($S_j$), $R_j$)
            if cost(p) < cost(bestPlan)
                bestPlan := p
        }
        optPlan($S$) := bestPlan
    }
}
return(optPlan($\{R_1, ..., R_n\}$))
```

Fig. 2. System R algorithm for linear join trees.

3. PREVIOUS APPROACHES

In this section, we review the basics of the System R optimization algorithm. We also discuss two approaches that were proposed for optimizing queries with user-defined predicates. However, since the results presented in this paper contain our results in Chaudhuri and Shim [1996], we do not discuss the latter paper in this section.

3.1 System R Dynamic Programming Algorithm

Figure 2 (adopted from Ganguly et al. [1992]) illustrates the System R dynamic programming algorithm that finds an optimal plan in the space of linear (left-deep) join trees [Selinger et al. 1979]. The input for this algorithm is a select-project-join (SPJ) query on relations $R_1, \ldots, R_n$. The 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. The cost function assigns a real number to any given plan in the execution space that we have chosen and satisfies the *principle of optimality* [Cormen et al. 1990]. The optimization algorithm chooses a plan of least cost from the execution space.

The algorithm proceeds by building optimal execution plans for increasingly larger subsets of the set of all relations in the join. In order to build an optimal plan for a set S of $(i + 1)$ relations, the optimal plan for each subset of S, consisting of i relations is extended (by invoking *joinplan* in Figure 2), and the cheapest of the extended plans is chosen. Such an approach guarantees the optimal, since the optimization problem satisfies the principle of optimality, i.e., an optimal plan for a set of relations must be an extension of an optimal plan for some subset of the set. Optimal plans for subsets are stored in the optPlan() array and are reused (rather than recomputed).

The algorithm does not expose two important details of the System R optimization algorithm. First, all selection conditions and secondary join

```
procedure DP_Algorithm_Bushy:
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 $S_1, S_2$ s.t. $S = S_1 \cup S_2$, $S_1 \neq \emptyset$, $S_2 \neq \emptyset$ and $S_1 \cap S_2 = \emptyset$. do
                p := joinPlan(optPlan($S_1$), optPlan($S_2$))
            if cost(p) < cost(bestPlan)
                bestPlan := p
        }
        optPlan($S$) := bestPlan
    }
}
return($optPlan(\{R_1, ..., R_n\})$)
```

Fig. 3. System R algorithm for bushy join trees.

predicates are evaluated as early as possible. Thus, 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. Although P' is a cheaper plan than P, the sort-order of P may be used to reduce the cost of a plan that extends P. Such an extension 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 Figure 2, one for each distinct such order, termed *interesting order* [Selinger et al. 1979]. An upper bound on the number of optimal linear join subplans stored for a query with a join of n tables is 2^n (the number of subsets of n tables) times the number of interesting orders. The enumeration complexity of the algorithm is $O(n2^{n-1})$.

The optimization algorithm for the space of bushy join trees is similar to the algorithm in Figure 2, except that the both inputs of a join operator can be an intermediate result. The dynamic programming algorithm in the space of bushy join trees is shown in Figure 3. An upper bound on the number of optimal subplans that must be stored for a query with joins among n tables is 2^n times the number of interesting orders. The corresponding enumeration complexity is $O(3^n)$.

3.2 LDL Approach

In this approach, a user-defined predicate is treated as a relation from the point of view of optimization. This approach was first used in the LDL project at MCC [Chimenti et al. 1989] and subsequently at the Papyrus project at HP Laboratories [Chaudhuri and Shim 1993]. 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.

This approach suffers from two drawbacks, both stemming from the problem of over-generalizing and viewing an expensive predicate as a relation. First, 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 execution trees (see Section 2 for the definition). In particular, the algorithm fails to consider plans that evaluate expensive predicates on both operands of a join prior to taking the join [Hellerstein 1994]. For example, consider a SPJ query with a join between the relations R_1 and R_2, having user-defined predicates e_1 and e_2 defined on R_1 and R_2 respectively. 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.[2] Thus, the LDL algorithm may produce plans that are significantly worse than plans produced by even the traditional optimization algorithm where all selections are evaluated as early as possible. Furthermore, the optimization algorithm is exponential not only in the number of relations but also in the number of expensive predicates. Let us look at 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 2^{n+k} optimal subplans. In other words, the cost of optimizing a query with n relations and k expensive predicates is as high as that of optimizing (n+ k) relations. As we will show, in many cases we can exploit the properties of join methods to achieve an algorithm that is polynomial in the number of user-defined predicates (when the number of relations is bounded).

3.3 Predicate Migration

This approach to optimizing queries with user-defined predicates has two components. First, we discuss the *predicate migration* algorithm, which given a linear join tree, chooses a way of interleaving the join and the user-defined predicates. Next, we describe how the predicate migration algorithm is integrated with a System R style optimizer. For additional details, we refer the reader to Hellerstein and Stonebraker [1993] and Hellerstein [1994].

The predicate migration algorithm takes as input a join tree, annotated with a given join method, for each join node and an access method for every scan node, and a set of expensive predicates. The algorithm places the user-defined predicates in their "optimal" position relative to the join nodes

[2]Such a plan will only be considered if the enumeration is extended to the space of bushy joins, which is significantly more expensive than the linear join space.

(see the following discussion about the shortcomings). 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 predicates in addition to assigning ranks for expensive predicates. The notion of rank has been studied previously in Monma and Sidney [1979] and Krishnamurthy et al. [1986]. 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 Hellerstein and Stonebraker [1993] that convergence occurs in a polynomial number of steps in the number of joins and user-defined predicates.

We now discuss how the algorithm is integrated with a 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 Q be the query $\sigma_e(R) \bowtie S \bowtie T$. Let P be the optimal subplan of $\sigma_e(R \bowtie S)$ and P' be the optimal subplan for $\sigma_e(R) \bowtie S$. If $cost(P) < cost(P')$, then the algorithm continues with the next step in the traditional join enumeration. In effect, application of the user-defined predicate is deferred and the plan P' is pruned without compromising the optimal. In contrast, if the plan for P' is cheaper, then the predicate migration algorithm needs to further extend P as well as P'. Note that it is *not* possible to discard plan P because doing so may compromise the optimal plan. For example, it is possible that although P' is cheaper, the optimal placement of e in the query succeeds the join among R, S, and T, i.e., the plan $\sigma_e(P \bowtie T)$ is optimal. In the predicate migration approach, once such a plan P' is found to be cheaper, the step of enumerating plans that extend P' is treated separately from the dynamic programming-based optimization algorithm. In particular, the predicate migration algorithm marked such a plan P' as *unprunable*. In the remaining steps of the dynamic programming algorithm, the unprunable plans are ignored. After the dynamic programming algorithm terminates, each such unprunable plan is extended through *exhaustive enumeration*, i.e., all possible ways of extending each such unprunable plan are considered. So completion of an unprunable plan is an expensive operation. In contrast, in our approach, it is not necessary to treat unprunable plans separately, they can be integrated with the dynamic programming algorithm in a seamless fashion.

The 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 while restricting the space of join ordering to linear join trees. Next, the algorithm is polynomial in the number of user-defined predicates. Unfortunately, this approach to optimization has serious drawbacks that limit its applicability. First, the algorithm *cannot guarantee* an optimal plan because it uses a heuristic to estimate the *rank* of join predicates that influence the choice of the plan. This is because using the predicate migration algorithm may force estimations of cardinality of relations used in determining the rank to be inaccurate. The migration algorithm 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 depend on whether the user-defined predicates have been evaluated! The predicate migration algorithm sidesteps this cyclic dependency by estimating the results of the joins, assuming that all user-defined predicates have been pushed down. This ad-hoc assumption sacrifices the guarantee of optimality (see Section 5.3 in Hellerstein [1994] for a detailed discussion). Next, since the predicate migration algorithm does not satisfy the principle of dynamic programming, it makes it difficult to do integration with a System R style optimization algorithm. In some cases the algorithm may require exhaustive enumeration. As an example, consider a query that has n relations and a *single* user-defined selection 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 $\sigma_e(R_1) \bowtie R_i$ $(i \neq 1)$ are 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, the optimization process may require *exhaustive enumeration* of the join space.

4. NAIVE OPTIMIZATION ALGORITHM

Our discussion in 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 in this section are adaptable for other join execution spaces (e.g., bushy joins) as well. Thus, our algorithm addresses the shortcomings of the predicate migration algorithm without sacrificing the benefit of considering the execution space of unconstrained linear join trees.

For notational convenience, we indicate ordering of the operators in a plan by nested algebraic expressions. For example, $(\sigma_e(R_1) \bowtie R_2) \bowtie \sigma_{e'}(R_3)$ designate 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 was reduced by application of a selection condition e'. Note that R_1, R_2, and R_3 need not necessarily be base tables but may themselves be query expressions. In describing the rest of this section, we assume that no traditional interesting orders are present; this assumption is for ease of exposition only.

4.1 Key Observations

The following two principles are key to our enumeration algorithm:

—*Equivalent plan pruning*. The strength of the traditional optimization algorithm for join enumeration is in its ability to compare the costs of different plans that represent the same subexpression but are evaluated in different orders. Since selection and join operations may be commuted, we can 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 subexpressions of the given query 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*. Consider a conjunction of a set of expensive selection predicates applied on a relation, as in the query $Q = \sigma_{e_1 \wedge .. \wedge e_n}(R)$ where $e_1, ..e_n$ are the expensive predicates. 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* with our assumed cost model for user-defined predicates. Furthermore, the ordering of the selections does *not* depend on the size of the relation on which they apply. The problem of selection ordering is addressed in Hellerstein and Stonebraker [1993] (cf., Krishnamurthy et al. [1986]; Monma and Sidney [1979; and Whang and Krishnamurthy 1990]). 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 $e = e_1 \wedge \ldots \wedge e_n$. 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. Thus evaluation of e' should precede that of e. The above technique of selection ordering can be extended to broader classes of boolean expressions. In particular, when the selection conditions form a pure disjunction, a dual of Theorem 4.1 applies where the predicates are ordered by ascending values of c/s. On the other hand, the selection ordering problem is intractable when the selection conditions are arbitrarily complex. However, efficient heuristics for the general case were proposed in Kemper et al. [1992].

The ability to order predicates using ranks (called "rank-order"-ing in this paper) is a key property that is exploited in this work. Although we leverage this property in this section, note that Theorem 4.1 does not allow us to order user-defined predicates across joins. However, in Section 5, we show how the relative order of ranks among predicates can be exploited even across joins that lead to the optimization algorithm with *complete* rank-ordering.

4.2 Tags: Exploiting Properties for Plan Representation

The *properties* concept that extends the idea of *interesting orders* [Selinger et al. 1979] was addressed in Graefe and Dewitt [1987]; Lee et al. [1988]; and Graefe and McKenna [1993]. Properties help describe the intermediate result of a plan. Properties include tables, projected columns, sort-order of tuples, and the set of predicates applied. The query optimizer keeps a single cheapest plan for each distinct set of properties during optimization. However, traditional optimization algorithms always process selection predicates as early as possible. In order to address the problem of optimizing queries with user-defined predicates, we exploit this well-understood notion of properties by incorporating the information on application of each user-defined predicate as part of the properties of a plan. We keep multiple plans that represent the join of the same set of relations but differ in the sets of user-defined predicates that have been evaluated. Thus, with every plan, we also record as a property the set of yet-to-be evaluated user-defined predicates applicable to the plan, i.e., it records the complement of the (applicable set of) predicates that were evaluated in the plan. In the rest of this paper, we refer to the above property as the *tag* of a plan. Of course, the representation of the tag can be combined with the representation of other properties, e.g., interesting orders. The following definition states formally how we associate a tag with an unconstrained join tree.

Definition 4.2 Let T be an unconstrained linear join tree that consists of a join among a set $\mathcal{R}$ of relations and evaluation of a set $\mathcal{U} \subseteq \mathcal{S}$ of user-defined predicates where $\mathcal{S}$ is the set of all user-defined predicates in the query that can be evaluated over the subexpression of the query that consists of the join among relations in $\mathcal{R}$. Then, the *tag* associated with the tree T is the *ordered set* of predicates $\mathcal{S} - \mathcal{U}$, sorted by rank order.

Example 4.3 Figure 4 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, P_3, and P_4 are possible plans for $R_1 \bowtie R_2$ (each with differing tags). The plans from P_6 to P_{15} are for $R_1 \bowtie R_2 \bowtie R_3$. The tags for P_6 and P_7 are $<>$ and $<e_2>$, respectively.

Two unconstrained join trees represent the same expression iff they have the same tag and consist of the join of the same set of relations. Notice that whenever two plans represent the join of the same set of relations and agree on the tags, they can be compared and pruned.

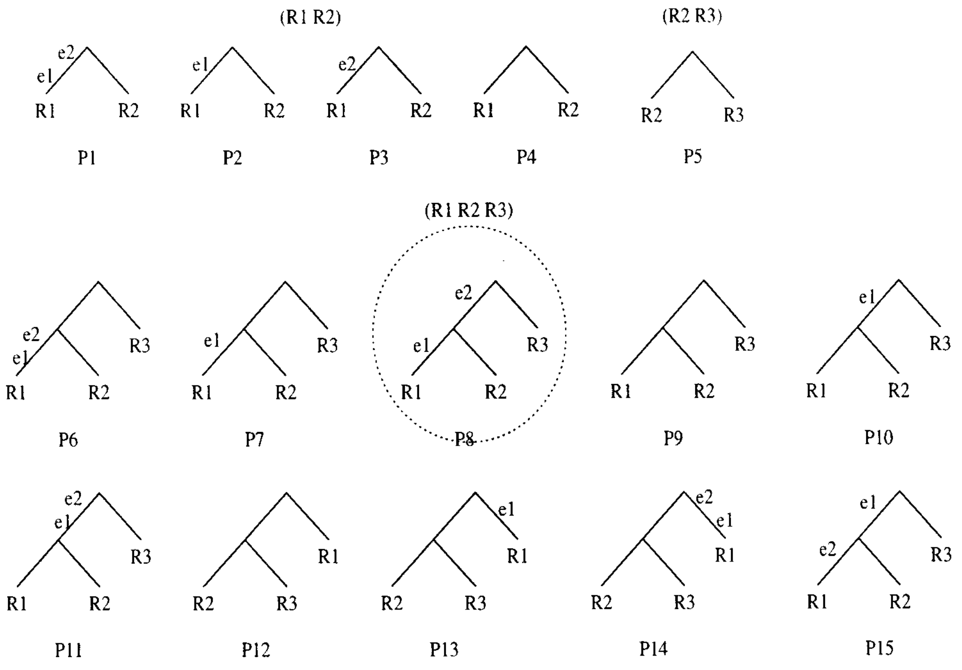


Fig. 4. Search space of naive optimization algorithm.

4.3 Algorithm

In this section we present the *naive optimization algorithm* that is widely applicable and makes no assumptions about the nature of the cost formulas, except that it satisfies the principle of optimality. The algorithm exploits the notion of tags, and thus demonstrates that the traditional notion of properties can be applied to the problem of optimizing queries with user-defined predicates to produce a general solution, independent of any assumptions on a cost model.

The equivalent plan-pruning rule allows us to compare two plans that represent the same expression. This observation enables the use of dynamic programming and integrates well with the System R algorithm. In addition, the naive optimization algorithm uses the result of Theorem 4.1, which tells us that ordering user-defined predicates on a relation is constrained by rank order (but not across joins). This optimization algorithm also introduces the key idea of exploiting tags. In Section 5, we present a much improved algorithm for which these foundational ideas are crucial. However, the algorithm in that section exploits these key ideas more aggressively and takes full advantage of rank ordering even across joins.

The naive optimization algorithm for the space of unconstrained linear join trees is shown in Figure 5. In determining the access methods and choice of join methods, the algorithm behaves exactly like the traditional

```
procedure Opt-Naive:
for i := 2 to $n$ do {
    for all $S \subseteq \{R_1, ..., R_n\}$ s.t. $||S|| = $ i do {
        initialize an array of bestPlan to a dummy plan with infinite cost
        for all $R_j, S_j$ s.t. $S = \{R_j\} \cup S_j$ and $\{R_j\} \cap S_j \neq \emptyset$ do {
            for all $P$ stored in plan table for $S_j^t$ with all different tag $t$ do {
                for all $u$ s.t. $u$ is a subset of predicates in the tag for $P$ do {
                    for all $v$ s.t. $v$ is a subset of user-defined selection predicates on $R_j$ do {
                        p := exjoinPlan($P$, $R_j$,u,v)
                        if cost(p) < cost(bestPlan[tag(p)])
                            bestPlan[tag(p)] := p
                    }
                }
            }
        }
        copy plans in bestPlan to optPlan($S$)
    }
}
finalPlan := a dummy plan with infinite cost
for all plan p $\in$ optPlan($\{R_1, ..., R_n\}$) do {
    if complete_cost(p) < cost(finalPlan)
        finalPlan := completed plan of p
}
return (finalPlan)
```

Fig. 5. Naive optimization algorithm for linear join trees.

algorithm in Figure 2. The *bestPlan* data structure stores all plans that need to be retained for the optimizer's future steps. For every subset of relation S_j (in Figure 2) and for each distinct tag, an optimal plan may need to be stored. For each optimal plan S_j^t with a tag t, instead of JoinPlan (as in Figure 2), *exjoinPlan* is invoked to extend the optimal plan S_j^t. Observe that *exjoinPlan* also takes as arguments two sets of user-defined predicates to be applied to the plan P for S_j^t and relation R_j, respectively (in rank order), prior to taking the join.

Let us assume that tag t has r applicable user-defined predicates and the base relation R_j has a set w of s applicable user-defined selections. The function *exjoinPlan* is invoked for each distinct subset of $(r + s)$ predicates (i.e., 2^{r+s} invocations). In the next section we show how, with broad assumptions on join costs, we can sharply reduce the overhead of so many invocations. The choices of u and v in the algorithm uniquely determine the tag for the plan p in Figure 5 as the set (sorted in rank order) $(t - u) \cup (w - v) \cup l$, where l is the set of additional user-defined predicates that are now evaluable due to the join between S_j^t and R_j. Note that Theorem 4.1 assumes that when a sequence of user-defined predicates is applied without any intervening joins (e.g., the set of predicates u or v), the predicates can be applied in the rank order without sacrificing optimality.

In our algorithm described in Figure 5, we use the array of bestPlans indexed by tag t, and the function *tag*(p) returns the tag of plan p. Plan p

is compared against plans over the *same set of relations* that have already been stored and that have the same tag. If p is more expensive than the plan in the *bestPlan* with the same tag, then plan p is pruned and the iteration procedes to the next (u, v) combination. Otherwise, plan p is added to *bestPlan*.

At the end of the final join, we consider all plans over relations $\{R_1, \ldots, R_n\}$. Some of these plans may need to be completed by adding the step to evaluate the remainder of the predicates. The function *completecost*(p) computes this completion cost of plan p. Finally, the cheapest among the set of completed plans is chosen.

Since we store the best plan for every subset of user-defined predicates per each distinct set of relations, the total number of stored plans per each distinct set of relations increases to 2^k, and therefore the number of plans that need to be stored increases to 2^{k+n}, where k is the number of user-defined predicates. Consider Figure 4 for Example 2. Since the tag for P_6 and P_7 are $<>$ and $<e_2>$ respectively, we distinguish between P_6 and P_7, but keep a single plan among P_6, P_8, P_{11}, P_{14}, and P_{15}.

The naive optimization algorithm is very general and makes no assumptions on the cost model. So it has wide applicability. However, the complexity of this algorithm is exponential in the number of user-defined predicates as well as in the number of relations in the query, as shown in Section 6. In the next section, we show how rank ordering among user-defined predicates may be exploited in conjunction with realistic assumptions on a cost model to develop an algorithm that is polynomial in the number of user-defined predicates (for a given number of relations) in a query.

5. OPTIMIZATION ALGORITHMS WITH COMPLETE RANK-ORDERING

The complexity of enumeration by the naive optimization algorithm is exponential in the number of user-defined predicates. In the algorithm, rank ordering (Theorem 4.1) is only used to order the execution of predicates that were applied prior to application of any other operators. It turns out that if we could exploit rank ordering irrespective of whether predicates are separated by join nodes, then we can make the optimization algorithm polynomial in the number of user-defined predicates (for a given number of relations). In this section, we show that as long as the join implementations follow the assumption of being a *regular join* (defined below), we can exploit rank ordering even if user-defined predicates are separated by join nodes. We provide a definition of regular join methods in Section 5.1 and justify that this definition reflects the cost of popular join methods in practice. In Section 5.2, we show that when all join operators follow the assumption of being regular joins, we can indeed restrict our enumeration to execution trees where *all* predicates are ordered by rank order. This key result immediately leads to a new algorithm that is polynomial in the number of user-defined predicates, which we present in Section 5.4. Indeed, the results in this section are robust enough to extend to execution trees containing other operators (in addition to join), as long as the dependence of operators on input relation size(s) follows Definition 5.1.

5.1 Regular Join Methods

In addition to the magnitude of relation sizes, the cost of a join method depends on a variety of system parameters such as prefetching techniques, specific I/O algorithms, and existing indexes. However, for a *given* set of system parameters, the following definition captures the first order dependence of join methods on input relations for a large class of implementations:

Definition 5.1 A join method is called *regular* if the cost $f(R_1, R_2)$ of joining two relations of sizes R_1 and R_2 depends on the sizes of the relations as follows: $f(R_1, R_2) = a + bR_1 + cR_2 + dR_1R_2$ where the constants a, b, c, d are independent of the sizes of relations R_1 and R_2.

An important subtlety in the above definition is that the value of factors a, b, c, and d may depend on system parameters (e.g., available indexes), rather than being global constants. In particular, each join node in the execution tree may have a different value of these parameters. This lends a significant generality to the above definition.

We now explain why a large number of join methods satisfy the constraints on join methods that are assumed by the above definition of regular joins. Furthermore, in the remainder of this paper, we represent the cost of joins in terms of I/O cost only. Although it is an approximation, this cost is predominantly used in many commercial optimizers. Furthermore, many components of even the CPU costs of joins fits the form of regular join. Below, we consider I/O costs for nested-loop, merge-scan, and hash joins [Shapiro 1986] and demonstrate that they follow the assumption of a regular join. The detailed cost formulas for the three common join methods can be found in Shim [1993]. Note that the cost of writing the result of the join to disk depends only on the size of the result of the join, and has the form $R_1 {*} R_2 {*} S_J$, where S_J is join selectivity. Since this cost element fits the form of regular joins, we ignore it in the rest of this discussion.

For a nested-loop join without any index being used, the cost formula becomes $f(R_1, R_2) = R_1W_1/P + R_2W_2/P$ when the buffer size available can hold the inner relation R_2. Here, W_i is the tuple width of relation R_i and P is the page size in bytes. Thus, R_iW_i/P is the number of pages occupied by relation R_i. If the available buffer is not enough, the join cost becomes $f(R_1, R_2) = R_1W_1/P + R_1(R_2W_2/P)$. With block-nested join methods, the cost becomes $f(R_1, R_2) = R_1W_1/P + \lceil (R_1W_1/P)/(B - 1) \rceil (R_2W_2/P)$, where B is the available buffer size. Thus, join cost formulas for nested loop joins without an indexed inner relation are consistent with Definition 5.1. For a nested-loop join with an indexed inner relation, the cost of scanning the inner relation per tuple of outer relation is replaced by the cost of probing the index, which is typically a constant number of page accesses. Thus, the cost formula in such a case is $f(R_1, R_2) = R_1W_1/P + R_1I$, where I is the cost of probing the index. Observe that the above is a degenerate form of a regular join cost formula.

For sort-merge and hash joins, the cost formula given in Shapiro [1986] fits our form of join cost, assuming large buffers. For example, the simple estimate of sorting costs for relation R_i in terms of I/O is two times the cost of scanning the base table, i.e., $2R_iW_i/P$. The merge cost is equivalent to scanning both relations, resulting in the estimate $R_1W_1/P + R_2W_2/P$, which is consistent with the form of regular join cost formulas. Similar comments apply to the cost formulas for hash join [Shapiro 1986]. With decreasing costs for memory, the large buffer assumption is often a realistic operating assumption. Finally, note that the cost formulas for regular joins generalize the assumption of the linear cost model in Hellerstein and Stonebraker [1993] and Hellerstein [1994]. The latter corresponds to a special case of Definition 5.1 where the coefficient $d = 0$.

5.2 Predicate Ordering on a Join Tree

The selection ordering rule (Theorem 4.1) applies when all predicates are applied on a relation without any intervening join nodes. In this section, we present a powerful generalization of the selection ordering rule. We show that for a given join tree, the placement of the user-defined-predicates cannot violate the rank order, i.e., for every such execution tree, there is another execution tree that has less or equal cost and where the sequence of predicate applications follows rank ordering. We refer to such execution trees as *rank-ordered*, as defined below:

Definition 5.2 The user-defined predicates in an unconstrained execution tree τ are *rank-ordered* if for any two user-defined predicates p and p' in τ such that $rank(p) < rank(p')$, either p precedes p' in the tree τ, or p is not evaluable in the tree τ' obtained by exchanging the positions of p and p' in τ.

Example 5.3 Consider Figure 4, in which there are 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. Assume that we have $rank(e_1) < rank(e_2)$. The user-defined predicates e_1 and e_2 in P_8 are rank-ordered. However, the two predicates in P_{15} are not rank-ordered.

The following theorem, which is the central result of this section, asserts that it is sufficient to restrict ourselves to the plans that correspond to rank-ordered trees. We prove the result by contradiction. We begin by assuming that indeed there is an execution tree that is not rank-ordered (let us call it the "spoiler"), but has a cost strictly lower than all rank-ordered trees. We pick a set of rank-ordered trees that are equivalent and syntactically "similar." We exploit the syntactic similarity to derive a constraint among the costs of these rank-ordered trees and the spoiler tree.

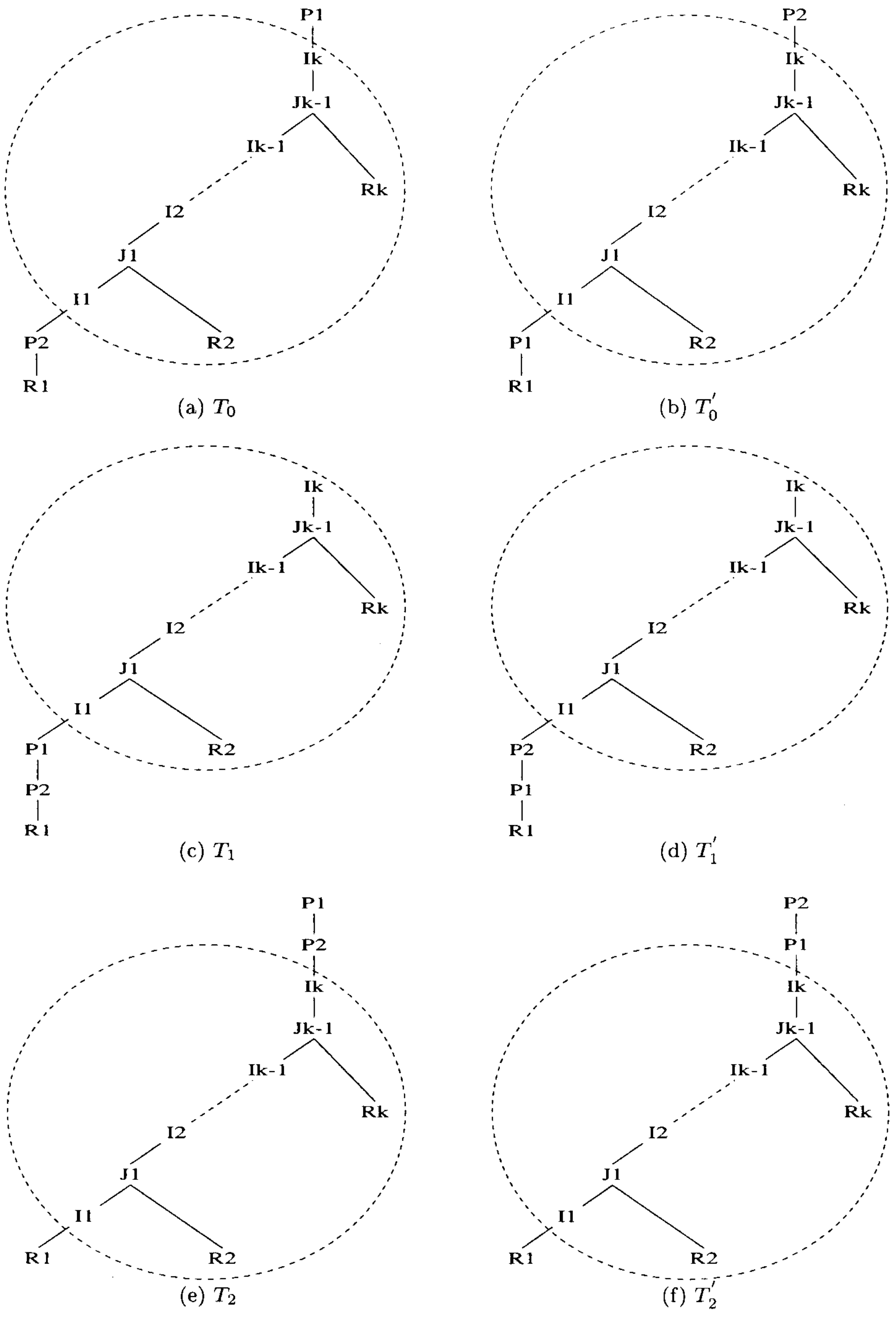


Fig. 6. Plans used in correctness proof.

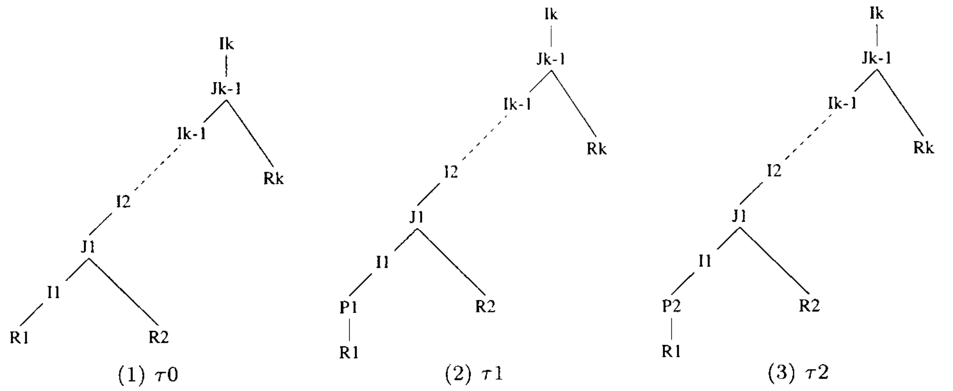


Fig. 7. Definitions of $\tau0$, $\tau1$ and $\tau2$.

Then we show that because join operators are assumed to satisfy the constraint of being regular joins, the above constraint cannot be satisfied. We now present the formal proof.

THEOREM 5.4 *If T is any unconstrained execution tree using only regular join methods, then there must exist an equivalent unconstrained execution tree T' such that $cost(T') \leq cost(T)$ and the user-defined predicates in T' are rank-ordered.*

PROOF. We prove the result by contradiction. For a contradiction to occur, the number of user-defined predicates in the execution tree must be at least 2. Furthermore, it must be the case that the cost of all equivalent rank-ordered trees are strictly higher. In other words:

(1) There is an unconstrained execution tree T_0 and two user-defined predicates p_1 and p_2 where $rank(p_1) < rank(p_2)$, and p_2 precedes p_1.

(2) The user-defined predicate p_1 is evaluable in the tree T'_0 obtained by exchanging the relative positions of p_1 and p_2 in T_0.

(3) In T_0, there must be one or more joins separating applications of p_1 and p_2.

(4) The cost of T_0 is strictly lower than the cost of any equivalent unconstrained rank-ordered execution tree.

Observe that condition (3) must be true, since if p_1 and p_2 were applied without any intervening joins, the selection ordering rule (Theorem 4.1) implies that selections must be in order of rank. Thus, an unconstrained violation of rank order can occur only if the predicates p_1 and p_2 are separated by one or more joins.

We now exploit condition (3) to derive a constraint on the relative costs of alternative execution trees. Specifically, we consider four additional execu-

Table I. Definition of Parameters for Correctness Proof

c_i	Cost of predicate P_i per tuple
s_i	Selectivity of predicate P_i
r_i	Number of tuples in relation R_i
$C_{\tau i}$	Cost of execution tree τi
$N_{\tau i}$	Size of the output of execution tree τi

tion trees that are equivalent. Figure 6 shows these execution trees, including T_0 and T'_0. The execution tree T_1 corresponds to positioning p_1 just after p_2 in the execution tree T_0. Let T'_1 be the same tree as T_1, except that the positions of p_1 and p_2 are exchanged. Observe that from Theorem 4.1 it follows that $cost(T') \leq cost(T_1)$. The execution tree T_2 is a tree obtained from T_0 by moving p_2 to a position immediately preceding p_1. We obtained T'_2 by exchanging the positions of p_1 and p_2 in T_2. Once again, by using Theorem 4.1, it follows that $cost(T'_2) \leq cost(T_2)$.

The encircled query expression in Figure 6 with a single input relation identifies a common subexpression among execution trees. Let us refer to this subexpression by $\tau(R)$, where the parameter R refers to the input relation of τ. Three instances of this common expression (with different input relations R) are depicted in Figure 7. The following identities are evident:

$$\tau 0 = \tau(R_1)$$

$$\tau 1 = \tau(\sigma_{p_1}(R_1))$$

$$\tau 2 = \tau(\sigma_{p_2}(R_1))$$

In the rest of this proof we also use $\tau 12$ to denote $\tau(\sigma_{p_1}(\sigma_{p_2}R_1))$ and $\tau 21$ to denote $\tau(\sigma_{p_2}(\sigma_{p_1}R_1))$. Note that relation sizes for $\tau 0$, $\tau 1$, $\tau 2$, and $\tau 12$ bear the obvious relationship. We can relate the execution trees in Figure 7 with those in Figure 6. In particular, the following correspondences hold:

$$T_0 = \sigma_{p_1}(\tau 2)$$

$$T' = \sigma_{p_2}(\tau 1)$$

$$T_1 = \tau 12$$

$$T'_1 = \tau 21$$

$$T_2 = \sigma_{p_1}(\sigma_{p_2}(\tau 0))$$

$$T'_2 = \sigma_{p_2}(\sigma_{p_1}(\tau 0))$$

We now estimate the costs of the execution trees using the parameters defined in Table I. We use notation $C_{\tau i}$ and $N_{\tau i}$ to designate the cost and size of the output relation resulting from the execution trees τ_i.

Since the cost model is consistent (see Section 2), it follows that $N_{\tau i} = s_i N_{\tau 0}$ (for $i = 1, 2$). Furthermore, $N_{12} = s_1 s_2 N_{\tau 0}$.

Observe that execution trees T'_0, T'_1, and T'_2 are rank-ordered. By our assumption, it must be that the cost of T_0 is strictly less than the cost of each of these trees. We now use these relationships to derive a contradiction.

By definition, $cost(T_0) = c_2 r_1 + C_{\tau 2} + c_1 N_{\tau 2}$. However, $N_{\tau 2} = s_2 N_{\tau 0}$. Hence,

$$cost(T_0) = c_2 r_1 + C_{\tau 2} + c_1 s_2 N_{\tau 0} \tag{1}$$

By our assumption, $cost(T_0) < cost(T'_1) \leq cost(T_1)$. Note that $cost(T_1) = c_2 r_1 + c_1 s_2 r_1 + C_{\tau 12}$. Therefore, using Eq. (1), we conclude that

$$N_{\tau 0} < r_1 - (C_{\tau 2} - C_{\tau 12})/s_2 c_1 \tag{2}$$

Next, note that $cost(T_2) > cost(T_0)$. We now estimate $cost(T_2)$:

$$cost(T_2) = C_{\tau 0} + c_2 N_{\tau 0} + c_1 s_2 N_{\tau 0} \tag{3}$$

Using Eqs. (1) and (3), we conclude that

$$N_{\tau 0} > r_1 - (C_{\tau 0} - C_{\tau 2})/c_2 \tag{4}$$

It follows from Eqs (2) and (4) that

$$(C_{\tau 2} - C_{\tau 12})/s_2 c_1 < (C_{\tau 0} - C_{\tau 2})/c_2 \tag{5}$$

The rest of this proof shows that when the joins follow the constraint of regular joins, Eq. (5) cannot be true. The cost of evaluating a tree is the sum of costs of the joins and the costs of applying user-defined predicates. We can represent that the cost of an expression $\tau(R)$ to be the sum of the following three components:

—Cost of evaluating expensive predicates in the expression $\tau(R)$: Denote such costs by $P_u(R)$ and the sum of all such costs as $\Sigma_u P_u(R)$. For our assumed cost model for user-defined predicates, $P_u(R) = c_u h_u(R)$, where the constant c_u is the cost of applying the user-defined predicate with per tuple cost c_u, where $h_u(R)$ is the size of the relation preceding application of the u-th application of a user-defined predicate.

—Cost of evaluating join nodes in the expression $\tau(R)$ that are ancestors of R: Denote this cost by $J_v(R)$ and the sum of all such costs as $\Sigma_v J_v(R)$. Note that since join is a binary operation, at most one input can be dependent on the input relation R for each join. Furthermore, in our assumed cost model, all the join methods are regular. Therefore, we assume that $J_v(R) = a_v + b_v m_v(R) + c_v q_v + d_v m_v(R) q_v$, where we assumed that $m_v(R)$ and q_v are the sizes of the input relations to the v-th join and only $m_v(R)$ depends on the input relation R. Thus, so far as reflecting the dependence of R on the cost of the join, we can simplify $J_v(R) = f_v m_v(R) + l_v$.

—Cost of evaluating all other operators that are not affected by the input relation R: We denote this cost by I.

We apply the above costing framework to each of the three instances of τ, i.e., $\tau 0$, $\tau 1$, and $\tau 2$ to establish a relationship between $C_{\tau 2}$ and $C_{\tau 0}$. We begin by rewriting the cost components for $C_{\tau 2}$ and $C_{\tau 0}$ and distinguishing the respective cost components by superscripts:

$$C_{\tau 0} = \sum_u c_u h_u^0 + \sum_v (f_v m_v^0 + l_v) + I$$

$$C_{\tau 1} = \sum_u c_u h_u^1 + \sum_v (f_v m_v^1 + l_v) + I$$

$$C_{\tau 2} = \sum_u c_u h_u^2 + \sum_v (f_v m_v^2 + l_v) + I$$

The only difference between $\tau 2$ and $\tau 0$ is in application of the predicate p_2 on R_1. In order to estimate m_v^2, we note that, because the cost formula is consistent, we can assume that size m_v^2 is the same whether the predicate p_2 is applied before any operator in τ is applied or the predicate p_2 is applied just before the v-th join is executed. By using the latter method, we note that $m_v^2 = s_2 m_v^0$. Likewise, $P_u^2 = s_2 P_u^0$. We can extend the same approach to compute $C_{\tau 12}$ and arrive at the following equalities: $m_v^{12} = s_1 s_2 m_v^0$, $P_u^{12} = s_1 s_2 P_u^0$. From the above, we conclude that: $C_{\tau 0} - C_{\tau 2} = (1 - s_2) C'_0$, $C_{\tau 2} - C_{\tau 12} = s_2(1 - s_1) C'_0$ where $C'_0 = \Sigma_u P_u^0 + \Sigma_v f_v m_v^0$. Therefore, from Eq. (5), we conclude that $(1 - s_1)/c_1 < (1 - s_2)/c_2$. In other words, $rank(p_1) > rank(p_2)$, which violates our assumption. □

5.3 Comments on the Generality of the Result

The above proof is extremely robust; we explain by discussing the following generalizations.

Execution space: Observe that the proof makes no assumption about the structure of the execution space. Our proof technique simply exploits the fact that for a violation of the theorem to occur, there must exist a nonempty sequence of join nodes $(J_1, \ldots, J_{k-1})$ in between the application of two predicates (p_2 and p_1). This observation is independent of any assumptions on whether the space of unconstrained join trees is linear or not. So the result is uniformly applicable for linear as well as bushy join trees.

Expensive join predicates: Theorem 5.4 makes no assumption on the structure of predicates, and so is uniformly applicable for user-defined predicates. This has two consequences. First, our theorem is applicable to user-defined predicates that may involve columns of *multiple* tables. Specifically, note that we do not assume R_1 in T_0 (and in other execution trees) is a *base* relation. Rather, R_1 represents the relation resulting from the execution of the subtree of which p_2 is the parent node. When p_2 is an expensive predicate over columns of multiple relations, the subtree below p_2 must include the join among the needed relations for T_0 to be an admissible execution tree. Next, our cost model for a user-defined predicate is based on a *constant per-tuple* cost. The internal structure of the user-defined predicate is irrelevant as long as the above assumption of a constant per-tuple cost is appropriate. For example, as argued in Hellerstein [1995], complex subqueries fit the above cost model in many cases. Another example is when user-defined predicates have disjunctions among them. In such cases, it may be possible to associate a per tuple cost with each conjunct when the boolean expression is translated into a conjunctive normal form.

Operators other than join: The unconstrained join trees consist of only user-defined predicates and join trees as the only operators in the execution tree. However, the above result is robust for *any* binary or unary operators that satisfy the syntactic condition specified in the definition of regular joins. This is because the proof only exploits the specific syntactic dependence of each J_j on the sizes of its input relations.

Relationship to results in Hellerstein [1995]: As explained earlier, Hellerstein's approach in Hellerstein [1995] suffers from the *cyclic dependency* that compromises the correctness of his approach. His approach *requires* that we assign an *a priori* rank to each traditional equijoin predicate ($J_1, \ldots, J_{k-1}$), so that their rank can be compared to those of user-defined (selection or join) predicates. Unfortunately, computation of the rank of a traditional equijoin predicate depends on the sizes of its input relations, which in turn depends on which user-defined predicates are applied prior to computing the join. This cyclic dependency is successfully avoided in our approach. In Theorem 5.4 the ranks of *only* user-defined predicates need to be assigned, but the algorithm is still able to correctly order user-defined predicates with respect to traditional join predicates using a dynamic programming algorithm. The property that the joins follow the constraints of regular joins implies that the executions of user-defined predicates must follow the order of rank to ensure optimality. However, we do *not* need to assign any rank to the traditional equijoin predicates, and thus unlike Hellerstein [1995], we do not encounter cyclic dependency. Finally, our definition of a regular join strictly generalizes the linearity assumption for a join cost formula in Hellerstein [1995].

```
procedure Opt-Rank:
for i := 2 to $n$ do {
    for all $S \subseteq \{R_1, ..., R_n\}$ s.t. $||S|| =$ i do {
        initialize an array of bestPlan to a dummy plan with infinite cost
        for all $R_j, S_j$ s.t. $S = \{R_j\} \cup S_j$ and $\{R_j\} \not\in S_j$ do {
            for all $P$ stored in plan table for $S_j^t$ with all different tag $t$ do {
                $s$ := Number of evaluable predicates on $P$
                $r$ := Number of evaluable predicates on $R_j$
                for all u := 0 to $s$ do {
                    for all v := 0 to $r$ do {
                        p := extjoinPlan($P$, $R_j$,u,v)
                        if cost(p) < cost(bestPlan[tag(p)])
                            bestPlan[tag(p)] := p
                    }
                }
            }
        }
        copy plans in bestPlan to optPlan($S$)
    }
}
finalPlan := a dummy plan with infinite cost
for all plan p $\in$ optPlan($\{R_1, ..., R_n\}$) do {
    if complete_cost(p) < cost(finalPlan)
        finalPlan := completed plan of p
}
return (finalPlan)
```

Fig. 8. Optimization algorithm with rank-ordering for linear join trees.

5.4 Algorithm

This algorithm, which, from now on, we refer to as the *optimization algorithm with complete rank-ordering* is shown in Figure 8. It assumes that the joins follow the assumptions of regularity. When the join methods are regular, Theorem 5.4 enables us to further restrict the sequence in which the user-defined predicates may be applied and reduces the complexity of enumeration from exponential to polynomial in the number of user-defined predicates. In particular, the user-defined predicates must now be applied in rank order even if there are intervening joins. Therefore, 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 (ordered by rank), where $0 \leq u \leq r$ and $0 \leq v \leq s$. This is the only change in the algorithm from the naive optimization algorithm.

6. COMPLEXITY

In this paper we study the complexity of the algorithms in terms of two parameters. The *number of subplans to be stored* provides a measure of the space requirement for optimization. The *number of enumerations* is the other measure, and it reflects the time complexity of optimization algo-

Table II. Definition of Parameters for Complexity Proof

n	Total number of relations
g	Number of relations with user-defined selection predicates applicable
h	Number of pairs of relations having user-defined join predicates
k	Total number of user-defined predicates
w	Maximum of number of user-defined selections applicable per relation and number of user-defined join predicates per pair of relations
u	Sum of g and h

rithms. We define the latter parameter as counting the number of calls made to *extjoinplan*, which is the step that extends an existing subplan with one more join to create a plan for a larger subquery of the given query.

We now introduce the input parameters that we use for complexity analysis. We assume that there are n relations. A user-defined predicate can be either a selection or a join predicate. We assume that there are at most g relations ($g \leq n$) that have one or more user-defined selections defined on them, i.e., there are g *types* of selection predicates. Likewise, we can characterize the *type* of a user-defined join predicate by the two relations over which the predicate is defined. We assume there are h types of user-defined join predicates ($h < n^2$). We let u be the total number of types (i.e. $u = g + h$) of user-defined predicates and let k denote the total number of user-defined predicates. We assume w denotes the maximum number of user-defined predicates (either selection or join) of the same type. Table II summarizes these parameters.

Example 6.1 Consider a query that represents a join among four relations R_1, .., R_4, where each of R_1 and R_2 has two user-defined selections and there is a secondary user-defined join predicate between R_1 and R_2. Since the number of relations in the query is four and among them only two relations have user-defined predicates (i.e. R_1 and R_2), we have $n = 4$ and $g = 2$. Furthermore, $h = 1$, $w = 2$, $u = 3$ and $k = 5$.

6.1 Number of Subplans Stored

In this section we examine the number of subplans to be stored by the naive optimization and the optimization algorithm with complete rank-ordering .

THEOREM 6.2 *The total number of subplans that need to be stored by the naive optimization algorithm is no more than* 2^{n+k}.

PROOF. The naive optimization algorithm potentially creates a subplan for every subset of relations that are joined and every subset of user-defined predicates present in the query. Since there are n relations and there are k user-defined predicates, the total number of such plans is at most 2^{n+k}. □

In contrast, the optimization algorithm with complete rank ordering does not need to store subplans for each subset of user-defined predicates. We begin by defining an upper-bound on the number of distinct tags that may need to be considered for optimization with complete rank ordering.

LEMMA 6.3 *The number of distinct tags to be stored for a distinct set of relations in the plan table is no more than* $(1 + w)^u$.

PROOF. Given a distinct set of relations, we can partition user-defined predicates into u sets. There can be at most w user-defined predicates that belong to the same partition. The application of these predicates must follow the order of rank. Therefore, over each partition, there can be at most $(1 + w)$ possibilities. Since there are altogether u partitions, there can be at most $(1 + w)^u$ distinct tags altogether. □

THEOREM 6.4 *The total number of subplans that need to be stored by the optimization algorithm with complete rank ordering where a query has user-defined selection predicates but no user-defined join predicates is no more than* $2^n(1 + w/2)^g$.

PROOF. We store a distinct plan corresponding to a join of a distinct set of relations and a distinct set of user-defined selections. However, user-defined selections can apply only after the relations over which they apply have been joined. First, we determine the number of ways in which one or more of the g relations (with user-defined selections on them) can be joined and user-defined selections are applied on them. If we pick i of such relations, then they may be picked in $\binom{g}{i}$ ways. With each such join of i relations there can be $(1 + w)^i$ tags (by Lemma 6). Therefore, the number of ways in which we can pick one or more of g relations and user-defined selections on them is

$$\sum_{i=0}^{g} \binom{g}{i} (1 + w)^i = (2 + w)^g.$$

Next, we must consider all possible ways in which we can pick one or more of the relations from the set of remaining $(n - g)$ relations that have no user-defined selections on them. The latter can be achieved in

$$\sum_{i=0}^{n-g} \binom{n - g}{i} = 2^{n-g}$$

ways. Hence, the upper bound on the total number of plans that need to be stored is $(2 + w)^g 2^{n-g}$, which equals $(1 + w/2)^g 2^n$. □

Since $g \leq n$ and $w \leq k$, the above formula can be used to derive an upper-bound of $(2 + k)^n$ for queries that have user-defined selection predicates only. This upper bound is polynomial in k but exponential in n, the number of relations in the query (as in traditional join enumeration).

The above analysis also shows that the complexity is sensitive to the number of relations that have one or more user-defined predicates, as well as to the number of predicates that may apply to a single relation. The complexity grows exponentially with the number of relations over which user-defined selection predicates occur, since they increase the number of tags exponentially. In contrast, complexity grows only polynomially as we increase the number of user-defined predicates for a bounded number of relations in the query where they occur. For example, when all user-defined predicates are selections and apply to only one relation, then the number of plans that need to be stored is at most $(1 + k/2)2^n$. The next theorem presents the complexity of the optimization algorithm with complete rank ordering when a query has user-defined join predicates as well.

THEOREM 6.5 *The total number of subplans that need to be stored by the optimization algorithm with complete rank-ordering where a query has both user-defined selections and join predicates is no more than* $2^n(1 + w)^u$.

PROOF. The proof is very similar to that of Theorem 6.4. For a given set of relations, the number of distinct plans that differ on tags are at most $(1 + w)^u$. Thus, by replacing $(1 + w)^i$ with $(1 + w)^u$ in the corresponding expression in Theorem 6.4, we get the upper bound

$$(1 + w)^u \sum_{i=0}^{g} \binom{g}{i} = (1 + w)^u 2^n. \qquad \square$$

Since $g \leq n$, $u \leq g + g(g - 1)/2 \leq g^2$ for nonnegative integers of g and $w \leq k$. Therefore, the above formula can be used to derive an upper-bound of $(1 + k)^{n(n+1)/2}2^n \leq ((1 + k)^n 2)^n$. Hence, the upper-bound is a polynomial in k for a given n.

6.2 Number of Enumerations

The complexities of enumeration for left-deep join space and bushy join space by a traditional system R style optimizer are known to be $O(n2^{n-1})$ and $O(3^n)$, respectively, where n is the number of relations in the query. We now study the complexities of enumeration for the naive optimization algorithm and the optimization algorithm with complete rank-ordering for linear as well as bushy join trees. The proofs of theorems presented in this section can be found in the appendix.

Linear Join Trees

THEOREM 6.6 *The total number of enumerations for the space of unconstrained linear join trees by the naive optimization algorithm is no more than* $3^k 2^{n-1}(n - g + g2^w) + (n - g + g2^{w+1})2^{n-g-1}$.

The above theorem reflects the fact that while we restrict the execution space to linear join trees, we do consider placement of user-defined predicates on base relations as well as on intermediate relations. This explains why the dependence on k is $O(3^k)$ while the dependence on n is $O(n2^{n-1})$ (see Section 3.1).

We now turn to the optimization algorithm with complete rank-ordering and show that unlike the naive optimization algorithm, the complexity of optimization algorithm with complete rank-ordering is polynomial in the number of user-defined predicates (for a given bound on the number of relations). We begin by providing an estimate on the number of enumerations in a crude but intuitive way. By multiplying maximum values of the following four factors, we can obtain a rough estimate of the multiplier M for the number of enumerations T of the traditional algorithm such that the total number of enumerations by our algorithm is approximately M^*T. In the following description, the maximum value for each factor is specified inside of the parentheses.

—number of tags for outer relation ($(1 + w)^u$);

—number of tags for inner relation (one for linear join trees and $(1 + w)^u$ for bushy join trees);

—number of evaluable predicates for outer relation ($1 + k$);

—number of evaluable predicates for inner relation ($1 + w$ for linear join trees and $1 + k$ for bushy join trees).

Thus the complexity of the optimization algorithm with complete rank-ordering for the space of unconstrained linear join trees is approximately $(k + 1)(1 + w)^{u+1}$ (i.e., $(1 + w)^u * 1 * (1 + k) * (1 + w)$) times the traditional algorithm. A similar crude estimate for bushy join space results in a multiplier of $(1 + k)^2(1 + w)^{2u}$. The following theorems provide enumeration complexities for the optimization algorithm with complete rank-ordering.

THEOREM 6.7 *The total number of enumerations for the space of unconstrained linear join trees by the optimization algorithm with complete rank-ordering, where there are only user-defined selections, is no more than* $n2^{n-1}(1 + k)(1 + w/2)^{g-1}(1 + (w/2)(1 + (g/n)(2w + 3)))$.

For a given n with user-defined selections only, since we have $w \leq k$, the upper bound is a polynomial in k. In particular, if all user-defined selections apply to the same relation, this complexity is $n2^{n-1}(1 + k)(1 + k/2 + k(2k + 3)/(2n))$. So the extension of our technique to the optimization algorithm for unconstrained linear join trees still guarantees that the complexity of enumeration is polynomial in k for a given n.

THEOREM 6.8 *The total number of enumerations for the space of unconstrained linear join trees by the optimization algorithm with complete* *rank-ordering, where there may be user-defined selections and join predicates, is no more than* $n2^{n-1}(1 + k)(1 + w)^u(1 + gw/n)$.

Since in many applications we expect the number of expensive user-defined predicates in the query to be only a few (and often fewer than the number of joins), it is desirable to ensure that the cost of join enumeration does not increase sharply due to presence of a few user-defined predicates. Unlike the predicate migration algorithm, our approach is able to ensure the above property.

Bushy Join Trees

We now analyze the complexity of the execution space of unconstrained bushy execution trees. The idea of using tags is general enough and is not restricted to unconstrained linear trees. Let us consider the step of joining two unconstrained bushy-join trees P_1 and P_2. Assume that $(p_1, \ldots, p_m)$ and $(q_1, \ldots, q_s)$ are the predicates applicable on P_1 and P_2, respectively, in the order of increasing rank. From the same arguments for unconstrained linear trees, there can be at most $(m + 1)$ and $(s + 1)$ possibilities for pushing down selections on P_1 and P_2, respectively, resulting in at most $(m + 1)(s + 1)$ plans for joining the trees P_1 and P_2. Thus, enumerating the application of user-defined predicates for a join of two relations for unconstrained bushy execution trees is similar to that for unconstrained linear execution trees.

We now turn to the complexities of the optimization algorithm with complete rank-ordering for space of unconstrained bushy join trees. We show that for a given n, the upper bound is still polynomial in k.

THEOREM 6.9 *The total number of enumerations for unconstrained bushy join trees by the optimization algorithm with complete rank-ordering where there are user-defined selections only is no more than* $3^n(1 + k)^2(1 + w/3)^g$.

THEOREM 6.10 *The total number of enumerations for searching the space of unconstrained bushy join trees by the optimization algorithm with complete rank-ordering, where there may be user-defined selections and join predicates, is no more than* $3^n(1 + k)^2(1 + w)^u$.

7. OPTIMIZATION ALGORITHM WITH PRUNING

The naive optimization algorithm, as well as the algorithm with rank ordering, compare plans that have the same tags only. In this section we present two pruning techniques that 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. The results in this section assume that the queries are SPJ queries. We show how these pruning techniques can be integrated with the optimization algorithm with complete rank ordering (or with the naive optimization algorithm). Key observations that drive the soundness of these strategies are: (a) no application of a user-defined predicate has the effect of increasing the size of the input relation; (b) no application of a user-defined predicate alters any other physical properties of the input data stream; and (c) if deferring evaluation of a user-defined predicate is locally no worse, then the choice of globally optimal plan will not be compromised by choosing to defer the evaluation of the user-defined predicate.

7.1 Udp-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.[3] For example, in Figure 4, if the cost of P_6 is less than the cost of P_9, we can prune P_9. In the optimization algorithm with complete rank-ordering, we had to keep both P_6 and P_9 since they had different tags, i.e., different numbers of expensive predicates were applied.

Definition 7.1 A plan Q for a query q is an *extension* of a plan P if P occurs as a subtree of Q.

Definition 7.2 A plan Q is *equivalent* to another plan Q' if they represent the plans for the same expression.

LEMMA 7.3 *Let R and S represent two subplans of a conjunctive query q. Let P' be a plan for $R \bowtie S$. Let P be the plan $\sigma_e(R) \bowtie S$. If $Cost(P) \le Cost(P')$, then for every plan for q that is an extension of plan P', there exists an equivalent plan that extends P, and is no more expensive.*

PROOF. For unconstrained linear join trees, we prove the result by contradiction. The same proof technique extends when the plans are unconstrained bushy join trees.

Let Q' be an extension of plan P' for which the above lemma is violated. In particular, let the sequence of operators that extend P' be $a_1, \ldots, a_n, \sigma_e, b_1, \ldots b_k$, where each of a_i and b_j is an operator. Let $\mathcal{R}_{P'}$ denote the cost of additional operators in Q' that are not present in P'. Therefore, the following equality holds: $Cost(Q') = Cost(P') + \mathcal{R}_{P'}$. Let us consider a plan Q for q that extends P by the sequence of operators $a_1, \ldots, a_n, b_1, \ldots b_k$. Let $\mathcal{R}_{\mathcal{P}}$ denote the cost of additional operators in Q that are not present in P. Because the queries are conjunctive, the size of output of P is no more than the size of output of P'. Thus, it follows that for each a_i, where $i = 1, \ldots, n$, the cost of a_i in Q is no more than that in Q' since the size of the input relation in Q is no larger and the cost of selection, projection, or join is monotonic in the sizes of the input relations. In the case of each operator b_i, the cost is the same for both execution trees. Hence, it follows that $\mathcal{R}_P \le \mathcal{R}_{P'}$. Hence, $cost(Q) \le cost(Q')$. This completes the proof. □

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

We refer to the above lemma as the *udp-pushdown rule*, where we use udp as an abbreviation for user-defined predicate. A corollary of this rule is that if P' 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) $Cost(P) \leq Cost(P')$ and (b) P has a set S of expensive predicates applied prior to a join where S is a superset of S'. Given two plans over the same set of relations, we can easily check (b) by examining the tags of P and P'. The following corollary states this conclusion formally.

Corollary 7.4 Let q be a conjunctive query. If P and P' are two partial plans of q over the same set of relations with the tags T and T' such that T is a subset of T', and $Cost(P) \leq Cost(P')$, then for every plan for q that is an extension of plan P', there exists an equivalent plan that extends P that is no more expensive.

PROOF. We use an argument similar to that used in Lemma 7.3. The key observations are that the size of the output of P is less than or equal to that of P', $cost(P) \leq Cost(P')$, the cost of a plan for a conjunctive query is monotonic in the size of input relations, and the number of operators in any extension of P' is strictly more than the number of operators in P. □

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

Example 7.5 Consider the plans that represent the join among $\{R_1, R_2, R_3\}$. Assume that e_1 and e_3 are applicable on R_1 and e_2 and e_4 are applicable on R_2. Furthermore, assume that the rank order of these predicates are e_1, e_2, e_3 and e_4. If the cost of the plan with the tag $<e_4>$ is lower than that with the tag $<e_2, e_3, e_4>$, we can use the udp-pushdown rule to prune the latter plan.

7.2 Udp-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 optimality of the plan. For example, if the cost of the plan extending P_7 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_6 in Figure 4, we can prune P_6. In the naive optimization algorithm and optimization algorithm with complete rank-ordering, we had to keep both P_6 and P_7 since they have different tags, i.e., different numbers of predicates were applied to each of the plans.

LEMMA 7.6 *Let R and S represent two subplans of a conjunctive query q. Let P, P', and P'' represent plans for $R \bowtie S$, $\sigma_e(R) \bowtie S$, $\sigma_e(R \bowtie S)$, respectively. If $Cost(P'') \leq Cost(P')$, then for every plan for q that is an extension of plan P', there exists an equivalent plan that extends P that is no more expensive.*

PROOF. Consider a plan Q' that extends a plan P' with a sequence of operators a_1, .., a_n. We can construct a new plan Q by extending P with the sequence of operators σ_e, a_1, .., a_n, which is equivalent to extending P'' by a_1, .., a_n. Since $Cost(P'') \le Cost(P')$, it follows that $Cost(Q) \le Cost(Q')$. □

We refer to the above as the *udp-pullover rule*, since the plan P in the lemma corresponds to the case where the predicate is pulled up. We can generalize the consequences of the above rule. Let us consider plans P and P' over the same set of relations but with different tags T and T'. If the tag T' is a subset of T, then all predicates that are evaluated in P are also evaluated in P'. Let $Diff(T, T')$ represent the set of predicates that are evaluated in T' but not in T. We can use the same proof technique as in Lemma 7.6 to conclude:

Corollary 7.7 Let q be a conjunctive query. Let P and P' be two partial plans with tags T and T' over the same set of relations and T a superset of T'. Let P'' be the plan obtained by applying the predicates in $Diff(T, T')$ to P. If $cost(P'') \le cost(P')$, then for every plan for q that is an extension of plan P', there exists an equivalent plan that extends P that is no more expensive.

For a given P, we can construct a set of plans (e.g., P') each of which may be used to prune P. We can refer to the above set as *pullovercheaper*(P). The following example illustrates the corollary. Consider Example 5 with the following change: the cost of the plan P with the tag $T = <e_4>$ is higher than the cost of the plan P' with the tag $T' = <e_2, e_3, e_4>$. Notice that the tag T is a subset of the tag T'. The set $Diff(T, T') = \{e_2, e_3\}$. 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 $\{e_2, e_3\}$ after the join is no more than the cost of P. Finally, note that the pruning strategies of both udp-pushdown rule and udp-pullover rule are still applicable for execution trees that are bushy.

7.3 An Algorithm that Exploits Pruning Strategies

We can use the udp-pushdown rule to conclude that if the *optimal* plan P for $\sigma_e(R) \bowtie S$ is no more expensive than the *optimal* plan P' for $R \bowtie S$, then for any extension to plan P', we will find an extension to P that is no more expensive. In other words, the udp-pushdown rule provides a sufficient condition for the predicate e to be pushed down, i.e., we do not need to consider plans where e is pulled up past future joins. This results in reducing the number of plans over the same set of relations, but with distinct tags that need to be considered in the future iterations of the dynamic programming-based optimization algorithm without sacrificing the optimality. A scenario where the udp-pushdown rule is extremely

```
procedure Opt-Rank-Pruning:
for i := 2 to $n$ do {
    for all $S \subseteq \{R_1, ..., R_n\}$ s.t. $||S|| =$ i do {
        initialize an array of bestPlan to a dummy plan with infinite cost
        for all $R_j, S_j$ s.t. $S = \{R_j\} \cup S_j$ and $R_j \notin S_j$ do {
            for all $P$ stored in plan table for $S_j^t$ with all different tag $t$ do {
                $s$ := Number of evaluable predicates on $P$
                $r$ := Number of evaluable predicates on $R_j$
                for all u := 0 to $s$ do {
                    for all v := 0 to $r$ do {
                        p := extjoinPlan($P$, $R_j$,u,v)
                        if addtotable(p) then {
                            remove pruneset(p)
                            add p to bestPlan
                        }
                    }
                }
            }
            copy plans in bestPlan to optPlan($S$)
        }
    }
}
finalPlan := a dummy plan with infinite cost
for all plan p $\in$ optPlan($\{R_1, ..., R_n\}$) do {
    if complete_cost(p) < cost(finalPlan)
        finalPlan := completed plan of p
}
return (finalPlan)
```

Fig. 9. Optimization algorithm with pruning for linear join trees.

effective is when the user-defined predicates are relatively cheap, much like built-in predicates in SQL. In such cases, the udp-pushdown rule is effective, since it can be used to pin the selections as soon as they are evaluable and helps avoid constructing plans where the predicates are pulled up.

Similar comments apply to the effectiveness of the udp-pullover rule. The udp-pullover rule allows us to conclude that if the *optimal* plan for $\sigma_e(R) \bowtie S$ that evaluates e prior to the join is no less expensive than a plan where e is evaluated immediately after the join, then we do not need to consider extensions to the former plan. Thus, this pruning rule helps us avoid generating alternative plans that push down user-defined predicates and are suboptimal. For example, consider the case where user-defined predicates are expensive and join operations decrease the cardinality of the relation. In such a case, the optimal plan corresponds to the case where all user-defined predicates follow evaluation of all joins. In such a case, the use of udp-pullover rule can help avoid unnecessary enumeration of many plans that push down predicates.

Figure 9 describes an optimization algorithm that augments the naive optimization algorithm with complete rank-ordering and pruning strategies. Given choices of u and v, rank ordering uniquely determines the tag for plan p in Figure 9. Plan p will be compared against plans over the *same set of relations* that have already been stored. 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 *bestPlan* with the same tag, if any; (2) if the set of plans $pullovercheaper(p)$ is empty, i.e., the udp-pullover rule cannot be used to prune p. Otherwise, the predicate *addtotable*(p) becomes true and plan p is added to *bestPlan*. Next, this new plan p is used to prune plans that are currently in *bestPlan*. 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 *pushdownexpensive*(p), i.e., plans that may be pruned with p using the udp-pushdown rule.

8. CONSERVATIVE LOCAL HEURISTIC

The optimization algorithms that we propose require tags to be maintained with plans. Potentially, for every subset of relations in the query, an optimal subplan may need to be stored for every distinct tag. In this section we present the conservative local heuristic, which provides a simpler alternative for implementation. First, this technique does not require tags to be maintained. So incorporating the heuristic in an existing System-R style optimizer is easier. 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 conservative local heuristic is best described by explaining the *Pull-Rank* heuristic that has been proposed but was found inadequate for generating plans of acceptable quality [Hellerstein 1994]. Pull-Rank 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 Pull-Rank 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 (iii) Cost of evaluating the remainder of the user-defined functions that are evaluable before the join but are deferred past the join. Pull-Rank chooses the plan that has the minimum completion cost. Thus, the algorithm *greedily* pushes down predicates. For example, if Pull-Rank 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., Pull-Rank will *not* consider any plans where u is evaluated after j. Thus, Pull-Rank fails to explore such plans where deferring evaluation of predicates past more than one joins is significantly better than choosing to

```
procedure Opt-Rank-Conservative:
for i := 2 to $n$ do {
    for all $S \subseteq \{R_1, ..., R_n\}$ s.t. $||S|| =$ i do {
        bestPlan_push := a dummy plan with an infinite cost
        bestPlan_complete := a dummy plan with an infinite cost
        for all $R_j, S_j$ s.t. $S = \{R_j\} \cup S_j$ and $R_j \notin S_j$ do {
            for all $P \in$ optPlan($S_j$) do { (* at most 2 plans possible *)
                $s$ := Number of evaluable predicates on $P$
                $r$ := Number of evaluable predicates on $R_j$
                for all u := 0 to $s$ do {
                    for all v := 0 to $r$ do {
                        p := extjoinPlan($P$, $R_j$,u,v)
                        if cost(p) < cost(bestPlan_push)
                            bestPlan_push := p
                        if complete_cost(p) < complete_cost(bestPlan_complete)
                            bestPlan_complete := p
                    }
                }
            }
        }
    }
    optPlan($S$) := { bestPlan_push, bestPlan_complete }
}
finalPlan := a dummy plan with infinite cost
for all plan p $\in$ optPlan($\{R_1, ..., R_n\}$) do {
    if complete_cost(p) < cost(finalPlan)
        finalPlan := completed plan of p
}
return (finalPlan)
```

Fig. 10. Optimization algorithm with conservative local heuristic for linear join trees.

greedily push down predicates based on local comparison of completion costs.

The conservative local heuristic guards against the above shortcoming of the pull-rank heuristic by retaining at most two intermediate plans with different tags over the same set of relations. In contrast, the pull-rank heuristic keeps at most one plan over the same set of relations. One of the two plans picked by the conservative local heuristic is the same as that of pull-rank. The additional plan over the same set of relations considered by the conservative local heuristic is where evaluation of all remaining applicable user-defined predicates is deferred. The latter plan is compared with variants of itself where the push-down rule is applied and the plan with the cheaper cost is retained. Therefore, the conservative local heuristic can choose among plans that result from application of a sequence of udp-pushdown and udp-pullover rules. The two plans picked by the conservative local heuristic complement each other, and the heuristic can guard against the choice of a poor plan resulting from greedily pushing down a predicate by the Pull-Rank algorithm. Thus, conservative local heuristic can find optimal plans that Pull-Rank and other global heuristics fail to

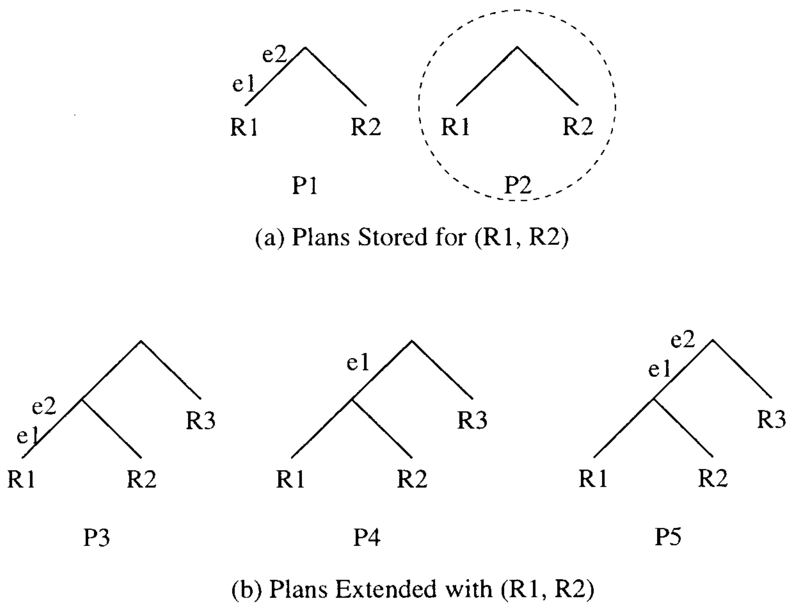

e2
e1
R1
R2
P1
R1
R2
P2
(a) Plans Stored for (R1, R2)
e2
e1
R1
R2
R3
P3
e1
R1
R2
R3
P4
e2
e1
R1
R2
R3
P5
(b) Plans Extended with (R1, R2)

Fig. 11. Search space with rank-ordering and conservative local heuristic.

find due to their greedy approach, but incurs only low computational overhead. This is illustrated by the following example.

Example 8.1 Consider the query $Q = \sigma_e(R_1) \bowtie R_2 \bowtie R_3 \bowtie R_4$. Assume that the plan $\sigma_e(R_1 \bowtie R_2 \bowtie R_3) \bowtie R_4$ is optimal. Note that nothing in the global heuristic that either pushes down or pulls up all the selections can find the optimal plan. If the plan for $\sigma_e(R_1) \bowtie R_2$ is cheaper than $\sigma_e(R_1 \bowtie R_2)$, then the pull-rank greedily pushes down P and fails to obtain the optimal. However, our algorithm with a conservative local heuristic 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 is optimal, and therefore only our algorithm is able to find it.

Example 8.2 Another example of search space explored by a conservative local heuristic is shown in Figure 11. It illustrates those parts of the execution plans that need to be considered for a query of three relations and two expensive selection predicates e_1 and e_2 on R_1. Plan P_2 is the additional plan stored by the conservative local heuristic for (R_1, R_2). By storing the plan, P_4 and P_5 are additionally enumerated for (R_1, R_2, R_3).

To implement the conservative local heuristic with a traditional system R algorithm, we pick one additional plan, in addition to the plan picked by pull-rank at each join step based on sum of the costs of the following two components: (1) cost of evaluating expensive predicates that are pushed down at this step; (2) cost of the join. Let us refer to the sum of these two costs as the *pushdown-join* cost. This is the same as assuming that *deferred* *predicates* are evaluated for "free" (i.e., the cost of evaluating all remaining evaluable predicates 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. Since conservative local heuristic picks two plans, one for completion cost and the other for pushdown-join cost, it is possible to consider 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 pull-rank) as possible candidates for the optimal plan.

The optimization algorithm with complete rank-ordering and the conservative local heuristic for the space of unconstrained linear join trees is shown in Figure 10. The algorithm works similarly to traditional optimization algorithms, except that we keep two plans for S rather than one. Unlike the optimization algorithm with complete rank-ordering where we keep a plan for each distinct tag of S, the conservative local heuristic retains at most two plans for S, i.e., the plans with the minimum pushdown join cost and the minimum completion cost, respectively (represented by the variables bestPlan push and bestPlan complete in the algorithm). At the end of the final join, we consider two plans stored for the relations $\{R_1, \ldots R_n\}$. As in the case of the optimization algorithm with complete rank-ordering, one plan may need to be completed by adding the step to evaluate the remainder of the predicates. Finally, the cheapest one between the plans is chosen.

Since, for the join of every subset of relations, at most two plans are stored by a conservative local heuristic, we never need to consider storing more than 2^{n+1} plans. Thus, unlike the algorithm in Figure 9, the number of subplans that need to be optimized does not grow with the increasing number of user-defined predicates. However, a conservative local heuristic may miss an optimal plan since, unlike the optimization algorithm with complete rank-ordering, distinctions among the tags are not made in this algorithm. Nevertheless, the experimental results of Section 9 indicate that the quality of the plan is very close to the optimal plan. Furthermore, as the following lemma states, the conservative local heuristic produces an optimal plan in several important special cases.

LEMMA 8.3 *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 user-defined predicates are evaluated as soon as possible.*

PROOF. Since the heuristic is a strict extension of pull-rank, (1) and (3) follow from the property of pull-rank. When there is only one predicate, at each step the heuristic considers the plans that defer as well as push down the selection. This guarantees optimality for case (2). Another way to view the same result is that, in this case, only two values of tags are possible, and therefore the heuristic is able to guarantee optimality. □

THEOREM 8.4 *The total number of plans enumerated by the optimization algorithm with conservative local heuristic for the space of unconstrained linear join trees, where there may be user-defined selections and join predicates, is no more than* $n2^{n-1}2(1 + k)(1 + gw/n)$.

9. PERFORMANCE EVALUATION

To assess the effectiveness of our optimization algorithms, we implemented the algorithms proposed in this paper by extending a System R style optimizer. In this section we present the results of doing performance evaluations on our implementations. In particular, we establish:

(1) the pruning strategies that we proposed significantly improve the performance of the optimization algorithm with complete rank-ordering;

(2) the plans generated by the traditional optimization algorithm suffer from poor quality;

(3) the plans generated by the pull-rank algorithm are better (less expensive) than the plans generated by the traditional optimizer, but are still significantly worse than the optimal;

(4) the conservative local heuristic reduces the optimization overhead significantly, and generates plans that are very close to optimal.

9.1 Experimental Framework

We used an experimental framework similar to that in Ioannidis and Kang [1990] and Chaudhuri and Shim [1994]. We performed experiments using a Sun Ultra-2/200 machine with 512 MB of RAM and running Solaris 2.5.

The algorithms were run on queries consisting of equijoins. The experiments were over a randomly generated relation catalog where relation cardinalities ranged from a thousand to one million tuples, and the numbers of unique values in join columns varied from 10% to 100% of the corresponding relation cardinality. The selectivities of expensive predicates were randomly chosen from 10^{-6} to 1.0 and the cost per tuple of expensive predicates was represented by the number of I/O (page) accesses and 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 was again uniformly random. We considered nested-loop, merge-scan, and simple and hybrid hash joins as join methods [Shapiro 1986]. In our experiments, only the cost for number of I/O (page) accesses was accounted for. For our experiments, we generated 4 join (i.e., join among five relations) queries, 6 join queries, and 10 join queries.

We performed two sets of experiments. In the first set, we varied the *number* of user-defined selection predicates that apply on one relation. In the second set, we varied the *distribution* of the user-defined selection predicates among *multiple* relations in the query, i.e., we kept the number of selection predicates fixed, but varied how these predicates are distributed among the relations in a query. At one extreme, all the user-defined selections were applied on the same relation, and at the other extreme, they were equally distributed among the relations in the query. These two sets of experiments are described in Section 9.3 and Section 9.4, respectively.

9.2 Candidate Algorithms

For each query instance, we ran the following five optimization algorithms. The plans that needed to be stored/enumerated due to interesting orders were taken into consideration in reporting the results.

—*Traditional algorithm*: The system R style optimization algorithm that evaluates all expensive (or otherwise) predicates as early as possible.

—*Pull-rank algorithm*: This algorithm, introduced in Hellerstein [1994], considers all possible placement of expensive predicates locally either immediately preceding or immediately after join, and picks the cheapest plan among them.

—*Opt-rank algorithm*: This is an optimization algorithm in Section 5 that uses rank-ordering. It compares plans that have the same tag over the same set of relations.

—*Opt-rank-pruning algorithm*: This is an extension of the Opt-rank algorithm that incorporates the pruning strategies described in Section 7.

—*Opt-rank-conservative algorithm*: This algorithm uses our conservative local heuristic with complete rank-ordering illustrated in Section 8.

9.3 Experiment 1: Effect of Number of User-Defined Predicates

We experimented to see how the optimization algorithms behave as we increased the number of user-defined selection predicates. In this set of experiments, the number of user-defined predicates was varied from 1 to 6. Figure 12 shows the *number of enumerated plans* and the *quality of plans* generated by each algorithm. The results presented here for each data point represents an average over 100 queries. These queries were generated by randomly choosing one relation on which all the user-defined predicates apply and then randomly picking the cost and selectivities of the predicates as well.

Number of enumerations. Figure 12 shows that the average number of enumerated plans for the opt-rank algorithm is approximately linear, and this number grows at a rate slower than the worst-case complexity, determined in Section 6. For 10 join queries (i.e., 11 relations), Table III shows the factor by which the worst-case complexity of enumeration grows,

Table III. Worst-Case Estimates for Enumerated Plans

Number of user-defined predicates	1	2	3	4	5	6
Multiplicative factor for opt-rank	3.5	7.9	14.9	25.0	38.7	56.6

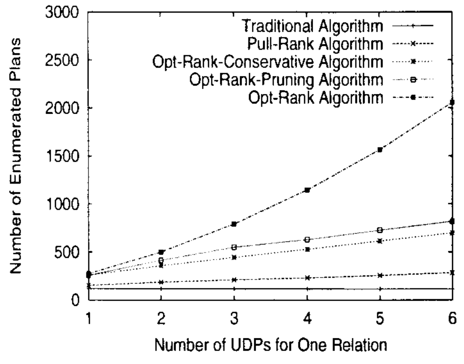
compared to the number of enumerations in the traditional optimizer. The enumerations necessary in the latter case is independent of the number of user-defined predicates. The results obtained for queries with fewer joins show a similar trend.

A comparison of the performances of the opt-rank algorithm and opt-rank-pruning algorithm shows that our proposed pruning techniques are extremely effective. While both algorithms are guaranteed to generate optimal plans, the gap in the number of enumerated plans between the opt-rank algorithm and opt-rank-pruning algorithm increases significantly as the number of user-defined predicates grows. When six user-defined predicates are applied, the opt-rank algorithm generated about three times more plans than the opt-rank-pruning algorithm. The pull-rank algorithm considers locally all possible cases of applying user-defined predicates. Thus, the number of enumerated plans is slightly more than that of the traditional optimizer. As expected, the number of enumerations for the pull-rank algorithm was smaller than for the opt-rank-conservative algorithm. However, as discussed below, the quality of plans generated by the pull-rank algorithm was significantly worse.

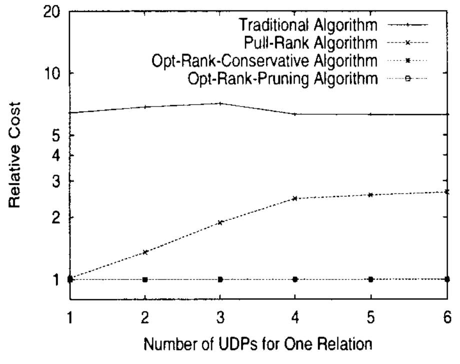
Quality of plans. We compare the relative costs of the plans generated by each algorithm. The cost of the plan generated by the opt-rank-pruning algorithm scaled as 1.0 and the relative cost was plotted in the log scale. Since the opt-rank and opt-rank-pruning algorithms always generate optimal plans, 1.0 represents the cost of plans generated by either of these algorithms.

The results show that the quality of plan generated by a traditional optimizer suffers significantly across the board. The quality of plans generated by the pull-rank algorithm gets worse as the number of expensive predicates increases. As noted in Section 8, pull-rank fails to explore plans where deferring evaluation of predicates past more than one joins is significantly better than choosing to greedily push down predicates based on a local comparison of completion costs. We examined the plans considered by the pull-rank algorithm and discovered that the number of plans generated, in which user-defined predicates were applied earlier compared to their placement in the corresponding optimal plans, increased gradually as we increased the number of applicable user-defined predicates.

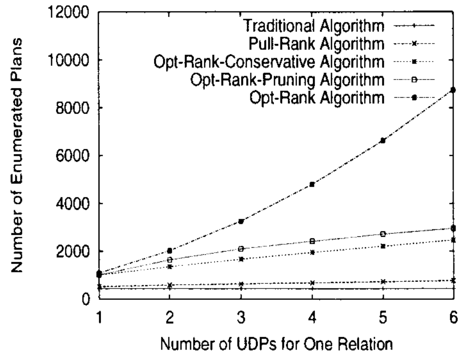
On the other hand, the opt-rank-conservative algorithm chooses plans that are identical or are very close (less than 0.4% difference) to the optimal. This is borne out by the fact that the graphs for the heuristic and the algorithms producing the optimal plans are practically indistinguishable. Note that the opt-rank-pruning algorithm is the same as the opt-rank-conservative algorithm when the number of expensive predicates is one.



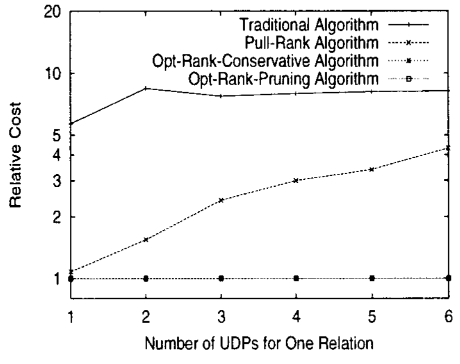
4 Join Query



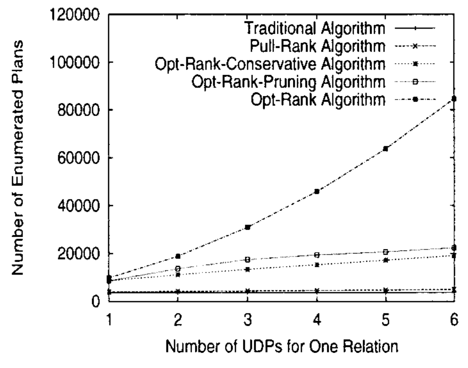
4 Joins Query



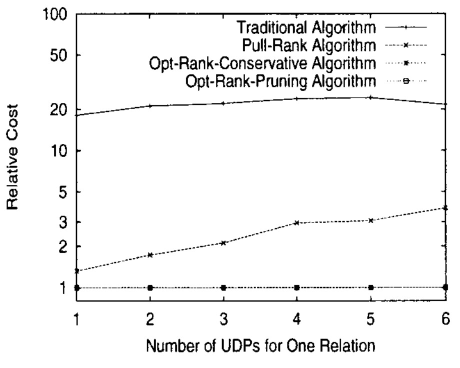
6 Join Query



6 Join Query



10 Join Query



10 Join Query

Fig. 12. Performance on a varying number of user-defined predicates.

Thus, the algorithms pick the same plan when we have only one user-defined predicate.

Table IV. Worst-Case Estimates for Enumerated Plans

Number of relations with UDF	1	2	3	4	5	6
Multiplicative factor for opt-rank	56.6	86.7	109.5	254.6	580.4	152.2

9.4 Experiment 2: Effect of Distribution of Predicates

In this experiment we varied the *distribution* of predicates. Quantitatively, we define distribution as the fraction of the relations in a query with applicable user-defined predicates. For the experiment in this section, we assumed that six predicates are evenly distributed among relations with applicable user-defined predicates. The results of these experiments are shown in Figure 13.

Number of enumerations. Compared to the traditional algorithm, the number of plans enumerated by the opt-rank-conservative algorithm is only modestly higher. This is because the opt-rank-conservative algorithm stores at most two plans for any subquery in the plan table for each interesting order. The graphs clearly demonstrate that the savings made by the opt-rank-conservative algorithm, with respect to the opt-rank algorithm, are significant across all distributions.

This set of experiments also indicates that the number of plans enumerated by the opt-rank algorithm increases up to a value of g (number of relations with user-defined predicates), after which a downward trend begins. Moreover, the value of g at which this occurs is nearly independent of the number of relations present in the query. We now explain each of these observations, based on the complexity analysis in Section 6.

Theorem 6.7 provides us with an upper bound on the number of plans enumerated by opt-rank. The upper bound stated in the theorem (using the notation in Section 6) is a function of n, k, g and w. Table IV shows the worst-case estimate for the number of enumerated plans for ten join queries with six user-defined predicates ($n = 11$, $\mathrm{k} = 6$) when the user-defined predicates are distributed among multiple relations (up to six). When g is 1, 2, 3, and 6, every relation having user-defined predicates has the same number of user-defined predicates, which is $w = k/g$. This results in a unique upper bound based on Theorem 6.7 and represented in Table IV. Thus, from the table, we observe that if all of six expensive predicates (i.e. $g = 1$, $w = 6$) are applied to one relation in ten join queries, we can have at most 56.6 times more enumerated plans than the traditional optimizer. When we distribute six expensive predicates among two relations (i.e., $g = 2$, $w = 3$), at most 86.7 times more enumerations are generated than the traditional optimizer. However, when $g = 4$ and $g = 5$, w can be either one or two. Table IV records the worst-case upper bound for each of these cases. From the table, it is apparent that the worst-case upper bound on the number of enumerated plan peaks and then takes a downward turn. Since the graphs shown in Figure 13 is the average number of enumerations out of 100 random queries, the above complexity cannot be used directly to explain our experiment. However, the experi-

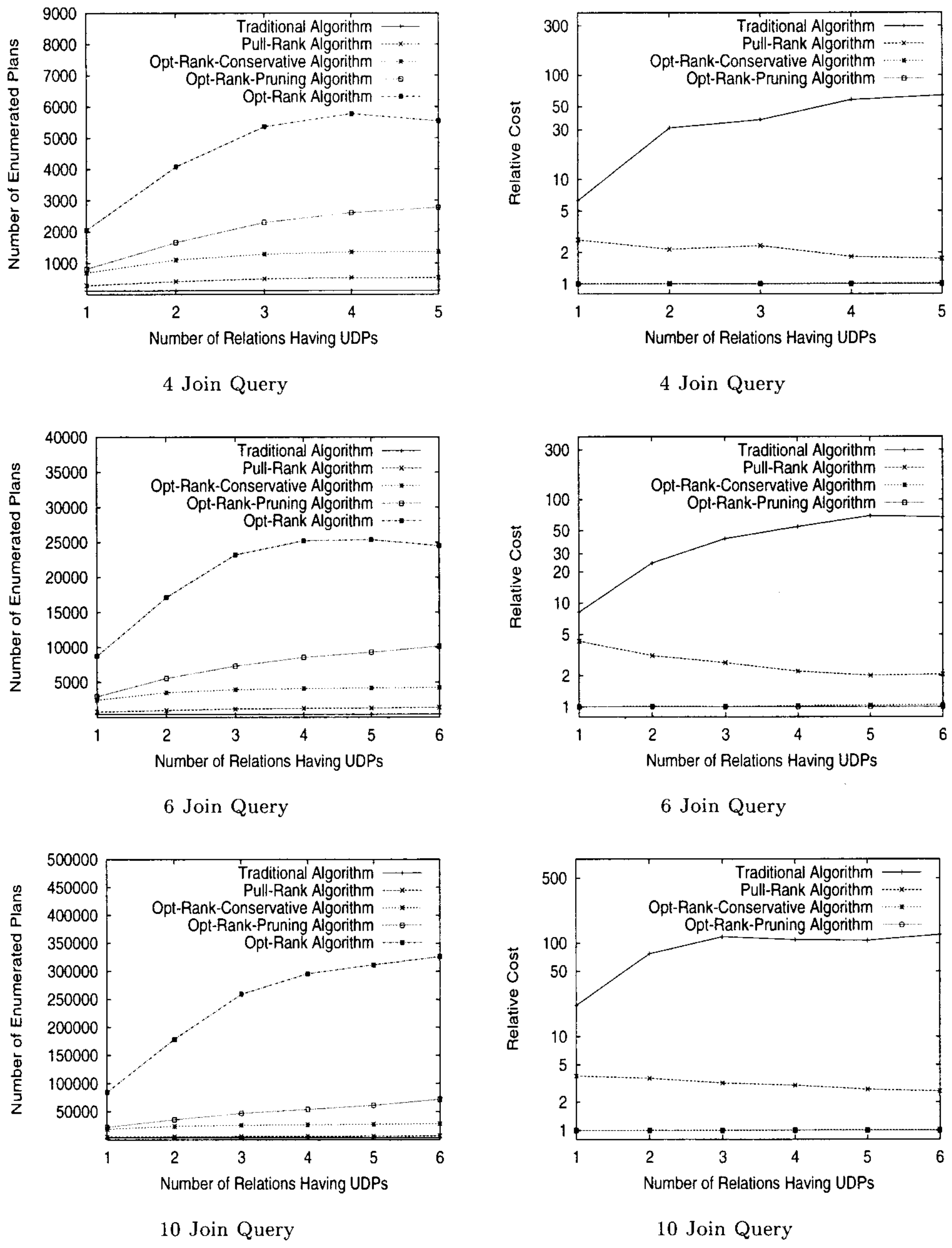


Fig. 13. Performance on varying distributions of user-defined predicates.

mental results do show a trend similar to that in Table IV. The trend can also be explained analytically. The leading dependence on g in Theorem 6.7 is the factor $(1 + w/2)^{g-1}$. However, for this set of experiments, we have

$w = k/g$. The effect of this factor can be intuitively grasped by analyzing the expression $(k/g)^g$ (assuming $w \geq 2$), an upper bound on the above factor. The latter expression, when viewed as a continuous function of g, has a maxima $g = k/e$ and decreases as g is increased further. The experimental result demonstrates such a variation with respect to g.

Quality of plans. As expected, the results show that the quality of plans generated by the traditional optimizer continues to suffer significantly for various distributions of user-defined predicates, in comparison to the quality of plans generated by the opt-rank-pruning and opt-rank-conservative algorithms. Across different distributions, opt-rank-conservative produced plans with cost at most 3% worse than that of opt-rank, and thus demonstrates its robustness as a heuristic. In contrast, the pull-rank algorithm generated plans that are about 80% more expensive than those generated by the opt-rank-conservative algorithm. This further justifies the algorithmic changes to pull-rank incorporated in opt-rank-conservative.

The graphs in Figure 13 indicate that as we distribute the user-defined predicates to more relations, the relative cost of plans generated by the pull-rank algorithm decreases slightly. When we examined the plans obtained by the pull-rank algorithm in our experiment, we observed that the number of plans generated, in which the user-defined predicates are applied earlier in the execution plan in comparison to their application in optimal plans, is reduced as we increase the number of relations with user-defined predicates. The effect of increasing the number of relations with user-defined predicates is to effectively reduce the number of user-defined selections applicable to every relation. This resulted in a fewer number of alternative plans not considered by pull-rank. So the probability that pull-rank picks a plan that is optimal or near-optimal is increased.

10. CONCLUSION

With the growing popularity of object-relational database systems, the problem of optimizing queries with user-defined predicates has become increasingly important. The user-defined predicates in the query may be join or selection predicates. In this paper we presented algorithms that find the optimal plans for conjunctive queries with user-defined predicates. These algorithms extend the traditional System R style optimization algorithms used by many commercial optimizers. The naive optimization algorithm that we present guarantees the optimal plan. The optimization algorithm with complete rank-ordering demonstrates that by exploiting the special structure of cost-models, we can find the optimal plan in time polynomial in the number of user-defined predicates in the query (for a given bound on the number of relations). This result depends on a novel and nontrivial proof that shows that placement of user-defined predicates in a linear sequence of joins must respect rank-ordering. We presented pruning techniques that significantly reduce the cost of searching execution space without compromising the optimality for both naive algorithms and the optimization algorithm with complete rank-ordering. We also described the efficient conservative local heuristic-based algorithm. Although this heuristic cannot guarantee generation of an optimal plan in all cases, it can guarantee the optimal in many important cases, and the experimental results show that the plans generated are either indistinguishable from or very close to the optimal. Overall, the optimization algorithm with complete rank-ordering is the natural choice wherever the guarantee of optimality is desirable and where the cost formulas respect the conditions of regular joins. Our results indicate that the conservative local heuristic is very efficient, and is therefore an excellent choice where suboptimality may be tolerated. The naive optimization algorithm should be used only where (a) suboptimality cannot be tolerated and (b) assumptions on the cost model made by the optimization algorithm with complete rank-ordering are not acceptable. Finally, note that the pruning techniques described in Section 7 should be used under all circumstances, since they simply speed up the search for a plan without sacrificing optimality.

Although in this paper we described algorithms for conjunctive queries, these algorithms need to be extended to the cases where the SPJ query has a more complex boolean expression in the `Where` clause. While a preprocessing step can convert the Where clause into a conjunctive normal form $A_1 \wedge \ldots \wedge A_k$, each A_i no longer needs to be atomic and independent. It is necessary to derive costs and selectivities for each A_i and take into account their dependence. Finally, while we showed that the optimization algorithm with complete rank-ordering guarantees the optimal for common join implementations, it is still an open problem if the same result extends to more general cost models.

APPENDIX

A. Proofs of Theorems.

PROOF. *(Theorem 6.6).* Let us consider the number of plan enumerations needed when the outer relation is a join of $(i + j)$ relations, where i of the relations have user-defined predicates defined on them and the remaining j relations do not have any user-defined predicates on them. Note that, using the notation in Table II, $i \leq g$, $j \leq n - g$. Prior to joining the outer relation with the inner, one or more of the user-defined predicates that are evaluable can be executed. Since there are k user-defined predicates, the number of tags of size p is $\binom{k}{p}$, where $0 \leq p \leq k$. Furthermore, for each tag of size p, there are 2^p enumerations where each enumertion corresponds to evaluating a subset of the evaluable user-defined predicates. Finally, for each choice of the outer and for each choice of the set of user-defined predicates applied on the outer, we must choose an inner reltion and a set of user-defined predicates to be applied on the inner relation prior to the join. If the inner relation is chosen among $g - i$ relations (i.e., a relation that has user-defined predicates), we must consider the application of predicates in inner relations. Since the maximum number of user-defined predicates applicable per relation is w, the number of enumerations due to evaluation of user-defined selections on inner relation 2^w. If an inner relation is chosen from among $n-g-j$ relations that do not have any user-defined predicates, then no predicates may apply on the inner relation. Thus, for each outer relation of distinct i and j, the number of enumerations for the join with an inner relation is

$$\sum_{p=0}^{k}\binom{k}{p}2^p((n-g-j)+(g-i)2^w).$$

We can now determine an upper bound on the total number of enumerations by summing over i and j, as below. The first term considers the case where the expression for the outer relation has at least one relation with a user-defined predicate on it. The second term considers the case where the outer relation has no user-defined predicate defined on it.

$$\begin{aligned}
E_G = {} & \sum_{i=1}^{g}\binom{g}{i}\sum_{j=0}^{n-g}\binom{n-g}{j}\sum_{p=0}^{k}\binom{k}{p}2^p((n-g-j)+(g-i)2^w) \\
& + \sum_{j=1}^{n-g}\binom{n-g}{j}((n-g-j)+g2^w) \\
& = \sum_{0\le i\le g}\binom{g}{i}\sum_{0\le j\le n-g}\binom{n-g}{j}3^k((n-g-j)+(g-i)2^w) \\
& + \sum_{0\le j\le n-g}\binom{n-g}{j}((n-g-j)+g2^w) \\
& - \sum_{0\le j\le n-g}\binom{n-g}{j}3^k((n-g-j)+g2^w)-((n-g)+g2^w) \\
& = 3^k2^{n-1}(n-g+g2^{w+1}-g2^w) \\
& + ((n-g+g2^w)2^{n-g}-(n-g)2^{n-g-1}) \\
& - 3^k((n-g+g2^w)2^{n-g}-(n-g)2^{n-g-1})-((n-g)+g2^w) \\
& \le 3^k2^{n-1}(n-g+g2^w)+(n-g+g2^{w+1})2^{n-g-1} \qquad \square
\end{aligned}$$

PROOF. *(Theorem 6.7)*. The idea of this proof is similar to that of Theorem 6.6. For each distinct set of relations of size $i+j$, in which i relations are chosen from the designated g relations with one or more user-defined selections defined on them, j relations are selected from the remaining $n-g$ relations with no user-defined predicates on them, at most $(1+w)^i$ distinct tags are possible because of complete rank-ordering.

Furthermore, evaluable predicates on the outer relations must also be applied in the rank order. Therefore, for each outer relation with at least one evaluable user-defined predicate (i.e., $i > 0$), and for each distinct tag we need to have only $1 + k$ enumerations, since the number of evaluable predicates is at most k. If the inner relation is chosen among $g - i$ relations, we have to consider the application of predicates on the inner relation prior to the join. There can be at most $(1 + w)$ enumerations due to evaluation of user-defined predicates prior to the join. Note that when there is no relation in the expression for an outer relation chosen from g relations that have any user-defined predicate, we cannot apply any user-defined predicate on the outer relation. Similarly, we cannot apply any user-defined selection predicate on the inner relation if the inner relation is chosen among $n - g - j$ relations that do not have any user-defined predicate. The first factor of the first expression below corresponds to the latter case. So the number of enumerations is upper-bounded by

$$
\begin{aligned}
E_L = {} & \sum_{i=1}^{g}\binom{g}{i}(1+w)^u\sum_{j=0}^{n-g}\binom{n-g}{j}((n-g-j)(1+k)+(g-i)(1+k)(1+w))(g-i) \\
& + \sum_{j=1}^{n-g}\binom{n-g}{j}((n-g-j)+g(1+w)) \\
= {} & \sum_{i=0}^{g}\binom{g}{i}(1+w)^u\sum_{j=0}^{n-g}\binom{n-g}{j}((n-g-j)(1+k)+(g-i)(1+k)(1+w)) \\
& + \sum_{j=0}^{n-g}\binom{n-g}{j}((n-g-j)+g(1+w)) \\
& - \sum_{j=0}^{n-g}\binom{n-g}{j}((n-g-j)(1+k)+g(1+k)(1+w)) \\
& \qquad - ((n-g)+g(1+w)) \\
= {} & (1+k)n2^{n-1}(1+w/2)^{g-1}(1+(w/2)(1+(g/n)(2w+3))) \\
& - k(n+2gw+g)2^{n-g-1} - (n+gw) \\
\le {} & (1+k)n2^{n-1}(1+w/2)^{g-1}(1+(w/2)(1+(g/n)(2w+3))) \qquad \square
\end{aligned}
$$

PROOF. *(Theorem 6.8).* The proof is similar to that of Theorem 6.7. However, since there are user-defined join predicates, the computation for the number of distinct tags needs to be modified. From Table II, we observe that u is the number of relations having selection predicates and the number of pairs of relations having user-defined join predicates. Therefore, an upper bound on the number of distinct tags associated with any plan during optimization is $(1 + w)^u$. Therefore, we obtain an upper bound on the number of enumerations by replacing $(1 + w)^i$ with $(1 + w)^u$ in the earlier proof:

$$E_L = \sum_{i=1}^{g}\binom{g}{i}(1 + w)^u\sum_{j=0}^{n-g}\binom{n-g}{j}((n - g - j)(1 + k) + (g - i)(1 + k)(1 + w))$$

$$+ \sum_{j=1}^{n-g}\binom{n-g}{j}((n - g - j) + g(1 + w))$$

$$= (1 + w)^u(1 + k)n2^{n-1}(1 + gw/n) - k(n + 2gw + g))2^{n-g-1} - (n + gw)$$

$$\leq (1 + w)^u(1 + k)n2^{n-1}(1 + gw/n) \qquad \square$$

PROOF. *(Theorem 6.9).* The formulation is similar to the proof for the unconstrained linear join trees. However, since the inner relation in a bushy join may itself consist of joins of multiple relations, the number of enumerations due to application of user-defined predicates on the inner relation prior to the join is no longer bounded by $(1 + w)$, but is bounded by $(1 + k)$ (as in the case of the outer relation). Thus, when each of outer and inner relations has at least one relation with a user-defined predicate, the number of enumerations due to application of user-defined predicates prior to the join between the outer and inner becomes $(1 + k)^2$. Let us index each subplan by two parameters (i, j) where i is the relation chosen among g relations and j is the relation chosen among $n - g$ relations in which user-defined predicates cannot be applied. The number of possible plans for inner relations in which no user-defined predicates are applicable is $2^{n-g-j} - 1$. This corresponds to the case in which all relations that contribute to the inner relation are chosen from $n - g$ relations. Thus, the number of plans for inner relations with applicable predicates is $2^{n-i-j} - 2^{n-g-j}$, where $(2^{n-i-j} - 1)$ is an upper bound on the total number of possible plans representing the inner relation. The number of enumerations is therefore upper-bounded as

$$E_B = \sum_{i=1}^{g}\binom{g}{i}(1 + w)^i\sum_{j=0}^{n-g}\binom{n-g}{j}((2^{n-g-j} - 1)(1 + k) + (2^{n-i-j} - 2^{n-g-j})(1 + k)^2)$$

$$+ \sum_{j=1}^{n-g}\binom{n-g}{j}((2^{n-g-j} - 1) + (2^{n-j} - 2^{n-g-j})(1 + k))$$

$$= (1 + k)^2((1 + w/3))^g3^n - k((1 + k)2^g + 1)3^{n-g}$$

$$- 2^n((1 + k)(1 + w/2)^g + (1 + k) - k(1/2)^g - k(1/2)^g) + 1$$

$$\leq (1 + k)^2(1 + w/3)^g3^n \qquad \square$$

PROOF. *(Theorem 6.10)*. The proof is very similar to that for Theorem 6.9. However, since there are user-defined join predicates, the computation for the number of distinct tags needs to be modified. An upper bound on the number of distinct tags associated with any plan during optimization is $(1 + w)^u$ (see Table II). Thus, we are able to obtain an upper bound on the number of enumerations by replacing $(1 + w)^i$ with $(1 + w)^u$ in Theorem 6.9.

$$E_B = \sum_{i=1}^{g}\binom{g}{i}(1 + w)^u\sum_{j=0}^{n-g}\binom{n-g}{j}((2^{n-g-j} - 1)(1 + k))$$

$$+ (2^{n-i-j} - 2^{n-g-j})(1 + k)^2$$

$$+ \sum_{j=1}^{n-g}\binom{n-g}{j}((2^{n-g-j} - 1) + (2^{n-j} - 2^{n-g-j})(1 + k))$$

$$= (1 + k)(1 + w)^u\sum_{i=0}^{g}\binom{g}{i}(((1 + k)2^{n-i} - k2^{n-g})(3/2)^{n-g} - 2^{n-g})$$

$$- k\sum_{j=0}^{n-g}\binom{n-g}{j}(((1 + k)2^n - k2^{n-g})(1/2)^j - 1) - ((1 + k)2^n - k2^{n-g} - 1)$$

$$\leq (1 + k)^2(1 + w)^u 3^n \qquad \square$$

PROOF. *(Theorem 8.4)*. The proof is similar to the proof for Theorem 6.8, except that we store at most two plans corresponding to the join of a distinct set of relations. Therefore, an upper bound on the number of distinct tags associated with any plan is two. Thus we can replace $(1 + w)^u$ in the proof for Theorem 6.8 by two. So the number of enumerations is upper-bounded by

$$E_C = \sum_{i=1}^{g}\binom{g}{i}2\sum_{j=0}^{n-g}\binom{n-g}{j}((n - g - j)(1 + k) + (g - i)(1 + k)(1 + w))$$

$$+ \sum_{j=1}^{n-g}\binom{n-g}{j}((n - g - j) + g(1 + w))$$

$$= 2(1 + k)(1 + gw/n)n2^{n-1}$$

$$- (1 + 2k)(n + g + 2gw)2^{n-g-1} - (n + gw)$$

$$\leq 2(1 + k)(1 + gw/n)n2^{n-1} \qquad \square$$

ACKNOWLEDGMENTS

We are indebted to Joe Hellerstein for giving us detailed feedback on our draft. We also thank the anonymous referees for many constructive comments that helped improve the draft. Without the support of Debjani Chaudhuri and Yesook Shim, it would have been impossible to complete this work.

REFERENCES

CHAUDHURI, S. AND SHIM, K. 1993. Query optimization in the presence of foreign functions. In *Proceedings of the 19th International Conference on Very Large Data Bases* (VLDB '93, Dublin, Ireland, Aug.). Morgan Kaufmann Publishers Inc., San Francisco, CA, 526–541.

CHAUDHURI, S. AND SHIM, K. 1994. Including group-by in query optimization. In *Proceedings of the 20th International Conference on Very Large Data Bases* (VLDB'94, Santiago, Chile, Sept.). VLDB Endowment, Berkeley, CA.

CHAUDHURI, S. AND SHIM, K. 1996. Optimization with user-defined predicates. In *Proceedings of the 22nd International Conference on Very Large Data Bases* (VLDB '96, Mumbai, India, Sept.).

CHIMENTI, D., GAMBOA, R., AND KRISHNAMURTHY, R. 1989. Towards an open architecture for LDL. In *Proceedings of the 15th International Conference on Very Large Data Bases* (VLDB '89, Amsterdam, The Netherlands, Aug 22–25), R. P. van de Riet, Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, 195–203.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

GANGULY, S., HASAN, W., AND KRISHNAMURTHY, R. 1992. Query optimization for parallel execution. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (SIGMOD '92, San Diego, CA, June 2–5), M. Stonebraker, Ed. ACM Press, New York, NY, 9–18.

GRAEFE, G. AND DEWITT, D. J. 1987. The exodus optimizer generator. In *Proceedings of the ACM SIGMOD Annual Conference on Management of Data* (SIGMOD '87, San Francisco, CA, May 27–29), U. Dayal, Ed. ACM Press, New York, NY, 160–172.

GRAEFE, G. AND MCKENNA, W. J. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the 9th International Conference on Data Engineering* (Vienna, Austria, Apr.). IEEE Computer Society, Washington, DC, 209–218.

HELLERSTEIN, J. M. 1994. Practical predicate placement. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (SIGMOD '94, Minneapolis, MN, May 24–27), R. T. Snodgrass and M. Winslett, Eds. ACM Press, New York, NY, 325–335.

HELLERSTEIN, J. M. 1995. Optimization and execution techniques for queries with expensive methods. Ph.D. Dissertation. University of Wisconsin at Madison, Madison, WI.

HELLERSTEIN, J. M. AND STONEBRAKER, M. 1993. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (SIGMOD '93, Washington, DC, May 26–28), P. Buneman and S. Jajodia, Eds. ACM Press, New York, NY, 267–276.

IOANNIDIS, Y. E. AND KANG, Y. C. 1990. Randomized algorithms for optimizing large join queries. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* (SIGMOD '90, Atlantic City, NJ, May 23–25, 1990), H. Garcia-Molina, Ed. ACM Press, New York, NY, 312–321.

KEMPER, A., MOERKETTE, G., AND STEINBRUNN, M. 1992. Optimizing boolean expressions in object-bases. In *Proceedings of the 18th International Conference on Very Large Data Bases* (Vancouver, B.C., Aug.). VLDB Endowment, Berkeley, CA.

KRISHNAMURTHY, R., BORAL, H., AND ZANIOLO, C. 1986. Optimization of nonrecursive queries. In *Proceedings of the 12th International Conference on Very Large Data Bases* (Kyoto, Japan, Aug.). VLDB Endowment, Berkeley, CA, 128–137.

LEE, M. K., FREYTAG, J. C., AND LOHMAN, G. M. 1988. Implementing an interpreter for functional rules in a query optimizer. In *Proceedings of the 14th International Conference on*

Very Large Data Bases (Los Angeles, CA). Morgan Kaufmann Publishers Inc., San Francisco, CA, 218–229.

MONMA, C. L. AND SIDNEY, J. 1979. Sequencing with series-parallel precedence constraints. *Math. Oper. Res. 4*, 215–224.

SELINGER, P. G., ASTRAHAN, M. M., LORIE, R. A., AND PRICE, T. G. 1979. Access path selection in a relational database management system. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (SIGMOD '79, Boston, MA, May 30–June 1). ACM Press, New York, NY, 23–34.

SHAPIRO, L. D. 1986. Join processing in database systems with large main memories. *ACM Trans. Database Syst. 11*, 3 (Sept. 1986), 239–264.

SHIM, K. 1993. Advanced query optimization techniques for relational database systems. Ph.D. Dissertation. University of Maryland at College Park, College Park, MD.

WHANG, K.-Y. AND KRISHNAMURTHY, R. 1990. Query optimization in a memory-resident domain relational calculus database system. *ACM Trans. Database Syst. 15*, 1 (Mar. 1990), 67–95.

Received: February 1997; revised: February 1998; accepted: May 1999