Rank-aware Query Optimization *

Ihab F. Ilyas   Rahul Shah   Walid G. Aref   Jeffrey Scott Vitter   Ahmed K. Elmagarmid

Department of Computer Sciences, Purdue University
250 N. University Street
West Lafayette, Indiana, 47907-2066
{ilyas,rahul,aref,jsv,ake}@cs.purdue.edu

ABSTRACT

Ranking is an important property that needs to be fully supported by current relational query engines. Recently, several rank-join query operators have been proposed based on rank aggregation algorithms. Rank-join operators progressively rank the join results while performing the join operation. The new operators have a direct impact on traditional query processing and optimization.

We introduce a rank-aware query optimization framework that fully integrates rank-join operators into relational query engines. The framework is based on extending the System R dynamic programming algorithm in both enumeration and pruning. We define ranking as an interesting property that triggers the generation of rank-aware query plans. Unlike traditional join operators, optimizing for rank-join operators depends on estimating the input cardinality of these operators. We introduce a probabilistic model for estimating the input cardinality, and hence the cost of a rank-join operator. To our knowledge, this paper is the first effort in estimating the needed input size for optimal rank aggregation algorithms. Costing ranking plans, although challenging, is key to the full integration of rank-join operators in real-world query processing engines. We experimentally evaluate our framework by modifying the query optimizer of an open-source database management system. The experiments show the validity of our framework and the accuracy of the proposed estimation model.

1. INTRODUCTION

Emerging applications that depend on ranking queries warrant efficient support of ranking queries in real-world database management systems. Supporting ranking queries gives database systems the ability to efficiently answer Information Retrieval (IR) queries. For many years, combining the advantages of both worlds, databases and information retrieval systems, has been the goal of many researchers. Database systems provide efficient handling of data with

solid integrity and consistency guarantees. On the other hand, IR provides mechanisms for effective retrieval and fuzzy ranking that are more appealing to the user.

One approach toward integrating databases and IR is to introduce IR-style queries as a challenging type of database queries. The new challenge requires several changes that vary from introducing new query language constructs to augmenting the query processing and optimization engines with new query operators. It may also introduce new indexing techniques and other data management challenges. A ranking query (also known as top-k query) is an important type of query that allows for supporting IR-style applications on top of database systems.

In contrast to traditional join queries, the answer to a *top-k join query* is an ordered set of join results according to some provided function that combines the orders of each input. The following query (Query **Q1**) is an example of a top-k join query expressed in `SQL99`.

```
Q1:    WITH RankedABC as (
         SELECT A.c1 as x ,B.c2 as y, rank() OVER
         (ORDER BY (0.3*A.c1+0.7*B.c2)) as rank
         FROM A,B,C
         WHERE A.c1 = B.c1 and B.c2 = C.c2)
       SELECT x,y,rank
       FROM RankedABC
       WHERE rank <=5;
```

where `A`, `B` and `C` are three relations and `A.c1`,`B.c1`,`B.c2` and `C.c2` are attributes of these relations. The only way to produce ranked results on the expression `0.3*A.c1+0.7*B.c2` in **Q1** is by using a sort operator on top of the join.

Efficient processing of ranking queries gained the attention of many researchers and database vendors. For example, strategies for answering top-k selection queries over relational databases have been introduced in [3] and were prototyped on top of Microsoft SQL Server. In [3], top-k selection queries are mapped to range queries with an adaptable range parameter to produce the top-k results. We further discuss related work on optimizing this implementation of top-k queries in Section 6. Other techniques that maintain materialized views or special indexes to enhance the pereresponse time of top-k queries are introduced in [8, 22, 29].

Although these techniques enhance the database system performance in answering top-k queries, they are implemented either at the application level or outside the core query engine. Hence, processing and optimizing top-k queries lose the benefit of true integration with other basic database query types. A more promising approach is to devise new core query operators that are *rank-aware* and that
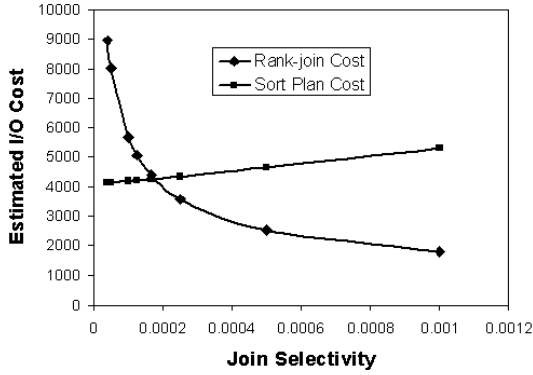
Figure 1: Estimated I/O Cost for Two Ranking Plans.

can be easily integrated in current query engines. Backed with many algorithms for rank aggregation [13, 14, 27, 17, 18], the new query operators combine both algorithmic efficiency (and optimality) and system practicality. Rank-join operators, introduced in [23, 24, 26], progressively rank the join results according to a given scoring function. The join operation has an *early out* condition that stops the operation once the top-ranked join results can be reported. Top-k join query evaluation was also introduced in [1] in querying XML data for the special case of left-outer joins.

1.1 Motivation

For the new rank operators to be practically useful they must be integrated in real-world query optimizers. Top-k queries often involve other query operations such as join, selection and grouping. A key challenge is how to choose a query execution plan that uses the new rank-join operators most efficiently.

An observation that motivates the need for integrating rank-join operators in query optimizers, is that a rank-join operator may not always be the best way to produce the required ranked results. In fact, depending on many parameters (for example, the join selectivity, the available access paths and the memory size) a traditional join-then-sort plan may be a better way to produce the ranked results.

Figure 1 gives the estimated I/O cost of two plans: a *sort plan* and a *rank-join plan*, for various values of the join selectivity. The sort plan is a traditional plan that joins two inputs and sorts the results on the given scoring function, while the rank-join plan uses a rank-join operator that progressively produces the join results ranked on the scoring function. The figure shows that for low values of the join selectivity, the traditional sort-plan is cheaper than the rank-join plan. On the other hand, for higher selectivity values, the rank-join plan is cheaper.

The previous example highlights the need to optimize top-k queries by integrating rank-join operators in query optimization. This approach, although appealing and intuitive, is hindered by the following challenges:

- How to generate plans that make use of rank-join operators? What will be the plan property that triggers the generation of such plans?
- How to estimate the cost of a rank-join query operator? What will be the value of k when pushed all the

way down in the query pipeline? What is the effect of other operators in the plan on the cost estimation?

Another way to phrase the first set of questions is how to make the query optimizer "aware" of the newly available ranking operators and their unique properties. Throwing these operators as yet another join implementation would not work without defining new physical properties that guarantee the best use of these operators.

Unlike traditional query operators, it is hard to estimate the cost of rank-join operators because of their "early out" feature; whenever the top k results are reported, the execution stops without consuming all the inputs. The "early out" feature poses many challenges in costing rank-join operators.

In this paper, we show how to generate the rank-join plan as an alternative execution plan to answer top-k queries. We also show how we came up with the cost estimation of the rank-join plan used in Figure 1, for effective query optimization.

1.2 Contribution

In this paper, we introduce a framework that extends traditional query optimization to be "rank-aware". We can summarize our contributions as follows:

- We extend the notion of interesting properties in query optimization to include *interesting rank expressions*. The extension triggers the generation of a space of rank-aware query execution plans. The new generated plans make use of the proposed rank-join operators and integrate them with other traditional query operators.

- We tackle the challenge of pruning rank-aware execution plans based on cost. Since a rank-join plan can stop at any time once the top-k answers are produced, the input cardinality of the operators (and hence the cost) can vary significantly and in many cases depends on the query itself. We provide an efficient probabilistic model for estimating the minimum input size (depth) needed by rank-join operators. We use the estimation in pruning the generated plans.

- We experimentally validate our probabilistic estimation of the input cardinality of rank-join operators. We show how we use the model to prune the space of generated plans and ultimately, in choosing the overall query execution plan.

The work introduced in this paper completes the picture of a full integration between IR-style rank aggregation algorithms and current relational query processing. We believe that this integration is the first in a series of extensions and modifications to current database management systems to efficiently support IR-style queries.

The remainder of the paper is organized as follows. Section 2 gives the necessary background on rank aggregation algorithms, rank-join operators and cost-based query optimization. We show how to extend traditional query optimization to be rank-aware in Section 3. Moreover, in Section 3, we show how to treat ranking as an interesting physical property and its impact on plan enumeration. In Section 4, we introduce a novel probabilistic model for estimating the input size (depth) of rank-join operators and hence estimating the cost and space complexity of these operators.

In Section 5, we experimentally verify the proposed estimation model and show the accuracy of estimating the input size and the maximum buffer size needed by rank-join operators. We discuss related work in Section 6 and conclude in Section 7 by a summary and final remarks.

2. BACKGROUND

For the paper to be self-contained, we give an overview of relevant techniques we rely on in this work. First, we briefly overview rank aggregation methods as an efficient approach to evaluate top-k queries. We highlight the underlying concept of most rank aggregation algorithms proposed in the literature. Second, we overview rank-join operators, an efficient implementation of rank aggregation algorithms in terms of physical join operators. Third, we give an overview of cost-based optimization as a practical optimization method used by current real-world database systems. Specifically, we briefly describe the dynamic programming optimization framework introduced in System R [28] as the basic optimization technique. We would like to emphasize that the ideas introduced in this paper can be applied to other cost-based optimization frameworks. Choosing the dynamic programming technique is merely for demonstrating the applicability of our approach.

2.1 Rank Aggregation

Rank aggregation is an efficient way to produce a global rank from multiple input rankings. The problem goes back to at least a couple of centuries in effort to come up with a "robust" voting technique [10, 2]. Rank aggregation can be achieved through various techniques. The simplest technique is positional ranking or Borda's method [2], since it is easy to compute in linear time and enjoys some nice properties such as consistency [12]. In a nut-shell, rank aggregation algorithms view the database as multiple lists. Each list contains a ranking of some objects; each object in a list is assigned a score that determines its rank within the list. The goal is to be more efficient than the naïve approach of joining the lists together, and then sorting the output list on the combined score. To get a total ranking, a rank aggregation algorithm incrementally maintains a temporary state that contains all "seen" object scores. The algorithm retrieves objects from the lists (along with their scores) until the algorithm has "enough" information to decide on the top ranked objects, and then terminates. The reader is referred to [13, 14, 27, 17, 18, 26, 4, 24, 21, 7] for more details on the various proposed algorithms.

In general, the proposed rank aggregation algorithms can be classified according to two major criteria. The first classification is based on the type of access available on the input lists. Each ranked input can support sorted access and/or random access. Sorted access enables object retrieval in a descending order of their scores. Random access enables probing or querying an input to retrieve a score of a given object. For example, the NRA algorithm introduced in [14] assumes only sorted access on the ranked inputs, while the TA algorithm, introduced also in [14], assumes the availability of both random access and sorted access on all inputs. On the other hand, the algorithms introduced in [4, 7] assume that at least one source has sorted access capability while other sources can have only random access (probing) available.

The second classification of rank aggregation algorithms

is based on the assumptions on the underlying ranked objects. In the first category, all inputs contain the same set of objects ranked on different criteria. Hence, all the inputs can be viewed as one list of objects, where each object has a set of score attributes. The output is the same set of objects ranked on a combination (aggregation) of these score attributes. We refer to this problem as *top-k selection*. Most of the proposed algorithms belong to this category, e.g., [13, 14, 27, 17, 18]. In the second category of algorithms, e.g., [26, 24], each input contains a different set of objects. A "join" condition among objects in different inputs joins them together in one output join result. Each join result has a combined score computed from the scores of participating objects. The goal is to produce the top-k join results. We refer to this problem as *top-k join*.

2.2 Rank-Join Query Operators

To support rank aggregation algorithms in a database system, we have the choice of implementing these algorithms at the application level as user-defined functions or to implement them as core query operators (rank-join operators). Although the latter approach requires more effort in changing the core implementation of the query engine, it supports ranking as a basic database functionality. The authors in [23, 24], show the benefit of having rank-aware query operators that can be smoothly integrated with other operators in query execution plans. In general, rank-join query operators are physical join operators that, besides joining the inputs, they produce the join results ranked according to a provided scoring function.

Rank-join operators require the following: (1) the inputs (or at least one of them) are ranked and each input tuple has an associated score, (2) the input may not be materialized, but a *GetNext* interface on the input should retrieve the next tuple in a descending order of the associated scores, and (3) there is a *monotone* scoring function, say f, that computes a total score of the join result by applying f on the scores of the tuples from each input.

Rank-join operators are *almost non-blocking*. The next ranked join result is usually produced in a pipelined fashion without the need to exhaust all the inputs. On the other hand, a rank-join operator may need to exhaust part of the inputs before being able to report the next ranked join result.

It is proved that rank-join operators can achieve a huge benefit over the traditional join-then-sort approach to answer top-k join queries especially for small values of k.

For clarity of the presentation, we give a brief overview on two possible rank-join implementations: *nested-loops rank-join* (NRJN) and *hash rank-join* (HRJN). For any rank-join operator, the operator is initialized by specifying the two inputs, the join condition, and the combining function.

HRJN can be viewed as a variant of the *symmetrical hash join* algorithm [20, 30] or the hash ripple join algorithm [19]. HRJN maintains an internal state that consists of three structures. The first two structures are two hash tables, i.e., one for each input. The hash tables hold input tuples seen so far and are used in order to compute the valid join results. The third structure is a priority queue that holds the valid join combinations ordered on their combined score.

At the core of HRJN is the rank aggregation algorithm. The algorithm maintains a *threshold* value that gives an upper-bound of the scores of all join combinations not yet

seen. To compute the threshold, the algorithm remembers and maintains the two top scores and the two bottom scores (last scores seen) of its inputs. A join result is reported as the next top-k answer if the join result has a combined score greater than or equal the threshold value. Otherwise, the algorithm continues by reading tuples from the left and right inputs and performs a symmetric hash join to generate new join results. In each step, the algorithm decides which input to poll depending on different strategies (e.g., depending on the score distribution of each input).

NRJN is similar to HRJN except that it follows a nested-loops strategy to generate valid join results. NRJN internal states contains only a priority queue of all seen join combinations. Similar to HRJN, NRJN maintains a threshold that upper-bounds the scores of all unseen join results.

The *depth* of a rank-join operator is defined as the number of input tuples needed to produce top-k join results. We will elaborate more on integrating rank-join operators in query optimization in the following sections.

2.3 Cost-Based Query Optimization

The optimizer is the component in the query processing engine that transforms a parsed input query into an efficient query execution plan. The execution plan is then passed to the run-time engine for evaluation. The task of a query optimizer is to find the best execution plan for a given query. This task is usually accomplished by examining a large space of possible execution plans and comparing these plans according to their "estimated" execution cost. To estimate the cost of an execution plan, the optimizer adopts a cost model that takes several inputs, such as the estimated input size and the estimated selectivity of the individual operations, and estimates the total execution cost of the query. Most of the different generated plans come from different organizations of the join operations. In general, the larger the space of possible plan, the higher the chance that the optimizer will get a better execution plan.

Plan Enumeration Using Dynamic Programming Since the join operation is implemented in most systems as a diadic (2-way) operator, the optimizer must generate plans that transform an n-way join into a sequence of 2-way joins using binary join operators. The two most important tasks of an optimizer are to find the optimal join sequence as well as the optimal join method for each binary join. Dynamic programming (DP) was first used for join enumeration in System R [28]. The essence of the DP approach is based on the assumption that the cost model satisfies the *principle of optimality*, i.e., the subplans of an optimal plan must be optimal themselves. Therefore, in order to obtain an optimal plan for a query joining n tables, it suffices to consider only the optimal plans for all pairs of non-overlapping m tables and $n - m$ tables, for $m = 1, 2, ..., n - 1$.

To avoid generating redundant plans, DP maintains a memory-resident structure (referred to as *MEMO*, following the terminology used in [15]) for holding non-pruned plans. Each MEMO entry corresponds to a subset of the tables (and applicable predicates) in the query. The algorithm runs in a bottom-up fashion by first generating plans for single tables. Then it enumerates joins of two tables, then three tables, etc., until all n tables are joined. For each join it considers, the algorithm generates join plans and incorporates them into the plan list of the correspond-
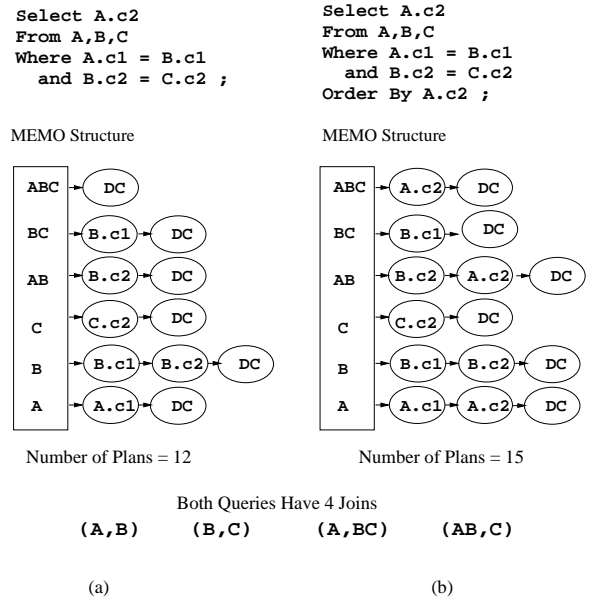

Figure 2: Number of Joins vs. Number of Plans.

ing MEMO entry. Plans with larger table sets are built from plans with smaller table sets. The algorithm prunes a higher cost plan if there is a cheaper plan with the same or more general properties for the same MEMO entry. Finally, the cheapest plan joining n tables is returned.

Plan Properties Such properties are extensions of the important concept of *interesting orders* [28] introduced in System R. Suppose that we have two plans generated for table R, one produces results ordered on $R.a$ (call it P_1) and the other does not produce any ordering (call it P_2). Also suppose that P_1 is more expensive than P_2. Normally, P_1 should be pruned by P_2. However, if table R can later be joined with table S on attribute a, P_1 can actually make the sort-merge join between the two tables cheaper than P_2 since it doesn't have to sort R. To avoid pruning P_1, System R identifies orders of tuples that are potentially beneficial to subsequent operations for that query (hence the name interesting orders), and compares two plans only if they represent the same expression and have the same interesting order. In Figure 2(a), we show a 3-way join query and the plans kept in the corresponding MEMO structure. For each MEMO entry, a list of plans is stored, each carrying a different order property that is still interesting. We use DC to represent a "don't care" property value, which corresponds to "no order". The cheapest plan with a DC property value is also stored in each MEMO entry if this plan is cheaper than any other plan with interesting orders. Modifying the query to that in Figure 2(b), by adding an *orderby* clause, increases the number of interesting order properties that need to be kept in all MEMO entries containing A. By comparing Figure 2(a) with Figure 2(b), we can see that the number of generated join plans changes, even though the join graph is still the same. The idea of interesting orders was later generalized to multiple physical properties in [16, 25] and is used extensively in modern optimizers. Intuitively, a physical property is a characteristic of a plan that is not shared by all plans for the same logical expression (corresponding to a MEMO entry), but can impact the cost of subsequent operations.

3. RANK-AWARE OPTIMIZATION

Having described an efficient implementation of the rank aggregation algorithms in terms of new rank-join query operators, we now describe how to extend the traditional query optimization–one that uses dynamic programming a la [28]– to handle the new rank-join operators. Integrating the new rank-join operators in the query optimizer includes two major tasks:(1) enlarging the space of possible plans to include those plans that use rank-join operators as a possible join alternative, and (2) providing a costing mechanism for the new operators to help the optimizer prune expensive plans in favor of more general cheaper plans.

In this section, we elaborate on the first task while in the following section we provide an efficient costing mechanism for rank-join operators. Enlarging the plan space is achieved by extending the enumeration algorithm to produce new execution plans. The extension must conform to the enumeration mechanism of other traditional plans. In this work, we choose the *DP* enumeration technique, described in Section 2. The *DP* enumeration is one of the most important and widely used enumeration techniques in commercial database systems. Current systems use different flavors of the original *DP* algorithm that involve heuristics to limit the enumeration space and can vary in the way the algorithm is applied (e.g., bottom-up versus top-down). In this paper, we stick to the bottom-up *DP* as originally described in [28]. Our approach is equally applicable to other enumeration algorithms.

3.1 Ranking as an Interesting Property

As described in Section 2.3, interesting orders are those orders that can be beneficial to later operations. Practically, interesting orders are collected from: (1) columns in equality predicates in the join condition, as orders on these columns make upcoming sort-merge operations much cheaper by avoiding the sort, (2) columns in the *groupby* clause to avoid sorting in implementing sort-based grouping, and (3) columns in the *orderby* clause since they must be enforced on the final answers. Current optimizers usually enforce interesting orders in an *eager* fashion. In the eager policy, the optimizer generates plans that produce the interesting order even if they do not exist naturally (e.g., through the existence of an index).

In the following example, we describe a top-k query using current `SQL` constructs by specifying the ranking function in the *orderby* clause.

```
Q2:
WITH RankedABC as (
  SELECT A.c1 as x ,B.c1 as y, C.c1 as z, rank() OVER
  (ORDER BY (0.3*A.c1+0.3*B.c1+0.3*C.c1)) as rank
  FROM A,B,C
  WHERE A.c2 = B.c1 and B.c2 = C.c2)
SELECT x,y,z,rank
FROM RankedABC
WHERE rank <=5;
```

where `A`, `B` and `C` are three relations and `A.c1`, `A.c2`, `B.c1`, `B.c2`, `C.c1` and `C.c2` are attributes of these relations. Following the concept of *interesting orders*, the optimizer considers orders on `A.c2`, `B.c1`, `B.c2` and `C.c2` as interesting orders (because of the join) and *eagerly* enforces the existence of plans that access `A`, `B` and `C` ordered on `A.c2`, `B.c1`, `B.c2` and `C.c2`, respectively. This enforcement can be done by *gluing* a sort operator on top of the table scan or

Interesting Order Expressions	Reason
`A.c1`	Rank-join
`A.c2`	Join
`B.c1`	Join and Rank-join
`B.c2`	Join
`C.c1`	Rank-join
`C.c2`	Join
`0.3*A.c1+0.3*B.c1`	Rank-join
`0.3*B.c2+0.3*C.c2`	Rank-join
`0.3*A.c1+0.3*C.c2`	Rank-join
`0.3*A.c1+0.3*B.c2+0.3*C.c2`	Orderby

Table 1: Interesting Order Expressions in Query Q2.

by using an available index that produces the required order. Currently, orders on `A.c1` or `C.c1` are "not interesting" since they are not beneficial to other operations such as a sort-merge join or a sort. The reason being that a sort on the expression (`0.3*A.c1+0.3*B.c1+0.3*C.c1`) cannot benefit from ordering the input on `A.c1` or `C.c2` individually.

Having the new rank-aware physical join operators, orderings on the individual scores (for each input relation) become *interesting* in themselves. In the previous example, an ordering on `A.c1` is interesting because it can serve as input to a rank-join operator. Hence, we extend the notion of interesting orders to include those attributes that appear in the ranking function.

> DEFINITION 1. *An Interesting Order Expression is ordering the intermediate results on an* expression *of database columns that can be beneficial to later query operations.*

In the previous example, we can identify some interesting order expressions according to the previous definition. We summarize these orders in Table 1. Like an ordinary interesting order, an interesting order expression *retires* when it is used by some operation and is no longer useful for later operations. In the previous example, an order on `A.c1` is no longer useful after a rank-join between table `A` and `B`.

3.2 Extending the Enumeration Space

In this section, we show how to extend the enumeration space to generate rank-aware query execution plans. Rank-aware plans will integrate the rank-join operators, described in Section 2.2, into general execution plans. The idea is to devise a set of rules that generate rank-aware join choices at each step of the *DP* enumeration algorithm. For example, on the table access level, since interesting orders now contain ranking score attributes, the optimizer will enforce the generation of table and index access paths that satisfy these orders. In enumerating plans at higher levels (join plans), these ordered access paths will make it feasible to use rank-join operators as join *choices*.

For a query with n input relations, T_1 to T_n, assume there exists a ranking function $f(s_1, s_2, \ldots, s_n)$, where s_i is score expression on relation T_i. For two sets of input relations, L and R, we extend the space of plans that join L and R to include rank-join plans by adapting the following:

- *Join Eligibility* L and R are rank-join-eligible if all the following apply:

  1. There is a join condition that relates at least one input relation in L to an input relation in R.
  2. f can be expressed as $f(f_1(S_L), f_2(S_R), f_3(S_O))$, where f_1, f_2 and f_3 are three scoring functions, S_L are the score expressions on the relations in
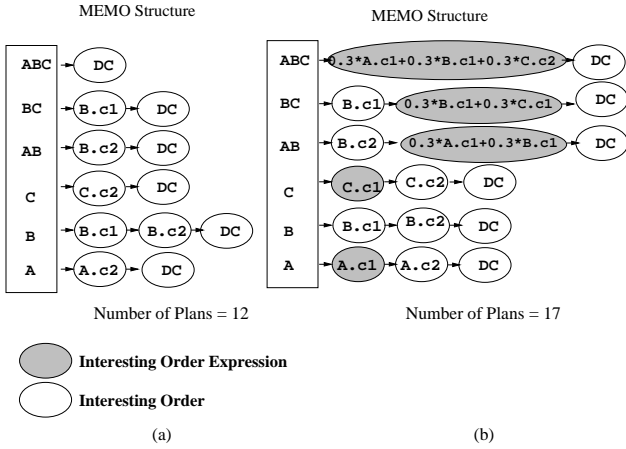
Figure 3: Enumerating Rank-aware Query Plans.

L, S_R are the score expressions on the relations in R, and S_O are the score expressions on the rest of the input relations.

3. There is at least one plan that accesses L and/or R ordered on S_L and/or S_R, respectively.

- *Join Choices* Rank-join can have several implementations as physical join operators. In Section 2.2, we presented the hash rank-join operators (HRJN) and the nested-loops rank-join operator (NRJN). For each rank-join between L and R, plans can be generated for each join implementation. For example, an HRJN plan is generated if there exist plans that access both L and R sorted on S_L and S_R, respectively. On the other hand, an NRJN plan is generated if there exists at least one plan that accesses L *or* R sorted on S_L or S_R, respectively.

- *Join Order* For symmetric rank-join operators (e.g., HRJN), there is no distinction between outer and inner relations. For the nested-loops implementation, a different plan can be generated by switching the inner and the outer relations. L (R) can serve as inner to an NRJN operator if there exists a plan that accesses L (R) sorted on S_L (S_R).

For example, for Query **Q2** in Section 3.1, new plans are generated by enforcing the interesting order expressions listed in Table 1 and using all join choices available including the rank-join operators. As in traditional *DP* enumeration, generated plans are pruned according to their cost and properties. For each class of properties, the cheapest plan is kept. Figure 3 gives the MEMO structure of the retained subplans when optimizing **Q2**. Each oval in the figure represents the best plan with a specific order property. Figure 3 (a) gives the MEMO structure for the traditional application of the *DP* enumeration without the proposed extension. For example, we keep two plans for Table **A**; the cheapest plan that does not have any order property (**DC**) and the cheapest plan that produces results ordered on **A.c2** as an interesting order. Figure 3 (b) shows the newly generated classes of plans that preserve the required ranking. For each interesting order expression, the cheapest plan that produces that order is retained. For example, in generating plans that join Tables **A** and **B**, we keep the cheapest plan that produces results ordered on `0.3*A.c1 + 0.3*B.c1`.
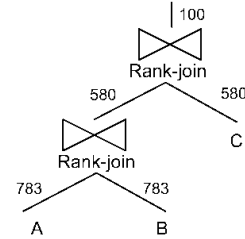

Figure 4: Example Rank-join Plan.

3.3 Pruning Plans

A subplan P_1 is pruned in favor of subplan P_2 if and only if P_1 has both higher cost and weaker properties than P_2. In Section 3.2, we discussed extending the interesting order property to generate rank-aware plans. A key property of top-k queries is that users are interested only in the first k results and not in a total ranking of all query results. This property directly impacts the optimization of top-k queries by optimizing for the first k results. Traditionally, most real-world database systems offer the feature of *First-N-Rows-Optimization*. Users can turn on this feature when desiring fast response time to receive results as soon as they are generated. This feature translates into respecting the "pipelining" of a plan as a physical plan property. For example, for two plans P_1 and P_2 with the same physical properties, if P_1 is a pipelined plan (e.g., nested-loops join plan) and P_2 is a non-pipelined plan (e.g., sort-merge join plan), P_1 cannot be pruned in favor of P_2, even if P_2 is cheaper than P_1.

In real-world query optimizers, the cost model for different query operators is quite complex and depends on many parameters. Parameters include cardinality of the inputs, available buffers, type of access paths (e.g., a clustered index) and many other system parameters. Although cost models can be very complex, a key ingredient of accurate estimation is the accuracy of estimating the size of intermediate results.

In traditional join operators, the input cardinalities are independent of the operator itself and only depend on the input subplan. Moreover, the output cardinality depends only on the size of the inputs and the selectivity of the logical operation. On the other hand, since a rank-join operator does not consume all of its inputs, the actual input size depends on the operator itself and how the operator decides that it has seen "enough" information from the inputs to generate the top k results. Hence, the input cardinality depends on the number of ranked join results requested from that operator. Thus, the cost of a rank-join operator depends on the following:

- *The number of required results k and how k is propagated in the pipeline.* For example, Figure 4 gives a real similarity query that uses two rank-join operators to combine the ranking based on three features, referred to as A, B and C. To get 100 requested results (i.e., $k = 100$), the top operator has to retrieve 580 tuples from each of its inputs. Thus, the number of required results from the child operator is 580 in which it has to retrieve 783 tuples from its inputs. Notice that while $k = 100$ in the top rank-join operator, $k = 580$ in the child rank-join operator that joins A and B. In other words, in a pipeline of rank-join operators, the input depth of a rank-join operator is the required number
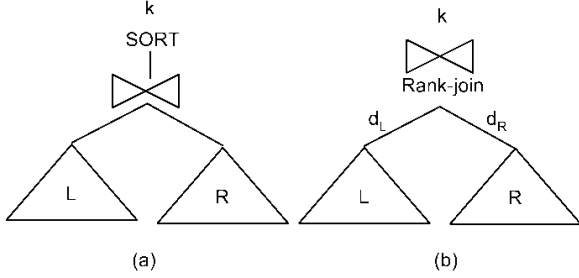
Figure 5: Two Enumerated Plans.

of ranked results from the child rank-join operator.

- *The number of tuples from inputs that contain enough information for the operator to report the required number of answers, k.* In the previous example, the top operator needs 580 tuples from both inputs to report 100 rankings, while the child operator needed 783 tuples from both inputs to report the required 580 partial rankings.

- *The selectivity of the join operation.* The selectivity of the join affects the number of tuples propagated from the inputs to higher operators through the join operation. Hence, the join selectivity affects the number of input tuples required by the rank-join operator to produce ranked results.

There are two ways to produce plans that join two sets of input relations, L and R, and produce ranked results: (1) by using rank-join operators to join L and R subplans, or (2) by gluing a sort operator on the cheapest join plan that joins L and R without preserving the required order. One challenge is in comparing two plans when one or both of them are rank-join plans. For example, in the two plans depicted in Figure 5, both plans produce the same order property. Plan (b) may or may not be pipelined depending on the subplans of L and R. In all cases, the cost of the two plans need to compared to decide on pruning. While the current traditional cost model can give an estimate total cost of Plan (a), it is hard to estimate the cost of Plan (b) because of its strong dependency on the number of required ranked results, k. Thus, to estimate the cost of Plan (b), we need to estimate the propagation of the value of k in the pipeline (refer to Figure 4). In Section 4, we give a probabilistic model to estimate the depths (d_L and d_R in Figure 5 (b)) required by a rank-join operator to generate top k ranked results. The estimate for the depths is parameterized by k and by the selectivity of the join operation. It is important to note that the cost of Plan (a) is (almost) independent of the number of output tuples pulled from the plan since it is a blocking sort plan. In Plan (b), the number of required output tuples determines how many tuples will be retrieved from the inputs and that greatly affects the plan cost.

Plan Pruning According to our enumeration mechanism, at any level, there will be only one plan similar to Plan (a) of Figure 5 (by gluing a sort on the cheapest non-ranking plan). At the same time, they may be many plans similar to Plan (b) of Figure 5 (e.g., by changing the type of the rank-join operator or the join order).

For all rank-join plans, the cost of the plan depends on k and the join selectivity s. Since these two parameters are the same for all plans, the pruning among these plans follows the same mechanism as in traditional cost based pruning. For example, pruning a rank-join plan in favor of another rank-join plan depends on the input cardinality of the relations, the cost of the join method, the access paths, and the statistics available on the input scores.

We assume the availability of an estimate of the join selectivity, which is the same for both sort-plans and rank-join plans. A challenging question is how to compare between the cost of a rank-join plan and the cost of a sort plan, e.g., Plans (a) and (b) in Figure 5, when the number of required ranked results is unknown. Note that the number of results, k, is known only for the final complete plan. Because subplans are built in a bottom-up fashion, the propagation of the final k value to a specific subplan depends on the location of that subplan in the complete evaluation plan.

We introduce a mechanism for comparing the two plans in Figure 5 using the estimated total cost of Plan (a) and the estimated cost of Plan (b), parametrized by k. Section 4 describes how to obtain the parametrized cost of Plan (b). For Plan (a), we can safely assume that $Cost_a(k) = TotalCost_a$ where $Cost_a(k)$ is the cost to report k results from Plan (a), and $TotalCost_a$ is the cost to report all join results of Plan (a). This assumption follows directly from Plan (a) being a blocking sort plan. Let k^* be that value of k at which the cost of the two plans are equal. Hence, $Cost_a(k^*) = Cost_b(k^*) = TotalCost_a$. The output cardinality of Plan (a) (call it n_a) can be estimated as the product of the cardinalities of all inputs multiplied by the estimated join selectivity. Since k cannot be more than n_a, we compare k^* with n_a. Let k_{min} be the minimum value of k for any rank-join subplan. A reasonable value for k_{min} would be the value specified in the query as the total number of required answers. Consider the following cases:

- $k^* > n_a$: Plan (b) is always cheaper than Plan (a). Hence Plan (a) should be pruned in favor of Plan (b).

- $k^* < n_a$ and $k^* < k_{min}$: Since for any subplan, $k \geq k_{min}$, we know that we will require more that $k*$ output results from Plan (b). In that case Plan (a) is cheaper. Depending on the nature of Plan (b) we decide on pruning:

  - If Plan (b) is a pipelined plan (e.g., a left deep tree of rank-join operators), then we cannot prune Plan (b) in favor of Plan (a) since it has more properties, the pipelining property.
  - If Plan (b) is not a pipelined tree, then Plan (b) is pruned in favor of Plan (a).

- $k^* < n_a$ and $k^* > k$: We keep both plans since depending on k, Plan (a) may be cheaper than Plan (b) and hence cannot be pruned.

As an example, we show how the value of k affects the cost of rank-join plans and hence the plan pruning decisions. We compare two plans that produce ranked join results of two inputs. The first plan is a *sort plan* similar to that in Figure 5(a), while the second plan is a *rank-join plan* similar to that in Figure 5(b). The sort plan sorts the join results of an index nested-loops join operator while the rank-join plan uses HRJN as its rank-join operator. The estimated cost formula for the sort plan uses the traditional cost formulas for external sorting and index nested-loops join, while the estimated cost of the rank-join plan is based on our model
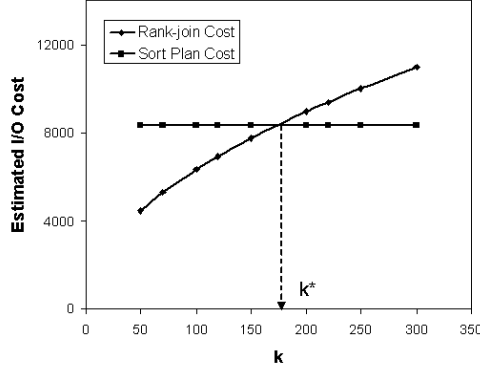
Figure 6: The Effect of k on the Rank-join Cost.

to estimate the input cardinality (as will be shown in Section 4). Both cost estimates use the same values of input relations cardinalities, total memory size, buffer size, and input tuple sizes. Figure 6 compares the estimate of the costs of the two plans for different values of k. While the sort plan cost can be estimated to be independent of k, the cost of the rank-join plan increases with increasing the value of k. In this example, $k^* = 176$.

4. ESTIMATING INPUT CARDINALITY OF RANK-JOIN OPERATORS

In this section, we give a probabilistic model to estimate the input cardinality (depth) of rank-join operators. The estimate is parameterized with k, the number of required answers from the (sub)plan, and s, the selectivity of the join operation. We describe the main idea of the estimation procedure by first considering the simple case of two ranked relations. Then, we generalize to the case of a hierarchy of rank-join operators.

Let L and R be two ranked inputs to a rank-join operator. Let m and n be the table cardinalities of L and R, respectively. Our objective is to get an estimate of depths d_L and d_R (see Figure 9) such that it is sufficient to retrieve only up to d_L and d_R tuples from L and R, respectively, to produce the top k join results. We denote the top i tuples of L and R as $L(i)$ and $R(i)$, respectively. We outline our approach to estimate d_L and d_R in Figure 7.

In the following subsections, we elaborate on steps of the outline in Figure 7. Figure 8 gives Algorithm *Propagate* used by the query optimizer to compute the values of d_L and d_R at all levels in a rank-join plan. We set k to the value specified in the query when we call the algorithm for the final plan.

We assume the following to simplify the analysis: (1) the combining scoring function is a linear combination of the scores (e.g., a weighted sum of the input scores), and (2) each tuple in L is equally likely to join with sn tuples in R and each tuple in R is equally likely to join with sm tuples in L.

4.1 Estimating Any-k Depths

In the first step of the outline in Figure 7, we estimate the depths c_L and c_R in L and R, respectively, required to get any k join results. "Any k" join results are valid join results, but not necessarily among the top k answers in score.

Outline EstimateTop-kDepth
INPUT: Two ranked relations L and R
The number of required ranked results, k
The join selectivity, s
Any-k Depths
1. Compute the value of c_L and c_R, where
   c_L is the depth in L and c_R is the depth in R such that,
   $\exists$ expected k valid join results between $L(c_L)$ and $R(c_R)$

Top-k Depths
2. Compute the value of d_L and d_R, where
   d_L is the depth in L and d_R is the depth in R such that,
   $\exists$ expected k top-scored join results between $L(d_L)$.
   and $R(d_R)$. d_L and d_R are expressed in terms of c_L and c_R.

Minimize Top-k Depths
3. Compute the values of c_L and c_R to minimize d_L and d_R.
   c_L, c_R, d_L and d_R are paramterized by k

Figure 7: Outline of the Estimation Technique.

Algorithm Propagate (subplan P, k)
INPUT: The number of required ranked results, k
The root of a subplan, P
OUTPUT: d_L and d_R for the operator rooted at P
1. Compute d_L and d_R according to the formulas in Section 4.3
2. Call Propagate(left subplan of P, d_L)
3. Call Propagate(right subplan of P, d_R)
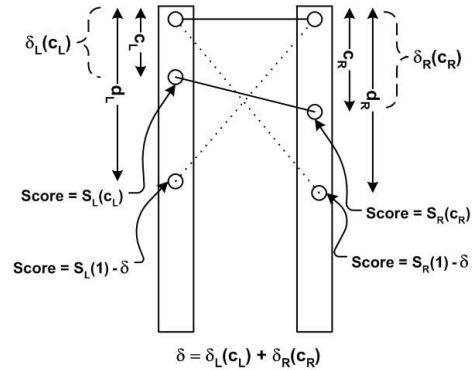
Figure 8: Propagating the Value of k.

Figure 9: Depth Estimation of Rank-join Operators.

THEOREM 1. *If c_L and c_R are chosen such that $sc_Lc_R \geq k$, then the expected number of valid join results between $L(c_L)$ and $R(c_R)$ is $\geq k$.*

PROOF. Let $X_{i,j}$ denote a random variable that is equal to the number of join results produced by joining the first i tuples from L and the first j tuples from R. Since every tuple in L is likely to join with sj tuples in $R(j)$, then the expected value of this random variable is $E[X_{i,j}] = sij$. Let $c_L = i$ and $c_R = j$, hence, if $sc_Lc_R \geq k$, then we can expect at least k valid join results between $L(c_L)$ (the top c_L tuples in L) and $R(c_R)$. $\square$

In general, the choice of c_L and c_R can be arbitrary as long as they satisfy $sc_Lc_R \geq k$. We show that we choose values for c_L and c_R in Section 4.3.

4.2 Estimating Top-k Depths

In the second step in the outline given in Figure 7, we aim at obtaining good estimates for d_L and d_R, where d_L

and d_R are the depths into L and R, respectively, needed to produce an expected number of top k join results. For the simplicity of presentation, the formulas presented in this section assume that the scoring function is the summation of individual scores.

Let $S_L(i)$ and $S_R(i)$ be the scores of the tuples at depth i in L and R, respectively. Moreover, let $\delta_L(i)$ and $\delta_R(i)$ be the score difference between the top ranked tuple and the score of the tuple at a depth i in L and R, respectively, i.e., $\delta_L(i) = S_L(1) - S_L(i)$ and $\delta_R(i) = S_R(1) - S_R(i)$

THEOREM 2. *If there are k valid join results between $L(c_L)$ and $R(c_R)$, and if d_L and d_R are chosen such that $\delta_L(d_L) \geq \delta_L(c_L) + \delta_R(c_R)$ and $\delta_R(d_R) \geq \delta_L(c_L) + \delta_R(c_R)$, then the top k join results can be obtained by joining $L(d_L)$ and $R(d_R)$.*

PROOF. Refer to Figure 9 for illustration. Let $\delta = \delta_L(c_L) + \delta_R(c_R)$ and $S = S_L(1) + S_R(1)$. Since, there are k join tuples between $L(c_L)$ and $R(c_R)$, the final score of each of the join results is $\geq S - \delta$. Consequently, the scores of all of the top k join results are $\geq S - \delta$. Assume that one of the top-k join results, J, joins a tuple t at depth d in L with some tuple in R such that $\delta_L(d) > \delta$. The highest possible score of J is $S_L(d) + S_R(1) = S - \delta_L(d) < S - \delta$. By contradiction, Tuple t cannot participate in any of the top k join results. Hence, any tuple in L (similarly R) that is at a depth $> d_L$ (d_R) cannot participate in the top k join results. □

Since the choice of c_L and c_R can be arbitrary as long as they satisfy the condition in Theorem 1, Step (3) of the outline in Figure 7 chooses the values of c_L and c_R that minimize the values of d_L and d_R. Note that both d_L and d_R are minimized when $\delta = \delta_L(c_L) + \delta_R(c_R)$ is minimized. Hence we minimize δ subject to the constraint $s c_L c_R \geq k$. The rationale behind this minimization is that an optimal rank-aggregation algorithm does not need to retrieve more than the minimum d_L and d_R tuples from L and R, respectively, to generate the top k join results.

4.3 Estimating the Minimum d_L and d_R

Till now, we did not have any assumptions on the score distributions of L and R. We showed that d_L and d_R are related to c_L and c_R in terms of the scores of the tuples at these depths.

To have a closed formula for the minimum d_L and d_R, we assume that the rank scores in L and R are from some uniform distribution. Let x be the average decrement slab of L (i.e., the average difference between the scores of two consecutive ranked objects in L) and let y be the average decrement slab for R. Hence, the expected value of $\delta_L(c_L) = x c_L$ and the expected value of $\delta_R(c_R) = y c_R$. To minimize $\delta = \delta_L(c_L) + \delta_R(c_R)$, we minimize $x c_L + y c_R$, subject to $s c_L c_R \geq k$. The minimization is achieved by setting $c_L = \sqrt{(y k)/(x s)}$ and $c_R = \sqrt{(x k)/(y s)}$. In this case, $d_L = c_L + (y/x) c_R$ and $d_R = c_R + (x/y) c_L$

In the simplistic case, where both the relations come from the same uniform distribution, i.e., $x = y$, then $c_L = c_R = \sqrt{k/s}$ and $d_L = d_R = 2\sqrt{k/s}$.

In a hierarchy of joins, where the output of one rank-join operator serves as input to another operator, the score distributions of the second level join are no longer uniform. Assuming the scoring function is the sum of two scores, the
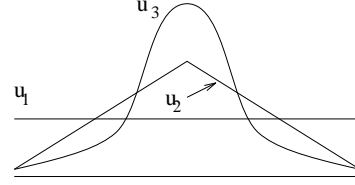
Figure 10: Central Limit Theorem.

scores of rank join with two uniform distributions follows a triangular distribution. As we go higher up in the join hierarchy, the distribution tends to be normal (bell-shaped curve) by central limit theorem (see Figure 10).

Let X, Y be two independent random variables from the uniform distribution $[0, n]$. We refer to this uniform distribution as u_1. We refer to the summation of j independent random variable from u_1 as u_j. The random variable $Z = X + Y$, which follows the distribution u_2, is a triangular distribution over $[0, 2n]$ with a peak at n. If we choose n elements from the u_2 distribution, the score of the ith element ($i \leq n/2$), in a decreasing order of the scores, is expected to be $2n - \sqrt{2in}$. In general, if we choose m elements from u_j, which ranges from $[0, jn]$, then the score of the ith element is expected to be

$$score_i = jn - (j! in^j / m)^{1/j} \qquad (1)$$

Using the described distribution scores, we estimate the values of c_L and c_R that give the minimum values of d_L and d_R for the general rank-join plan in Figure 5 (b). Let the output of L be the output of rank-joining l ranked relations. Let the output of R be the output of rank-joining r ranked relations. Let k be the number of output ranked results required from the subplan, and s be the join selectivity. Then minimizing $\delta = \delta_L(c_L) + \delta_R(c_R)$ amounts to minimizing $\delta = (l! c_L n^{l-1})^{1/l} + (r! c_R n^{r-1})^{1/r}$. We substitute $c_R = \frac{k}{s c_L}$ and minimize δ with respect to c_L. The minimizations yield:

$$c_L^{r+l} = \frac{(r!)^l k^l n^{r-l} l^{rl}}{s^l (l!)^r r^{rl}} \qquad (2)$$

$$c_R^{r+l} = \frac{(l!)^r k^r n^{l-r} r^{rl}}{s^r (r!)^l l^{rl}} \qquad (3)$$

$$d_L = c_L [1 + r/l]^l \qquad (4)$$

$$d_R = c_R [1 + l/r]^r \qquad (5)$$

Note that d_L and d_R are strict upper-bounds assuming worst-case behavior. For an average case analysis, assume that L follows a u_l distribution and R follows a u_r distribution with each having n tuples. The join of L and R produces another relation, G with a u_{l+r} distribution and $s n^2$ tuples. Using Equation 1 and setting $j = l + r$ and $m = s n^2$, the score of the top kth tuple in G is $score_k = (l + r)n - ((l + r)! k n^{l+r-2}/s)^{1/(l+r)}$. Hence, we need to check in L (R) up to a tuple that joins with R (L) to produce $score_k$. We can show that on average, d_L and d_R can be computed as follows:

$$d_L^{l+r} = \frac{((l+r)!)^l k^l n^{r-l}}{(l!)^{l+r} s^l} \quad and \quad d_R^{l+r} = \frac{((l+r)!)^r k^r n^{l-r}}{(r!)^{l+r} s^r}$$

Because the distribution of the depths is tight around the mean, we can apply the formulas recursively in a rank-join plan, as shown in the algorithm in Figure 8, by replacing

k of the left and right subplans by d_L and d_R, respectively. The value of k for the top operator is the value specified by the user in the query.

5. EXPERIMENTAL VERIFICATION OF THE ESTIMATION MODEL

In this section, we experimentally verify the accuracy of our model for estimating the depths (input size) of rank-join operators and estimating an upper-bound of the buffer size maintained by these operators. Estimating the input size and the space requirements of a rank-join operator make it easy to estimate the total cost of a rank-join plan according to any practical cost model.

5.1 Implementation Issues and Setup

All experiments are based on a research platform for a complete video database management system running on a Sun Enterprise 450 with 4 UltraSparc-II processors running SunOS 5.6 operating system. The prototype is built on top of an open-source database management system that allows us to implement a simple cost-based rank-aware optimizer in the query engine (details are omitted for expository reasons). We have implemented a simple *DP* join enumerator that generates all possible rank-join plans in a bottom-up fashion.

In the experiments conducted in this section, the user query provides the system with an example image and requests the most similar video objects (segments or snapshots) to the query image based on multiple visual features. The visual features are extracted from the video data and are stored in separate relations. High-dimensional index access paths are available on these relations to rank the objects according to each of the corresponding features. Example features include color histograms (*ColorHist*), color layout (*ColorLayout*), texture (*Texture*) and edge orientation (*Edges*). Hence, for a multi-feature similarity query, each input ranks the stored video objects according to a single feature. The top-k query produces the k objects with the top combined scores. We use the following top-k query:

Q: *Retrieve the k most similar video shots to a given image based on m visual features.*

In the implemented prototype, we automatically show the generated evaluation (sub)plans at each level of the *DP* algorithm. We only display "templates" of the execution plans. Each of these plan templates generates several evaluation plans by changing the join implementation choices, switching the join order, or gluing sort operators to enforce interesting order properties.

Figure 12 gives a snapshot of the plan generation interface for joining 4 inputs. We focus on the first complete generated plan and annotate it in Figure 11 for easy referencing. We refer to this plan as Plan P.

5.2 Verifying Input Cardinality Estimation

In this experiment, we evaluate the accuracy of the depth estimates of rank-join operators. We conducted several experiments on a variety of example evaluation plans of Query **Q**. Since all experiments show similar behavior, we show a representative sample results for this experiment. The results shown here represent the estimates for Plan P in Figure 11. We use HRJN as the implementation of the rank-join operator. k ranked results are required from the top rank-join operator in the plan.

Varying the Number of Required Answers (k) For different values of k, Figure 13 (a) compares the actual values of d_1 and d_2 (refer to Figure 11) with two estimates:(1) *Any-k Estimate*, the estimated values for d_1 and d_2 to get any k join results (not necessary the top k), and (2) *Top-k Estimate*, the estimated values for d_1 and d_2 to get the top k join results. *Any-k Estimate* and *Top-k Estimate* are computed according to Section 4. The actual values of d_1 and d_2 are obtained by actually running the query and by counting the number of retrieved input tuples by each operator. Figure 13 (b) gives similar results for comparing the actual values of d_5 and d_6 to the same estimates. The figures show that the estimation error is less than 25% of the actual depth values. In general, for all conducted experiments, this estimation error is less than 30% of the actual depth values. Note that the measured values of d_1 and d_2 lie between the *Any-k Estimate* and the *Top-k Estimate*. The *Any-k Estimate* can be considered as a *lower-bound* on the depths required by a rank-join operator.

Varying the Join Selectivity Figure 14 compares the actual and estimated values for the depths of Plan P in Figure 11 for various values of the join selectivity. For low selectivity values, the required depths increase as the rank aggregation algorithm needs to retrieve more tuples from each input to have enough information to produce the top ranked join results. The maximum estimation error is less than 30% of the actual depth values.

5.3 Estimating the Maximum Buffer Size

Rank-join operators usually maintain a buffer of all join results produced and cannot yet be reported as the top k results. Estimating the maximum buffer size is an important parameter in estimating the total cost of a rank-join operator. In this experiment, we use Plan P in Figure 11. The left child rank-join operator in Plan P needs d_1 and d_2 tuples from its left and right inputs, respectively, before producing the top k results. The worst case (maximum) buffer size occurs when the rank-join operator cannot report any join result before retrieving all the d_1 and d_2 tuples. Hence, an upper bound on the buffer size can be estimated by $d_1 d_2 s$, where s is the join selectivity. We use our estimates for top-k depths, d_1 and d_2, to estimate the upper bound of the buffer size. We compare the actual (measured) buffer size to the following two estimates: (1) *Actual upper-bound*, the upper bound computed using the measured depths d_1 and d_2, and (2) *Estimated upper-bound*, the upper bound computed using our estimation of top-k depths.

Figure 15 shows that the estimated upper-bound has an estimation error less than 40% of the actual upper-bound (computed using the measured values of d_1 and d_2). Figure 15 also shows that the actual buffer size is less than the upper-bound estimates. The reason being that in the average case, the operator progressively reports ranked join results from the buffer before completing the join between the d_1 and d_2 tuples. The gap between the actual buffer size and the upper-bound estimates increases with k, as the probability of the worst-case scenario decreases.

6. RELATED WORK

Another approach to evaluate top-k queries is the filter/restart approach [6, 5, 11, 3]. Ranking is mapped to a filter condition with a cutoff parameter. If the filtering
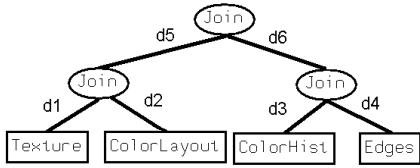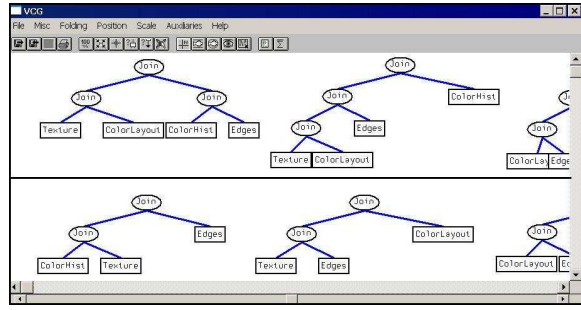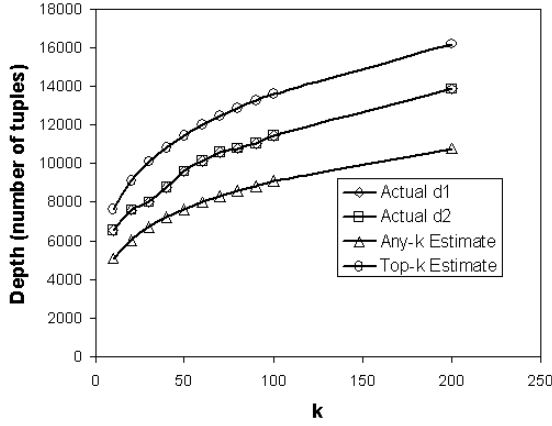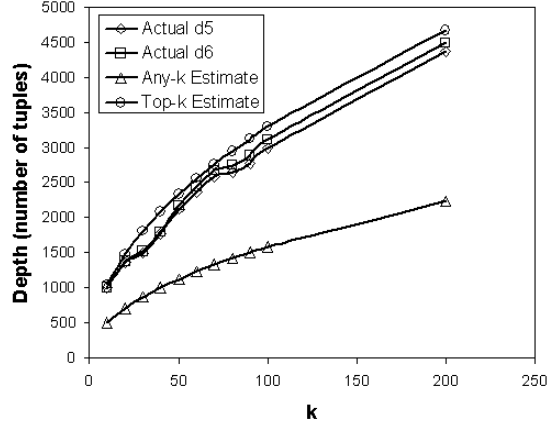
Figure 11: Example Rank-join Plan.
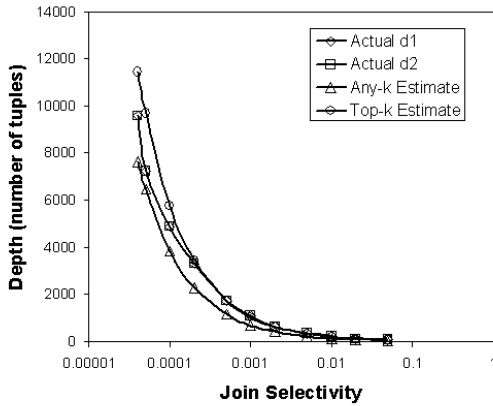

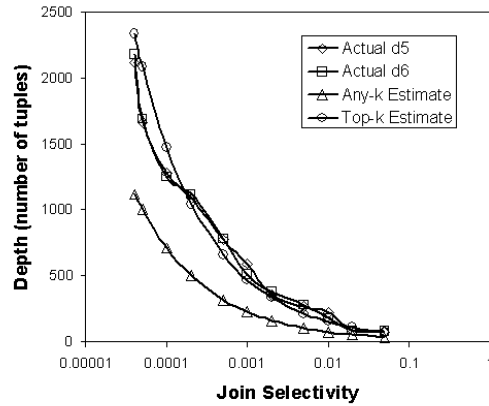Figure 12: A Snapshot of the Plan Generation Interface.

(a)

(b)

Figure 13: Estimating the Input Cardinality for Different Values of k.

(a)

(b)

Figure 14: Estimating the Input Cardinality for Different Values of Join Selectivity.

produces less than k results, the query is *restarted* with a less restrictive condition. The final output results are then sorted to produce the top k results. A probabilistic optimization of top-k queries is introduced in [11] to estimate the optimal value of the cutoff parameter that minimizes the total cost including the risk of restarts. Optimizing top-k queries that contain only selection has been studied in [9] in the context of querying multimedia repositories. The optimization in [9] focuses on determining the best way to execute a set of filtering conditions given different costs of searching and probing the available indexes.

In contrast to previous work, we focus on optimizing ranking queries that involve joins. Moreover, our ranking evaluation encapsulates optimal rank aggregation algorithms. To the best of our knowledge, this is the first work that tries to estimate the cost of optimal rank aggregation algorithms and incorporate them in relational query optimization. We believe that the proposed optimization model for filtering operations in [9] can be used in tandem with our proposed optimization technique for selection predicates.
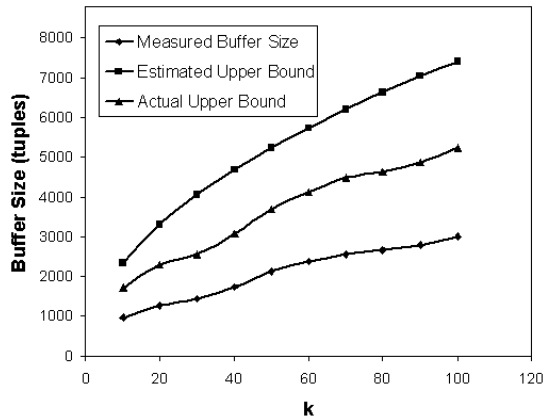
Figure 15: Estimating the Buffer Size of Rank-join.

7. CONCLUSION

We introduced a framework for integrating rank-join operators in real-world query optimizers. Our framework was based on two key steps. First, we extended the enumeration phase of the query optimizer to generate rank-aware plans. The extension was achieved by providing rank-join operators as possible join choices, and by defining ranking expressions as a new physical plan property. The new property triggered the generation of a new space of ranking plans either "naturally" by using rank-join operators or "enforced" by gluing sort operators to sort the partial results. Next, we provided a probabilistic technique to estimate the minimum required input cardinalities by rank-join operators to produce top k join results. Estimating the minimum required input cardinalities emerged from realizing the unique "early-out" property of rank-join operator. Unlike traditional join operators, rank-join operators do not need to consume all their inputs. Hence, estimating the cost of rank-join operator depends on estimating the number of tuples required from the input.

Our proposed estimation model captured this property with estimation error less than 30% of the actually measured input cardinality under some reasonable assumptions on the score distributions. We also estimated the space needed by rank-join operators with estimation error less than 40%. We conducted several experiments to evaluate the accuracy of our estimation model and the validity of our enumeration extension. The results proved the concept and showed the robustness of our estimation to several parameters such as the number of required answers and the join selectivity.

8. REFERENCES

[1] Sihem Amer-Yahia, SungRan Cho, and Divesh Srivastava. Tree pattern relaxation. In *EDBT*, 2002.
[2] J.C. Borda. M.émoire sur les élections au scrutin. *Histoire de l'Académie Royale des Sciences*, 1781.
[3] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *TODS*, 27(2), 2002.
[4] Nicolas Bruno, Luis Gravano, and Amelie Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, 2002.
[5] Michael J. Carey and Donald Kossmann. Reducing the braking distance of an SQL query engine. In *VLDB*.
[6] Michael J. Carey and Donald Kossmann. On saying "Enough already!" in SQL. In *SIGMOD*, 1997.

[7] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*, 2002.
[8] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R. Smith. The onion technique: indexing for linear optimization queries. In *SIGMOD*, pages 391–402, 2000.
[9] Surajit Chaudhuri, Luis Gravano, and Amelie Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE Transactions on Knowledge and Data Engineering, to appear.*
[10] M.-J. Condorcet. *Éssai sur l'application de l'analyse à la probabilité des décisions rendues à la puralité des voix*, 1785.
[11] Donko Donjerkovic and Raghu Ramakrishnan. Probabilistic optimization of top N queries. In *VLDB*, 1999.
[12] Cynthia Dwork, S. Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the web. In *World Wide Web*, 2001.
[13] Ronald Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences (JCSS)*, 58(1), Feb 1999.
[14] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS, Santa Barbara, California*, May 2001.
[15] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.
[16] Goetz Graefe and David J. DeWitt. The exodus optimizer generator. In *SIGMOD*, 1987.
[17] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, 2000.
[18] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, 2001.
[19] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, june 1999.
[20] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, 1(1), Jan. 1993.
[21] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, 2003.
[22] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD*, 2001.
[23] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Joining ranked inputs in practice. In *VLDB*, 2002.
[24] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, 2003.
[25] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *SIGMOD*, 1988.
[26] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, 2001.
[27] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, 1999.
[28] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path election in a relational database management system. In *SIGMOD*, 1979.
[29] Panayiotis Tsaparas, Themistoklis Palpanas, Yannis Kotidis, Nick Koudas, and Divesh Srivastava. Ranked join indices. In *ICDE*, 2003.
[30] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1), 1993.