Adaptive Rank-aware Query Optimization in Relational Databases

IHAB F. ILYAS
University of Waterloo
WALID G. AREF and AHMED K. ELMAGARMID and HICHAM G. ELMONGUI and RAHUL SHAH and JEFFREY SCOTT VITTER
Purdue University

Rank-aware query processing has emerged as a key requirement in modern applications. In these applications, efficient and adaptive evaluation of top-k queries is an integral part of the application semantics. In this paper, 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 physical 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 is the first effort in estimating the needed input size for optimal rank aggregation algorithms. Costing ranking plans is key to the full integration of rank-join operators in real-world query processing engines.

Since optimal execution strategies picked by static query optimizers lose their optimality due to estimation errors and unexpected changes in the computing environment, we introduce several adaptive execution strategies for top-k queries that respond to these unexpected changes and costing errors. Our reactive reoptimization techniques change the execution plan at run-time to significantly enhance the performance of running queries. Since top-k query plans are usually pipelined and maintain a complex ranking state, altering the execution strategy of a running ranking query is an important and challenging task.

We conduct an extensive experimental study to evaluate the performance of the proposed framework. The experimental results are two-fold: (1) we show the effectiveness of our cost-based approach of integrating ranking plans in dynamic programming cost-based optimizers; and (2) we show a significant speedup (up to 300%) when using our adaptive execution of ranking plans over the state-of-the-art mid-query reoptimization strategies.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems

General Terms: Advanced Query Processing, Ranking

Additional Key Words and Phrases: top-k, adaptive processing, rank-aware optimization

Authors Address: University of Waterloo, 200 University Ave. West, Waterloo, Ontario, Canada N2L 3G1; and Purdue University, 250 N. University Street, West Lafayette, Indiana 47907, USA.

1. INTRODUCTION

Emerging applications that depend on ranking queries warrant efficient support of ranking in database management systems. Supporting ranking gives database systems the ability to efficiently answer Information Retrieval (IR) queries. For many years, combining the advantages of databases and information retrieval systems has been the goal of many researchers. While database systems provide efficient handling of data with solid integrity and consistency guarantees, 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 query. 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 that combines the orders of each input, according to some provided function.

Several application examples exist in which rank-aware query processing is essential; in the context of the Web, the main applications include building meta-search engines, combining scoring functions and selecting documents based on multiple criteria [Dwork et al. 2001]. Efficient rank aggregation is the key to a useful search engine. In the context of multimedia and digital libraries, an important type of query is similarity matching. Users often specify multiple features to evaluate the similarity between the query media and the stored media. Each feature may produce a different order of the media objects similar to the given query, hence the need to combine these rankings, usually, through joining and aggregating the individual scores to produce a global ranking. Similar applications exist in the context of information retrieval, sensor networks and data mining.

1.1 Query Model

Most of the aforementioned applications have queries that involve *joining* multiple inputs, where users are usually interested in the top-k join results based on some score function. 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.

Hence, we focus on join as an expensive and essential operation in most top-k queries. However, several efficient algorithms have been proposed to answer top-k queries on a single table (top-k selection) [Bruno et al. 2002; Chang and Hwang 2002]. Most of these algorithms are directly applicable as a rank-aware scan operator. In this paper, we focus on rank-aware join operators and on addressing their impact on query processing and optimization.

DEFINITION 1. *Consider a set of relations R_1 to R_m. Each tuple in R_i is associated with some score that gives it a rank within R_i. The* top-k join query *joins R_1 to R_m and produces the results ranked on a total score. The total score is computed according to some function, say F, that combines individual scores. Note that the score attached with each relation can be the value of one attribute or a value computed using a predicate on a subset of its attributes.*

A possible SQL-like notation for expressing a top-k join query is as follows:

```
SELECT *
FROM R_1, R_2, ..., R_m
WHERE join_condition(R_1, R_2, ..., R_m)
ORDER BY F(R_1.score, R_2.score, ..., R_m.score)
LIMIT k;
```

where `LIMIT` limits the number of results reported to the user, `join_condition` is a general join condition that associates objects from different relations, and $R_i.score$ is the score of a tuple in Relation R_i. Note that *score* can be a single attribute or a *function* on a set of attributes in R_i. For simplicity of the presentation, we assume that *score* is a single attribute since we concentrate on the processing and optimization of the rank-aware join operators.

1.2 Motivating Examples and Challenges

Below are two real-life examples in which efficient evaluation of ranking queries is essential. Example 1 illustrates a key-equality top-k join scenario, while Example 2 gives a general top-k join scenario. We also discuss the effect of changes and fluctuations in the computing environment on the performance of query evaluation.

EXAMPLE 1. *Consider a video database that contains several extracted features, e.g., color histograms, texture, and edge orientation. These features are extracted for each video frame and are stored in separate tables. Each feature is indexed by a high-dimensional index for a faster query response. Suppose that a user is interested in the k frames most similar to a given image based on both color and texture. We refer to this type of queries as* multi-feature *or* multi-criteria *ranking queries.*

To answer single-feature queries (i.e., similarity based on texture or color individually), a database system supporting approximate matching ranks tuples based on how well they match the query. The challenge in multi-feature queries is to obtain a *global* ranking of frames based on the color and texture similarities to the query image. In the case of multi-criteria ranking, it is not clear how the database system combines the individual rankings of the multiple criteria.

EXAMPLE 2. *A family is interested in buying a house with a school nearby, with the objective of minimizing the total cost. Consider a simple cost function that sums the price of the house and* 5*-year school tuition. Searching the two web databases, HOUSES and SCHOOLS, the family issues the following query:*

```
SELECT *
FROM HOUSES H, SCHOOLS S
WHERE Distance (H.location,S.location) < d
ORDER BY H.price + 5 * S.tuition
LIMIT 10;
```

Distance is a user-defined function that evaluates how near a school is to a house. The family is interested only in the top 10 results instead of all possible join results. The *order by* clause provides the ranking function of the results. Note that each of

the two external web sources, $HOUSES$ and $SCHOOLS$, may be able to provide a list of records sorted on the price and tuition, respectively.

In current database systems, the queries in the previous examples are evaluated as follows: First, the input tables are joined according to the join condition. Then, for each join result, the global score is computed according to the given function. Finally, the results are sorted on the computed combined score to produce the top-k results and the rest of the results are dropped. Two major expensive operations are involved: joining the individual inputs and sorting the join results.

A key observation in these examples is that even if the database system can efficiently rank the inputs, the query engine cannot use these rankings to obtain a global rank on the final result. Instead, a sort operation after the join is the only way to get the final ranked query answers. We call this traditional approach *Materialize-then-Sort*, where a blocking sorting operator on top of the join is unavoidable. If the inputs are large, the cost of this execution strategy can be prohibitive.

Rank-aware query processing [Ilyas et al. 2002; 2003; Chang and Hwang 2002; Li et al. 2005] has emerged as an efficient approach to answer top-k queries. In rank-aware query processing, ranking is treated as a first-class functionality and is fully supported in the query engine. In [Ilyas et al. 2002; 2003], we introduced a set of new logical and physical *rank-join* operators to augment current relational query processors with ranking capabilities. Other rank-aware operators have also been introduced in [Chang and Hwang 2002; Bruno et al. 2002; Natsev et al. 2001].

With these operators, query engines can generate new rank-aware query execution plans that avoid the naïve *materialize-then-sort* approach. Instead, joins progressively generate ranked results, eliminating the blocking sort operator on top of the plan. Moreover, the query optimizer will have the opportunity to optimize a ranking query by integrating the new operator in ordinary query execution plans. Figure 1 gives alternative execution plans to rank-join three ranked inputs.

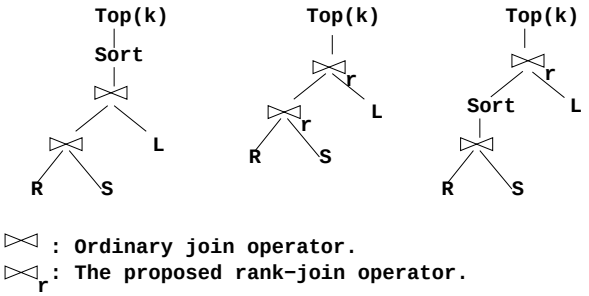
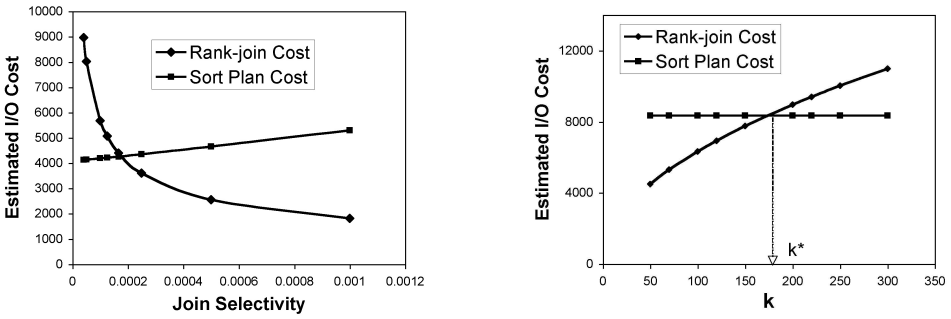


Fig. 1. Alternative execution plans to rank-join three ranked inputs.

However, the absence of a cost model of these novel operators hinders their integration in a real-world query engine. 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



(a) (b)

Fig. 2. Estimated I/O cost for two ranking plans.

access paths and the memory size), a traditional join-then-sort plan may be a better way to produce the ranked results.

Figure 2 gives the estimated I/O cost of two plans: a *sort plan* and a *rank-join plan*. Figure 2 (a) compares the two plans for various join selectivities, while Figure 2 (b) compares the two plans for various k (the required number of results). 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. Figure 2 (a) shows that for low values of the join selectivity, the traditional sort-plan is cheaper than the rank-join plan. On the other hand, Figure 2 (b) shows the superiority of the rank-join plan for low values of k. Figure 2 highlights the need to optimize top-k queries by integrating rank-join operators in cost-based query optimization.

However, even with a way to cost ranking operators and generate rank-aware query plans, Example 2 introduces another important challenge: Since the ranking involves external sources, query execution plans may stall due to the blocking calls of the *GetNext* interface when these data sources experience disconnection or unexpected long delays. Moreover, the optimality of the execution plan is compromised when certain execution characteristics change permanently or become inconsistent with the estimated values. For example, this may happen when a source becomes slower for the rest of the execution period, or the monitored selectivity shows large estimation error. Hence, the overall execution plan becomes sub-optimal and may result in a significant performance degradation. The only effective solution in these scenarios is to *alter* the execution plan to a more efficient execution strategy. We summarize the two main challenges addressed in this paper as follows:

—*Costing and Optimizing Rank-aware Plans*: We need to develop cost-based enumeration and pruning techniques to integrate ranking operators in real-world query engines. As we show, 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.

—*Coping with the Fluctuations and changes in the Computing Environment*: As more applications migrate to hand-held devices, sensor networks and the Internet,

many of the assumptions we make about the availability of data are no longer valid due to unexpected delays and frequent disconnections. Ranking queries are dominant in these less-stable environments. Providing an adaptive processing mechanism for ranking queries has a direct effect on the adaptability and performance of many modern applications.

To the best of our knowledge, there have been no proposals to handle adaptive execution of ranking queries. However, in recent years, several techniques have been proposed for adaptive query processing of traditional Boolean queries. Refer to Section 2 for further details on these proposals. As we demonstrate in Section 5, applying these techniques in rank-aware query processing is hindered by the pipelined and incremental nature of rank-aware query processing. We introduce novel mid-query reoptimization techniques that handle the specific nature of ranking operators and reuse most of the state information of rank-join plans.

1.3 Contributions

We summarize our proposed solution for the aforementioned challenges in the following contributions:

—*Ranking as a physical property:* 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.

—*Cost-based rank-aware optimization:* we tackle the challenge of pruning rank-aware execution plans based on cost. A rank-join plan stops once it produces the top-k answers. Thus, 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 to estimate the minimum input size (depth) needed. We use these estimates in pruning the generated plans.

—*Adaptive optimization of ranking plans:* we present a novel adaptive execution framework for ranking queries. We outline general adaptive processing algorithms based on the types of change in the optimality conditions of the current executing plan (e.g., cost parameters and fluctuations in the computing environment). We introduce a novel algorithm to alter the current pipelined ranking plan in run-time and to resume with the new optimal (or better) execution strategy. The plan alteration mechanism employs an aggressive reuse of the old ranking state from the current plan in building the state of the new plan.

—*Experimental study:* we conduct an extensive experimental study to evaluate all the aspects of the proposed techniques based on an implementation in PostgreSQL. The experiments show orders of magnitude performance gain over current top-k evaluation approaches in database systems and significant improvement over recent rank-join algorithms proposed in the literature. The experiments also show the accuracy and effectiveness of the proposed cost model. In evaluating the adaptive execution framework, the experiments show significant performance gain (more than 300% speedup and 86% more results in the case of source disconnection) by changing sub-optimal execution strategies in run-time.

The experiments also show significant superiority over current reoptimization techniques of pipelined query plans based on re-executing the whole query.

In an earlier work [Ilyas et al. 2004], we proposed the static rank-aware optimization techniques presented in this paper. The work in [Ilyas et al. 2004] does not address adaptive optimization of rank-aware query plans, when changes in the computing environment cause the current execution plan to become sub-optimal. In this paper, we propose several new techniques and algorithms for adaptive optimization of ranking queries, supported by extensive experimental study to evaluate our approach. We believe that the new contributions are more than 40% of this paper. We summarize the new material as follows:

—In the introduction, we provide motivating examples and new research challenges to highlight the need for adaptive rank-aware optimization (Section 1.2).
—We give the necessary background on adaptive query processing in Section 2.4.
—We enhance the cost analysis in Section 4 by discussing the effects of assumptions violation, and by showing how to change the model to relax some of these assumptions.
—We introduce a new section, Section 5, which covers our proposed adaptive optimization techniques for ranking query plans, by changing the execution strategy at run-time.
—In the experiments, we introduce a *new* section on the experimental evaluation of our proposed adaptive optimization techniques (Section 6.3).

1.4 Outline

The remainder of the paper is structured as follows. In Section 2, we give an overview of rank-aggregation and rank-aware query operators; we also give background information on traditional and adaptive query processing. 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. Section 5 describes algorithms to adaptively execute pipelined ranking queries, and describes how to reuse current state information in building new execution strategies. Section 5 also studies the applicability of eddies and operator scheduling in the context of ranking queries. We experimentally evaluate the cost estimation model and the adaptive execution techniques in Section 6. Section 7 concludes by a summary and final remarks.

2. BACKGROUND AND RELATED WORK

In this section, we highlight the research efforts to introduce efficient algorithms that integrate (aggregate) the ranking of multiple ranked inputs. Also, we present the necessary background for the new proposed rank-join algorithms.

2.1 Rank Aggregation

Rank aggregation is an efficient way to produce a global rank from multiple input rankings, and can be achieved through various techniques. 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 [Fagin 1999; Fagin et al. 2001; Nepal and Ramakrishna 1999; Güntzer et al. 2000; 2001; Natsev et al. 2001; Bruno et al. 2002; Ilyas et al. 2003; Hristidis et al. 2003; Chang and Hwang 2002] for more details on the various proposed algorithms.

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. In [Ilyas et al. 2002; 2003], we 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, produce the join results ranked according to a provided scoring function.

Ranking join operators: (1) achieve code reusability by pushing ranking from the application to the core database system; (2) exploit orderings on the inputs to produce ordered join results unlike current Boolean join operators; and (3) allow for global optimization of queries that involve other non-ranking operations such as ordinary join and selection. On the same time, rank-join operators require the following: (1) ranked inputs (or at least one of them) and each input tuple has an associated score, (2) a *GetNext* interface on the input to retrieve the next tuple in descending order of scores, and (3) 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 has been 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 [Ilyas et al. 2003].

For clarity of the presentation, we give a brief overview of one possible rank-join implementation, *hash rank-join* (HRJN): 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 [Hong and Stonebraker 1993; Wilschut and Apers 1993] or the hash ripple join algorithm [Haas and Hellerstein 1999]. 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).

In [Li et al. 2005], the authors show how to integrate rank-aware operators (including rank-join) in query engines through an extended rank-aware relational algebra. Our proposed cost-model and adaptive optimization techniques in this paper complement the approach introduced in [Li et al. 2005].

2.3 Cost-Based Query Optimization

The optimizer transforms a parsed input query into an efficient query execution plan. The execution plan is then passed to the run-time engine for evaluation. To find the best execution plan for a given query, the optimizer examines a large space of possible execution plans and compares these plans according to their "estimated" execution cost.

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. Dynamic programming (*DP*) was first used for join enumeration in System R [Selinger et al. 1979]. To avoid generating redundant plans, *DP* maintains a memory-resident structure (referred to as *MEMO*, following the terminology used in [Graefe and McKenna 1993]) 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 corresponding 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* [Selinger et al. 1979] 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

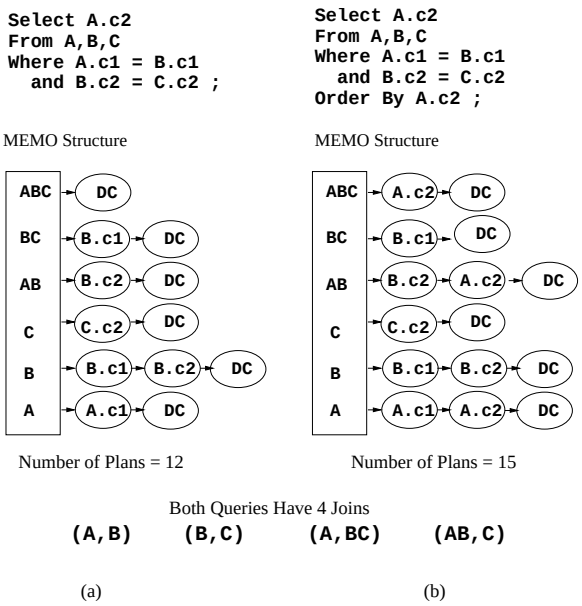


Fig. 3. Number of joins vs. number of plans.

join between the two tables cheaper than P_2 since P_1 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 3(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 3(b), by adding an *orderby* clause increases the number of interesting order properties that need to be kept in all MEMO entries containing A. The idea of interesting orders was later generalized to multiple physical properties in [Graefe and DeWitt 1987; Lohman 1988] and is used extensively in modern optimizers.

Optimization of Top-k Queries Another approach to evaluate top-k queries is the filter/restart approach [Carey and Kossmann 1997; 1998; Donjerkovic and Ramakrishnan 1999; Bruno et al. 2002]. Ranking is mapped to a filter condition with a cutoff parameter. If the filtering 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 [Donjerkovic and Ramakrishnan 1999] to estimate the optimal value of the cutoff parameter that minimizes the total cost including the risk of restarts.

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.

2.4 Adaptive Query Processing and Reusing Query Results

In recent years, several techniques have been proposed for adaptive query processing. One approach is to collect statistics about query sub-expressions during execution and to use the accurate statistics to generate better plans in the future [Bruno and Chaudhuri 2002; Stillger et al. 2001]. The mid-query reoptimization technique [Kabra and DeWitt 1998] exploits blocking operations as *materialization points* to reoptimize parts of a running query, by rewriting the query using the materialized sub-queries. Scheduling execution of query operators [Amsaleg et al. 1996] activates different parts of the query plan to adapt to high latency incurred by remote data sources. The eddies architecture and its variants [Avnur and Hellerstein 2000; Raman et al. 2003; Deshpande and Hellerstein 2004] continually optimize a running query by routing individual tuples to the different query processing operators, eliminating the traditional query plan altogether. Adaptive query operators such as ripple joins [Haas and Hellerstein 1999] are proposed to allow for changing the order of inputs within an operator. The robust progressive optimization technique (POP) [Markl et al. 2004] allows for reusing intermediate results in reoptimizing the running query by invoking the query optimizer with the availability of the materialized intermediate results.

A variant of the CHECK operator in the progressive optimization framework [Markl et al. 2004] allows for the reoptimization of *pipelined* query plans. In this variant, a *compensation* mechanism is used to avoid duplicates by storing the record identifiers (*rids*) of the already reported results, however, the whole query is re-executed and old computations cannot be reused. Recently, on-the-fly reoptimization of continuous queries in the context of data streams has been proposed in [Zhu et al. 2004]. The work in [Zhu et al. 2004] focuses on reusing state information of windowed join operations to transform one continuous query plan to another semantically equivalent one. Although our proposed techniques in Section 5 share the same objective of the state migration techniques in [Zhu et al. 2004], we focus on the particular nature of the state information in ranking query plans and incorporating the proposed techniques in real-world relational database engines. Moreover, we study the applicability of other adaptive processing techniques such as eddies and query scrambling in the context of ranking queries.

Recent work on reusing query results in the context of top-k processing [Chakrabarti et al. 2004] focuses on refined ranking queries. The work in [Chakrabarti et al. 2004] is similar to our proposal and to most of mid-query reoptimization techniques in the general principle of reusing old accessed objects to minimize the I/O cost of new queries. However, our proposed approach is significantly different as: (1) we focus on reusing the state of the rank-join operators (including hash tables and ranking queues) for the same ranking criteria but for different join orders, in response to unexpected delays and source disconnections; and (2) our technique to reuse the ranking state is completely different from reusing the cached index nodes in [Chakrabarti et al. 2004] and includes tree comparison,

promotion and demotion operations as we explain in Section 5. We view the techniques introduced in [Chakrabarti et al. 2004] as orthogonal to our proposal and can be used in tandem with the approach introduced in this paper.

3. INTEGRATING RANK-JOIN IN QUERY OPTIMIZERS

In this section we describe how to extend the traditional query optimization–one that uses dynamic programming a la [Selinger et al. 1979]–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 [Selinger et al. 1979]. 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 specify the ranking function in the *orderby* clause, in order to describe a top-k query using current `SQL` constructs.

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

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 I. Interesting order expressions in Query **Q2**.

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 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. This is because a sort on the expression (0.3*A.c1+0.3*B.c1+0.3*C.c1) cannot significantly benefit from ordering the input on A.c1 or C.c1 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 2. *An Interesting Order Expression is one that orders the intermediate results on an* expression *of database columns and 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 I. 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

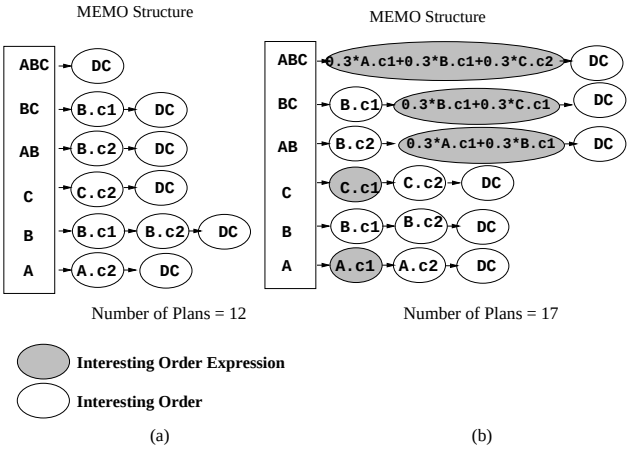


Fig. 4. Enumerating rank-aware query plans.

operators as join *choices*.

For a query with n input relations, T_1 to T_n, we assume there exists a ranking function $F(s_1, s_2, \ldots, s_n)$, where s_i is a 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 of the following conditions 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 $F(F_1(F_L), F_2(F_R), F_3(S_O))$, where F_1, F_2 and F_3 are three scoring functions, S_L are the score expressions on the relations in 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, e.g., 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 I 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 4 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 4 (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 4 (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`.

3.3 Costing and Pruning Ranking 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 5 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

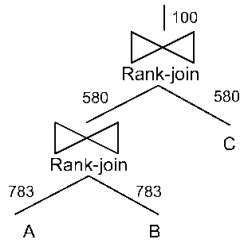


Fig. 5. Example rank-join plan.

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 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 to 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 6, 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 be compared to decide on pruning. While the current traditional cost model can give an estimated 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 5). In Section 4, we give a probabilistic model to estimate the depths (d_L and d_R in Figure 6 (b)) required by a rank-join operator to generate top k ranked results. The estimate for the depths is parametrized 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.

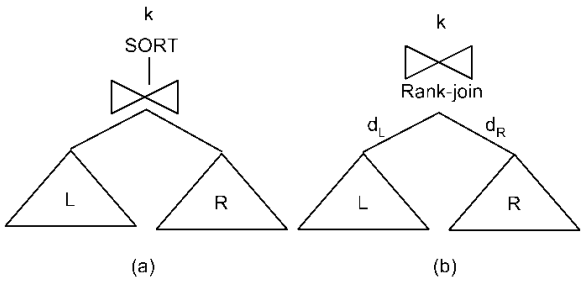


Fig. 6. Two enumerated plans.

Plan Pruning According to our enumeration mechanism, at any level, there will be only one plan similar to Plan (a) of Figure 6 (by gluing a sort on the cheapest non-ranking plan). At the same time, there may be many plans similar to Plan (b) of Figure 6 (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 the cost of a rank-join plan to the cost of a sort plan, e.g., Plans (a) and (b) in Figure 6, 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 6 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 than 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, i.e., 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 $k^* > k_{min}$: 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 6(a), while the second plan is a *rank-join plan* similar to that in Figure 6(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 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. 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. Figure 2 (b) compares the estimate of the costs of the two plans for different values 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 parametrized 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. Our objective is to get an estimate of depths d_L and d_R (see Figure 8) 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. Algorithm 1 gives Procedure *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.

4.1 Model Assumptions

We make the following assumptions on the input scores, scoring function, join selectivity and distributions. The reason for most of these assumptions are: (1) analytical convenience: the assumptions facilitate smooth symbolic computation needed to quickly estimate the input cardinalities and hence the join costs; and (2) statistics cost: relaxing these assumptions require complex statistics collection that can be prohibitively expensive or unavailable.

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 possible values of c_L and c_R, where
c_L is the depth in L and c_R is the depth in R such that,
$\exists$ 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,
$\exists$ 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 parametrized by k.

Fig. 7. Outline of the estimation technique.

Algorithm 1 Propagating the value of k.

PROPAGATE(P : *root of a subplan*, k : *number of results*)
1 Let d_L and d_R be the depths of inputs to the operator rooted at P to get k results
2 Compute d_L and d_R according to the formulas in Section 4.3
3 Call Propagate(left subplan of P, d_L)
4 Call Propagate(right subplan of P, d_R)

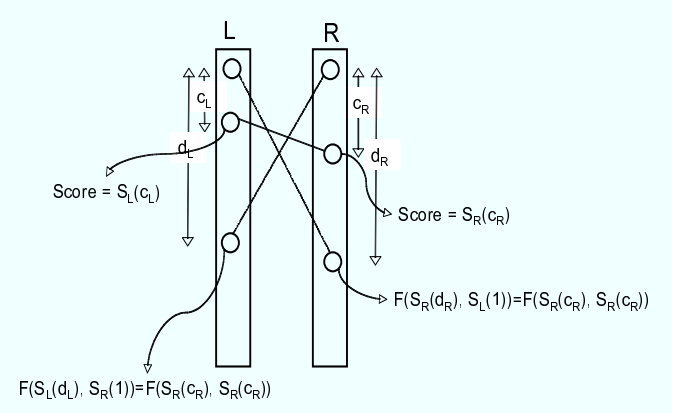


Fig. 8. Depth estimation of rank-join operators.

As in the case of traditional cost estimation, these assumptions may not hold

in practical scenarios, and hence will affect the accuracy of our estimates. We relax some of these assumptions and comment on the effect of their violation in the subsequent sections.

For two ranked relations L and R, let L_i and R_j be the i^{th} tuple and the j^{th} tuple from L and R, respectively, and let $|L|$ and $|R|$ be the total number of tuples in L and R, respectively.

—*Join Independence:* We assume that the probability for a pair of tuples being valid join result is independent of the fact that some other pair of tuples was chosen or rejected to join. Although not true in general, this assumption is always made in current optimization cost models. Also, we assume that these probabilities are independent of the tuple ranking. This can be violated when there is correlation between input rankings. We elaborate on this case in Section 4.5.

—*Join Uniformity:* We assume that a tuple from L is equally likely to join with any tuple from R. Formally, $\forall i, j\ Prob(L_i \bowtie R_j) = s$. This assumption can be violated in practice, e.g., when the rankings of the two relations are correlated. Consider the case when L and R are results from two different search engines using the same search query. In the case of positive correlation, tuples ranked high in L are more likely to join with tuples ranked high in R, violating the join uniformity assumption. Join uniformity is usually assumed in current cost-based optimizers as it simplifies the analysis significantly, besides it is usually hard to capture the general non-uniform join distribution in a cheap and scalable way. We assume join uniformity in our proposed cost model, however in Section 4.5, we show how to relax this assumption for simple rank correlation models.

—*Score Distribution:* We assume that scores of each input relations are from some standard symbolic distributions like uniform, Gaussian or Poisson and that the parameters are known. We use the parameters for deriving the resultant score distribution and also to estimate the rank of a tuple with score x and the score of the ith ranked tuple. We use uniform distribution for demonstration purposes. When distributions are not standard, we may approximate them by histograms. As long as we can manipulate the histograms for resulting distributions with relatively low cost compared to the join operation, histograms can serve as a useful estimation tool for the optimizer. We assume uniformity when the score distribution is not available.

—*Monotonic Score Aggregation Function:* The proposed rank-join algorithms assume monotone scoring function for effective pruning.

Under join uniformity and independence assumptions, we can view a join as $|L| \times |R|$ independent Bernoulli trials. In each trial the probability of success (a tuple from L joins a tuple from R) is s, where s in this case is the "join selectivity". In the following sections, we rely on these assumptions to derive estimate of depths d_L and d_R in L and R, respectively to produce the top k join results.

When these assumptions are violated, we may tend to underestimate or overestimate d_L and d_R. As in the case of traditional query processing, statistics on score distributions and rank correlation significantly enhance the accuracy of the estimate and can be easily incorporated in our model as we discuss in the following sections.

4.2 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.

THEOREM 1. *Under join uniformity and independence assumptions, the expected number of join combinations that need to be evaluated to produce k valid join results is k/s. Hence, c_L, c_R should be chosen such that their product $c_L c_R$ is $\geq k/s$ to produce k valid join results.*

PROOF. Under the assumptions in Section 4.1, each join combination is a Bernoulli trial with a probability of success s. Let X_k be a random variable that represents the number of join combinations we need to check to produce k valid join results. The expected value $E(X_k)$ is the expected number of Bernoulli trials to get k successes $= \frac{k}{s}$. In joining c_L tuples from L and c_R tuples from R, the number of trials (possible join combinations) is $c_L c_R$. We can allow any combination c_L, c_R that satisfies $c_L c_R \geq k/s$ as the estimates to produce any k valid results. Note that X_k follows a negative binomial distribution whose mean is k/s and deviation is $\sqrt{k(1-s)}/s$. This distribution is tight about its mean as k increases. Chernoff bounds analysis indicates that if we take $(k + 3.1\sqrt{k})/s$ trials we can be 99% sure there are at least k outcomes. □

Note that if the join uniformity assumption is violated, the constraint in Theorem 1 will not be accurate. We show the effect of relaxing the join uniformity assumption on Theorem 1 in Section 4.5.

4.3 Estimating Top-k Depths

In the second step in the outline given in Figure 7, we aim to obtain 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 the top k join results. The estimation procedure is inspired by the rank-join algorithm.

Assuming a monotone score aggregation function F, let $S_L(i)$ and $S_R(i)$ be the scores of the tuples at depth i in L and R, respectively.

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 $F(S_L(d_L), S_R(1)) \leq F(S_L(c_L), S_R(c_R))$ and $F(S_L(1), S_R(d_R)) \leq F(S_L(c_L), S_R(c_R))$, then the top k join results can be obtained by joining $L(d_L)$ and $R(d_R)$.*

PROOF. Refer to Figure 8 for illustration. Since there are k join tuples between $L(c_L)$ and $R(c_R)$, the final score of each of the join results is $\geq F(S_L(c_L), S_R(c_R))$. Consequently, the scores of all of the top k join results are $\geq F(S_L(c_L), S_R(c_R))$. Assume that one of the top-k join results, say J, joins a tuple t at depth d in L with some tuple in R such that $S_L(d) < S_L(d_L)$ (i.e., $d > d_L$). The highest possible score of J is $F(S_L(d), S_R(1))$. Since F is monotone, the highest possible score of J is $< F(S_L(d_L), S_R(1))$. Since we chose d_L such that $F(S_L(d_L), S_R(1)) \leq F(S_L(c_L), S_R(c_R))$, hence there are at least k join results with scores > the highest possible score of J. 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. □

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 $F(S_L(c_L), S_R(c_R))$ is maximized subject to the constraint in Theorem 1 ($sc_Lc_R \geq k$). 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.

Obtaining closed-form formulas for the estimated value of c_L, c_R, d_L, and d_R depends on the score distribution and the aggregation function. In Section 4.4, we give an example estimation by specifying these parameters.

4.4 Example Estimation of d_L and d_R

In this section, we study an example implementation of the estimation framework described in Figure 7. To have a closed-form formula for the minimum d_L and d_R, we assume that: (1) the rank aggregation function is the sum of scores; and (2) 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 $S_L(c_L)$ is $S_L(1)-xc_L$ and the expected value of $S_R(c_R)$ is $S_R(1) - yc_R$. To maximize $F(S_L(c_L), S_R(c_R))$, we maximize $(S_L(1) + S_R(1)) - (xc_L+yc_R)$, i.e., we minimize xc_L+yc_R, subject to $sc_Lc_R \geq k$. The minimization is achieved by setting $c_L = \sqrt{(yk)/(xs)}$ and $c_R = \sqrt{(xk)/(ys)}$. Applying Theorem 2, $S_R(1) + S_L(d_L) = S_R(c_R) + S_L(c_L)$ and $S_L(1) + S_R(d_R) = S_R(c_R) + S_L(c_L)$. In this case, $d_L = c_L + (y/x)c_R$ and $d_R = c_R + (x/y)c_L$. In a simplistic case, both the relations come from the same uniform distribution, i.e., $x = y$, then $c_L = c_R = \sqrt{k/s}$ and $d_L = d_R = 2\sqrt{k/s}$.

4.5 The Effect of the Join Uniformity Assumption with Rank Correlation

In the previous sections, we assume the independence of the join from the input rankings, and we assume the uniformity of the join (cf. Section 4.1). In practical scenarios the rank scores between two joining relations can be correlated. For example, the highly ranked tuples from one relation are more likely to join with highly ranked tuples in the other relation (positive correlation). Another example is when there is a trade-off between the two ranking criteria, e.g., ranking houses on price and location (negative correlation).

In general, it may be hard to capture the nature of this correlation between the ranks of two lists. The correlation may exist between highly ranked tuples only and does not exist in the rest of the two relations. Input cardinality estimates can be significantly different when such correlations are strong. In the context of Theorem 1, we can no longer assume that $L(c_L) \bowtie L(c_R)$ can produce k output tuples by just ensuring that $c_Lc_R \geq k/s$. Hence, we need to change the constraint for maximization. That is, we maximize $F(S_L(c_L), S_R(c_R))$ subject to the constraint that $L(c_L) \bowtie L(c_R)$ produces k join tuples.

We calculate the error in estimating d_L and d_R assuming join uniformity in the two worst case scenarios: Let $s = 1/n$, then in a perfect positive correlation

scenario, an object with rank i in L will have a rank i in R as well. In this case, it is not hard to see that the rank-join algorithm retrieves only k objects from L and R to produce the top-k objects. In this case, we overestimate d_L and d_R with a factor of $\frac{2\sqrt{kn}}{k} = 2\sqrt{n/k}$. In the other worst case scenario, only the last k objects from L and R join to get the top-k objects. In this case, the rank-join algorithm may need to retrieve up to n objects from both relations. In this case, we underestimate d_L and d_R by a factor of $\frac{n}{2\sqrt{kn}} = (\sqrt{n/k})/2$. The error factors increase (slowly) as n increases, and they can cause the optimizer to choose a suboptimal execution plan.

To illustrate the effect of relaxing the join uniformity and rank-independence, we assume the following simple rank correlation model: Consider two relations L and R with n tuples each and with scores uniformly distributed $[0, n]$. Assume that the join probability between the i^{th} ranked tuple in L and the j^{th} ranked tuple in R is $prob(R_i \bowtie S_j) = 1/\theta$ if $|i - j| \leq \theta/2$ and 0 otherwise. Therefore, each ranked tuple has a "window" of similarly ranked tuples in the other relation with which it can join. Note that the expected value of the join selectivity is still approximately $1/n$, however the join is not uniformly distributed. We can show that c_L and c_R can be estimated by $k + \theta/4$ and hence $d_L = d_R = 2k + \theta/2$ (see Appendix A.1). Note that with this simple model of correlation our estimate of d_L and d_R has decreased from $2\sqrt{k/s}$ to $2k + \theta$ to reflect the effect of rank correlation. Similar analysis can be obtained for other simple forms of rank correlation.

4.6 Tighter Bounds in a Join Hierarchy

Algorithm 1 gives an outline of how to propagate the estimate of d_L and d_R in a hierarchy of joins. For simplicity of the presentation, we assume a sum scoring function and uniformly distributed scores for base relations.

In a join hierarchy, the score distributions of the second level joins are no longer uniform. The output scores of a rank-join with two uniformly distributed inputs follow a triangular distribution. As we go higher up in the join hierarchy, the distribution tends to be normal (a bell-shaped curve) by the Central Limit Theorem.

Let u_p be the sum of p distributions, each is a uniform distribution on $[0, n]$. Let L be the output of rank-joining l ranked relations and let R be the output of rank-joining r ranked relations. For simplicity, assume that each of L and R has n tuples and follows u_l and u_r, respectively. Let k be the number of output ranked results required from the subplan, and s be the join selectivity. We can show that maximizing $S_L(c_L) + S_R(c_R)$ (according to Theorem 2) amounts to minimizing $(l!c_L n^{l-1})^{1/l} + (r!c_R n^{r-1})^{1/r}$ (see Appendix A.2). The minimization yields:

$$c_L^{r+l} = \frac{(r!)^l k^l n^{r-l} l^{rl}}{s^l (l!)^r r^{rl}} \tag{1}$$

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

$$d_L = c_L[1 + r/l]^l \tag{3}$$

$$d_R = c_R[1 + l/r]^r \tag{4}$$

Note that d_L and d_R are upper-bounds assuming expected values of $c_L c_R$. Although the estimates might be accurate for a small number of joins, the error

propagation becomes more significant as we proceed through the hierarchy. We introduce an alternative framework for tighter bounds in estimating d_L and d_R based on estimating the score distribution of the join results.

Let T be the output join relation $T = L \bowtie R$, $S_{rel}(i)$ be the score of the i^{th} tuple in Relation rel, and $rank_{rel}(x)$ be the rank of the first tuple (in decreasing order of scores) with score $\leq x$ in Relation rel. i.e., $rank_{rel}(x) = min_i|S_{rel}(i) \leq x$. According to the rank-join algorithm, we can estimate:

$$d_L = rank_L(S_T(k) - S_R(1)) \tag{5}$$

$$d_R = rank_R(S_T(k) - S_L(1)) \tag{6}$$

In general, the ability to manipulate the score distributions to produce $S_{rel}(i)$ and $rank_{rel}(x)$ allows us to estimate d_L and d_R in a join hierarchy. We give an example estimation of these quantities in the case of uniform score distributions for the base relations.

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 the relation T with a u_{l+r} distribution and sn^2 tuples. Using Equation 7 from Appendix A.2, by setting $j = l + r$ and $m = sn^2$, the score of the top kth tuple in T is $S_T(k) = (l + r)n - ((l + r)!kn^{l+r-2}/s)^{1/(l+r)}$. Hence, we need to check in L up to a tuple that joins with R to produce $S_T(k)$. Using Equations 5 and 6, we can show that on average, d_L and d_R can be computed as follows:

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

Similarly, bounds for other distributions like Gaussian and Poisson can be symbolically computed under weighted sum scoring function. We can apply the formulas recursively in a rank-join plan, as shown in the Algorithm 1, 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. ADAPTIVE EXECUTION OF RANKING QUERIES

Since adaptive execution of ranking plans depends heavily on the state maintained by these plans, we briefly summarize the state information of a ranking plan. We concentrate on the hash rank-join implementation.

$HRJN$ maintains an internal state that consists of the following components:

—*Input Hashes*: For each input, we maintain a hash table that stores objects seen so far from that input. We can view this state as "half" the state of a symmetric hash-join operator.

—*Threshold*: The rank-join threshold is an upper-bound ranking score of all uncomputed (partial) join results.

—*Operator Rank-queue*: Each rank-join operator buffers intermediate join results that have not qualified yet as top-k results. A join result qualifies as an output if its score (partial score in the case of an intermediate result) is greater than the threshold maintained by the operator.

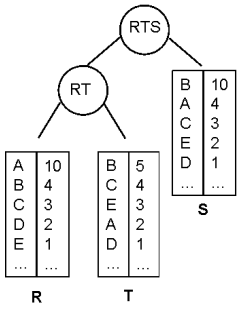


Fig. 9. An initial plan for Example 3.

We use the following example to further describe these components of the ranking state information.

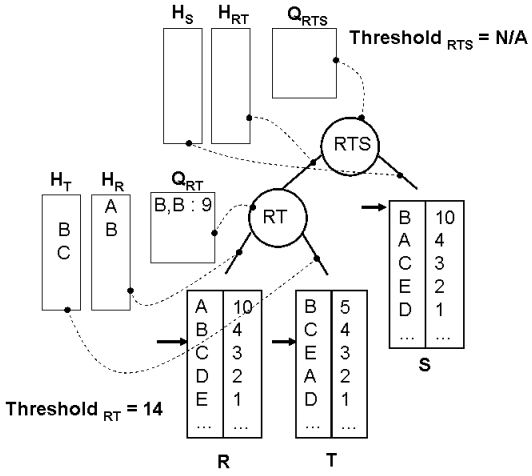
EXAMPLE 3. *Assume that we are interested in the top-k results of joining three inputs R, S and T. Each input provides objects along with their scores ordered in descending order of their scores. For simplicity, let the join condition be on the object id, and the ranking function be the sum of the individual scores. Assume that based on statistics, the plan in Figure 9 is chosen to be the best execution plan.*

Figure 10(a) gives the state information of the execution plan in Figure 9, at some point during the execution. The marker arrows on the inputs show the objects retrieved from each input. At this point in the execution, the input hash tables H_R and H_T have the objects A, B and B, C, respectively. There is a match (a valid join result) (B, B) with a combined score of 9. Since the threshold at this point of time is computed as 14, this valid join result is not guaranteed to be in the top-k answers. Hence (B, B) is kept in the operator rank-queue, Q_{RT}. The rank-queue of Operator RTS and its input hashes are empty at this point of time. Now assume that changes in the computing environment result in Plan P_{old} being sub-optimal. After updating the statistics and system parameters, the optimizer chooses a new plan, say P_{new}. Figure 10 (b) gives the new plan, P_{new} along with its new state information. In this paper, we show how to transform the state between these two plans at run-time without re-executing the query.

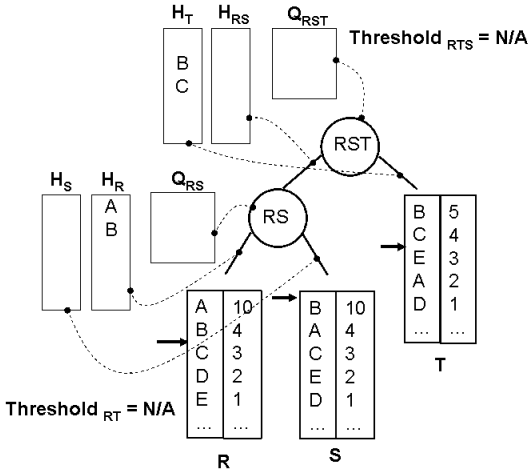
Applying current adaptive query processing techniques (discussed in Section 2.4) in rank-aware query processing is hindered by the following problems:

—*Non-standard execution models*: Techniques that require drastic changes in the database engine such as query scrambling [Amsaleg et al. 1996] or eddies [Avnur and Hellerstein 2000] (eliminating the query plan altogether) may experience high run-time overhead and integrating them in practical query engines is still an open question. However, both operator scheduling and eddies architectures can be used in adaptive rank-aware query processing. We briefly show how to modify these techniques in our context.

—*The pipelined and incremental nature of ranking queries*: Techniques that allow for mid-query reoptimization, such as the pioneering work in [Kabra and DeWitt 1998], depend on the existence of *materialization points*. The main goal in rank-



(a) The initial plan P_{old}



(b) The new plan P_{new}

Fig. 10. Ranking state information of two execution plans for Example 3.

aware query processing is to avoid blocking sort operators and to use incremental and pipelined join operators such as ripple-join. Hence, techniques that depend on materialization points cannot be applied.

—*Ranking state information*: In contrast to traditional pipelined join operators, ranking operators maintain a computation state. Changing the execution plan at run-time has to deal with this state and then reuse it in computing the state of the new plan.

5.1 Altering and Reusing Ranking Plans

In this section, we introduce an adaptive query processing technique for ranking queries, which allows for altering the query execution plan in run-time to cope with

unexpected changes and fluctuations in the underlying computing environment.

Handling the state information is the main challenge in run-time reoptimization of query execution plans. The core idea of the proposed technique is to transform the ranking state information of the current plan, say P_{old}, to another state that can fit in the new execution plan, say P_{new}. The objective of the transformation is to produce a new ranking state that reflects the scenario as if P_{new} were chosen from the beginning of the execution. The proposed technique assumes the ability of accessing the state information of the inputs and the operators using simple primitives. For example, we assume that we can store objects in the input hash tables, probe the input hash tables for matches, and retrieve objects from the operators' rank-queue. In this section, we limit our focus to plans with rank-join operators only. The proposed state transformation algorithm in this section will be used in the reoptimization frameworks in Section 5.2.

For clarity, we show the state transformation mechanism by a simple application on Example 3. In Example 3, let the initial execution plan, say P_{old}, be the plan depicted in Figure 9, and let Figure 10(a) reflect the state information after executing P_{old} for some time. Now assume that changes in the computing environment result in Plan P_{old} being sub-optimal. After feeding the optimizer with the updated statistics and system parameters, the optimizer chooses a new plan, say P_{new}. For simplicity, let P_{new} be the same rank-join plan as P_{old} except for switching the two inputs T and S. The state transformation should result in a new state that realizes the scenario as if P_{new} were the initial plan. Figure 10 (b) gives the new plan, P_{new}, along with its new state information. Before showing how to transform the state between these two plans, we make the following observations:

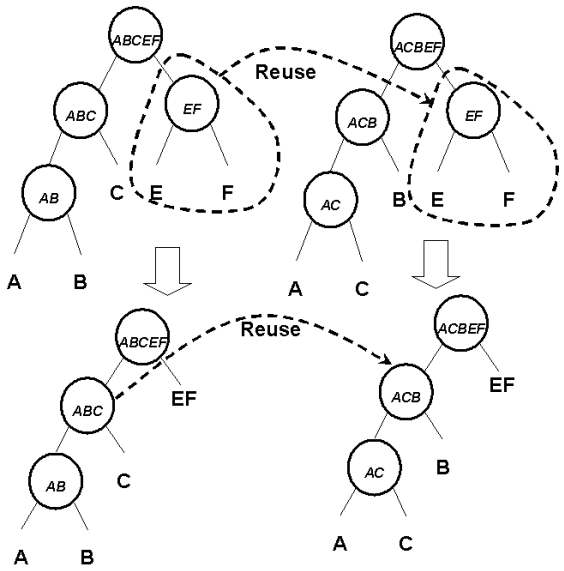


Fig. 11. Collapsing common sub-plans.

—By switching the two inputs, T and S, the operator RT has disappeared and is replaced by the operator RS. Hence, the rank-queue Q_{RT} should no longer exist in the new plan.

—Since the new operator RS is created in the new plan, the state information associated with that operator needs to be created, i.e., a new rank-queue, Q_{RS}, and a new threshold.

—The threshold values can always be computed for the newly created operators since they depend only on the joined relations.

—The main objective of the transformation is to continue the execution of P_{new} without having to redo most of the work of P_{old}. This objective, however, does not guarantee that repeating some work already done by P_{old} will be eliminated.

We identify two major steps in our proposed state transformation mechanism: (1) *Merging the old state* by "removing" the effect of the old input, and deleting the state information that has no equivalent in the new plan; and (2) *Creating a new state* that is unique to the new plan. Merging the old state has the effect of losing some of the work done in the old plan. New state creation may involve accessing the old state to fill the new state with all information that can be "reused" from the old plan. In Algorithm 2, we give the generalized algorithm for reusing the state information in the current sub-optimal plan, P_{old}, to build the state of an arbitrary new ranking execution plan, P_{new}.

Procedure *RegainState* in Algorithm 2 takes as input two rank-join plan structures, P_{new} and P_{old}. The algorithm first reduces the two input plans by "collapsing" all common sub-plans (sub-trees). A common sub-plan is the one that appears in both P_{new} and P_{old}, possibly in different locations. Figure 11 gives an example of this step by collapsing the common sub-plan that joins inputs E and F. The state information of common sub-plans are copied as is into the new generated plan. The algorithm proceeds by initializing all ranking-queues and hash tables of the non-leaf nodes in P_{new}. Building the new state of P_{new} is carried out by calling the Procedure *JoinAndPromote*. The reuse of old information in Procedure *RegainState* is limited to identifying common sub-plans in P_{new} and P_{old}. A more aggressive state reuse is carried out in Procedure *JoinAndPromote* by exploiting the commutativity and symmetry of rank-join operators.

Procedure *JoinAndPromote* in Algorithm 2 builds the new state information by reusing useful old state information and by performing all the remaining joins. Duplicate results are prevented by the check in Step 14, making use of the invariant that no rank-join result has been reported with a score less than the global threshold (maintained by the root of the rank-join tree) [1].

The novelty of the algorithm is in reusing the state of "non-leaf" nodes by identifying those non-leaf nodes with the same set of leaf nodes (inputs) even if they are joined in a different join order. This approach is more elaborate than mid-query reoptimization techniques that identify only materialization points. For example, consider the plans in Figure 11. Since the nodes ABC and ACB have the same set

[1]For ties in global score, the algorithm as described may miss results with the last reported score. One solution is to buffer all the reported results in a root $InputHash$ and compare the reported results against the buffered reported results.

Algorithm 2 Building the new state information of the newly constructed plan.

REGAIN STATE(P_{new} : *The New Plan*)

```
1  Identify all common sub-plans in both plans.
2  Collapse common sub-plans in both plans.
3  Copy state information of all common sub-plans (including all leaf nodes).
4  Reset InputHash and RankQueue of all non-leaf nodes in P_new.
5  CALL JoinAndPromote(P_new.Root)
```

JOINANDPROMOTE(X : *Plan Node*)

```
 1  if X is a leaf
 2       return
 3  CALL JoinAndPromote(LeftChild(X))
 4  CALL JoinAndPromote(RightChild(X))
 5  if ∃ a node in P_old with same leaves of X
 6       Let the node in P_old be X_old
 7       Copy X_old state to X.InputHash
 8       return
 9  JOIN LeftChild(X).InputHash and RightChild(X).InputHash
10  for each join result j
11    do
12       if X is root
13            if j.score < X.Threshold
14                 Put j in X.RankQueue
15            else
16                 Put j in X.InputHash
```

of leaf nodes, $\{A, B, C\}$, the state information stored in node ABC in P_{old} can be reused in node ACB in P_{new}.

The state migration in Algorithm 2 can be modified to handle the state of other rank-join implementations (e.g., nested-loops rank-join). We omit the details of the modifications for space limitation.

5.2 Adaptive Execution Frameworks

As mentioned in Section 1.2, inter-operator adaptability (altering execution plan in run-time) depends on the optimizer being able to produce a new plan in response to the detected changes. The success of the proposed technique relies on our ability to "monitor" the query execution and the changes in critical parameters such as source delays, source availability, and selectivity. In query monitoring, we build upon the work introduced in [Kabra and DeWitt 1998] and in the POP framework [Markl et al. 2004] (progressive optimization in query processing) to monitor and to check the optimality of the currently executing plan. The CHECK operator detects unexpected execution behavior such as significant errors in selectivity estimation. It can also be extended to detect large source delays.

Upon detecting changes in *optimizer-sensitive parameters* (e.g., selectivity estimates), we introduce a mid-query reoptimization framework that uses the optimizer

during execution to generate a new optimal execution strategy. We handle the special characteristics of ranking query plans to reuse old execution state to build the state of the new plan. Section 5.2.1 presents our solution for this case.

Upon encountering changes in *optimizer-insensitive parameters* (e.g., unexpected delays), the optimizer cannot reflect these changes in a new query execution plan. In Section 5.2.2, we introduce a heuristic algorithm to enhance the performance of the currently executing query or resolve blocking situations, where otherwise, no progress can be achieved using the current plan.

Note that adaptive processing techniques can suffer from *thrashing* if we spend most of the time adapting the plan to changes in the environment. Minimizing thrashing can be achieved through several techniques, e.g., by limiting the number of times reoptimization can be invoked [Markl et al. 2004], or by setting a minimum number of tuples to be processed (or a time interval) between two consecutive reoptimizations [Deshpande and Hellerstein 2004].

5.2.1 *Optimizer-based Plan Altering.* We describe a framework for progressive optimization of pipelined ranking queries that adapts to changes in optimizer-sensitive parameters. We summarize the general technique as follows:

(1) Continually check the running plan P_{old} for unexpected changes or fluctuations in cost parameters.
(2) Stop plan execution.
(3) For all blocking subtrees, materialize the results and make them available for reoptimization.
(4) Re-invoke the optimizer to reoptimize the query taking into account the adjusted statistics, to produce a new execution plan P_{new}.
(5) Produce the rank-join plans from both plans (plans with rank-join operators only), RP_{old} and RP_{new}, from P_{old} and P_{new}, respectively.
(6) Build the state information of RP_{new} by reusing old state information in RP_{old} according to the algorithms in Section 5.1.
(7) Resume execution using P_{new} from where P_{old} stopped.

In Step 3, we use the same framework in POP [Markl et al. 2004] or in mid-query reoptimization [Kabra and DeWitt 1998] taking advantage of the available materialization points in the query plan, should one exist. We note that most ranking plans are pipelined plans with ranking state information, as discussed in Section 5.1. In Step 5, we limit our attention to the rank-join operators, where the ranking state information is maintained. All rank-join operators use the adaptive implementation of Section 2. Step 6 invokes the state transformation algorithms in Section 5.1 to build the state information of the new query plan. A sketch of the mid-query reoptimization framework is given in Figure 12.

5.2.2 *Heuristic Plan Altering Strategy for Unexpected Delays.* The current execution plan may become sub-optimal because of changes in the environment conditions that do not affect the optimizer's objective function. In this case, reoptimizing the query according to the algorithm in Section 5.2.1 will not produce a new, more efficient execution strategy. Rather, heuristic techniques are usually applied to generate a *more efficient* execution strategy. One example of a heuristic to deal

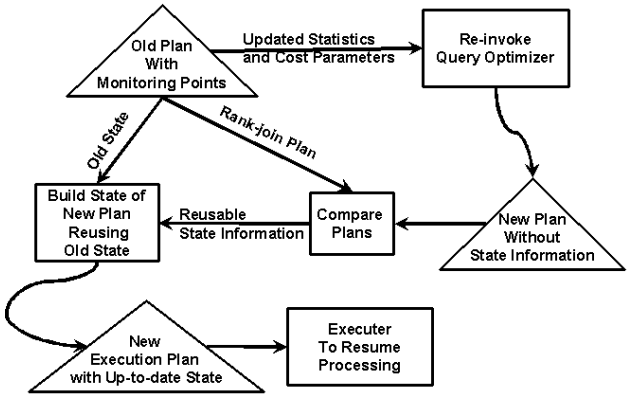


Fig. 12. Mid-query reoptimization framework for ranking queries.

with unexpected delays and source disconnections is the *Pair* strategy in query scrambling [Amsaleg et al. 1996]. The heuristic is designed to delay the execution of stalled sources and generate useful sub-plans to mask these delays. Another heuristic for optimizing rank-join operations is to take score correlations into account [Ilyas et al. 2003]. The idea is to greedily cluster similar score distributions (based on the foot-rule similarity metric) together in the execution plan.

For example, consider the three-way rank-join plan depicted in Figure 13. The RJ_i nodes are pipelined rank-join operators, and A, B, and C are three ranked inputs from external sources. For simplicity, we place monitoring points [Kabra and DeWitt 1998; Markl et al. 2004] (e.g., CHECK operators) on every edge in the query plan. Now assume that Source B experiences long delays. The rank-join operator that joins A and B (RJ_1) tries to deal with the unexpected delays by pulling more records from A than B. If the delays persist and more progress can be achieved by rank-joining B with C first, the reoptimization logic will change the execution strategy through a number of steps: (1) stop the execution of the current plan; (2) swap the two inputs B and C; and (3) compute the computation state of the new plan using the old state information. Note that after the shuffling of B and C, the state information of RJ_1 becomes invalid and needs to be rebuilt. The goal of reusing the old state information is to simulate the scenario as if the new plan were the original execution strategy.

The heuristic used in the previous example pushes the delayed data sources up in the tree as close as possible to the root join operator. The problem in the proposed heuristic occurs when dealing with source availability. Assume that Source B becomes unavailable for a long period of time. Without assuming any non-standard operator scheduling mechanism (e.g., as the one proposed in [Amsaleg et al. 1996]), the plan will still block even after pushing B up in the plan tree. Fortunately, the intra-operator adaptability of the rank-join operators will deal with this problem, as described in Section 1.2. Effectively, the rank-join operator with a delayed or non-responding input will cut this input from the currently executing plan. Whenever the input can be accessed, it joins the current execution plan.

The following steps give the general framework to alter a running execution plan

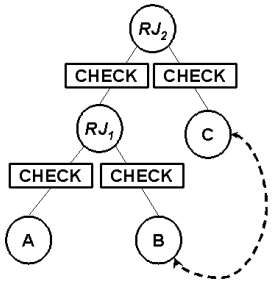


Fig. 13. A simple reoptimization example.

that stalls or experiences large unexpected delays. For simplicity, we assume that we monitor the execution of the plan by fixing monitoring points (e.g., CHECK operators) on top of every data source and rank-join operator.

(1) Identify the sub-tree with the maximum experienced delay greater than a threshold T_{delay}. The threshold is set to allow the rank-join operators to adapt to short, non-persistent delays experienced by remote sources and is an application-dependent parameter.
(2) Stop the current execution and perform *one swapping step* to swap the sub-tree with a child of the top rank-join operator.
(3) Transform the computation state of the current query evaluation plan to reflect the new generated plan according to the algorithms in Section 5.1.
(4) Resume execution using the new generated execution plan.

The rationale behind the heuristic is to maximize the amount of work carried out assuming the unavailability of the problematic part of the plan. In Section 6, we experimentally evaluate the effect of this heuristic on performance.

5.3 Non-traditional Adaptive Techniques

In the literature, there have been several proposals for adaptive query processing that do not follow the traditional optimizer design. We identify two of these techniques, namely, query scrambling and eddies. In query scrambling [Amsaleg et al. 1996; Urhan et al. 1998], scheduling the execution of query operators activates different parts of the query plan to adapt to the high latency incurred by remote data sources in a wide area network. The eddies architecture and its variants [Avnur and Hellerstein 2000; Raman et al. 2003; Deshpande and Hellerstein 2004] continually optimize a running query by routing tuples to various query processing operators, eliminating the traditional query plan altogether. In this section, we show that how to extend these techniques to adaptively process ranking and top-k queries.

Scrambling of Ranking Queries In general, the query scrambling framework consists of two phases: a scheduling phase and an operator synthesis phase. The scheduling phase does not change the query plan structure; rather, it allows for different operators to be executed independently in their own execution threads. Communication among operators is through queues that hold intermediate results.

If an operator cannot proceed or experiences long delays, other operators in the plan are scheduled to execute. The scheduling policy can be set to maximize the amount of work that can be carried out before the plan completely stalls. In the second phase, new operators are formed when the original plan structure cannot produce any useful work. The authors in [Urhan et al. 1998] introduced several techniques to alter the stalled execution plan depending on the type of the underlying optimizer.

The first phase of query scrambling is directly applicable to rank-aware query processing and can be combined with our intra-operator adaptability. Our proposed plan alteration technique can be considered as one approach to implement the second phase in query scrambling.

Ranking Eddies Adaptive rank-aware query processing can be achieved by implementing rank-join as an eddy. We show how to implement rank-join using eddies state modules (STEMS) [Raman et al. 2003]. In contrast to rank query processing in traditional database engines, all the ranking information is stored in the eddy; no local rankings are kept among inputs. Effectively, an eddy can be viewed as a multi-way rank-join operator with adaptive capabilities. All recently proposed enhancements [Raman et al. 2003; Deshpande and Hellerstein 2004] can be applied to minimize the run-time overhead of tuple routing.

For presentation clarity, we give an implementation of the rank-join algorithm [Ilyas et al. 2003] to join ranked inputs with arbitrary join conditions. The rank-join algorithm joins these inputs to produce the top-k join results. The total score is computed according to some monotone functions that aggregate the object scores from the input lists. We list the modifications to the eddy internal state to support ranking the results according to a monotone function F as follows:

—*A global ranking threshold* T, which is an upper-bound of the final ranking score of all unseen join results. The threshold can be computed according to the rank-aggregation algorithms in [Fagin et al. 2001; Ilyas et al. 2003].

—*A ranking queue* Q, which is a priority queue of all seen join results with scores less than the threshold, and hence cannot be reported as top-k answers.

Algorithm 3 gives a *GetNext* implementation of a rank-aware eddy. An example execution is depicted in Figure 14, where we rank-join the three ranked inputs of Example 3. Instead of forming a rank-join query plan, we construct an eddy to retrieve the ranked inputs, route the inputs and intermediate results among the STEMS and produce the output in the order of the ranking function (in this example, the sum of individual scores). The STEMs are just hash tables on the join attribute with *insert* and *probe* interfaces. At this point in the execution, the eddy has retrieved the first two tuples from each input and the threshold is computed to be 14[2]. There is only one completed join result so far (B, B, B), with a total score of $14 \geq T$. Hence, the result can be reported as a top-k result.

6. PERFORMANCE EVALUATION

In this section, we conduct two sets of experiments. In the first set, we experimentally verify the accuracy of our model for estimating the depths (input size) of

[2]We apply the algorithm in [Ilyas et al. 2003] for threshold computations.

Algorithm 3 Rank-join implementation in an eddy.

EddyRankJoin.GetNext()

```
1  let I be a set of ranked inputs
2  Initialize the ranking queue Q to be empty
3  Initialize ranking threshold T to a large number
4  if Q is not empty
5       let top be the top of Q
6       if Score(top) > T
7            Remove from Q and Return top
8  while true
9    do Retrieve next ranked tuples from inputs and insert them in corresponding STEMs
10      Adjust T with the input scores
11      Route input tuples and intermediate results according to the routing strategy
12      let j be a valid join result
13      Compute the score of j according to F
14      Insert j in Q
15      if Q is not empty
16           let top be the top of Q
17           if Score(top) > T
18                break
19 Remove top from Q
20 return top
```

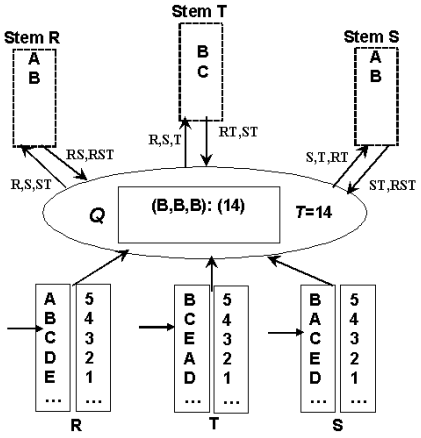


Fig. 14. Rank-join in eddies.

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 makes it easy to estimate the total cost of a rank-join plan according to any practical cost model. In the second set of experiments, we evaluate the effect

of the proposed adaptive techniques on top-k query performance.

6.1 Implementation Issues and Setup

We implemented the rank join operators in [Ilyas et al. 2003] into PostgreSQL 7.4.3 running Linux with 4 XEON $2GHz$ processors and $4GB$ of memory.

For the first set of experiments, Figure 15 gives a ranking plan (termed Plan P) that joins 4 sorted inputs. d_i labels on the edges represent the number of ranked tuples flowing from the children of the rank-join operator. We compare the actual (monitored) values of d_i to the estimated values according to our cost model.

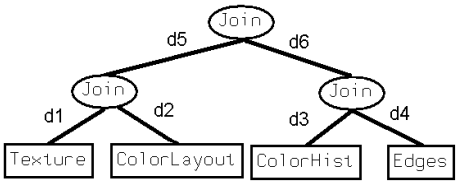


Fig. 15. Example rank-join plan.

For the second set of experiments, we implemented the following simple monitors:

—*Delay Monitors*: Delay monitors are placed on top of all input relations. The monitors report the time to retrieve one input record. If the monitored delay exceeds a specified threshold value, we raise a *need-to-reoptimize* flag.

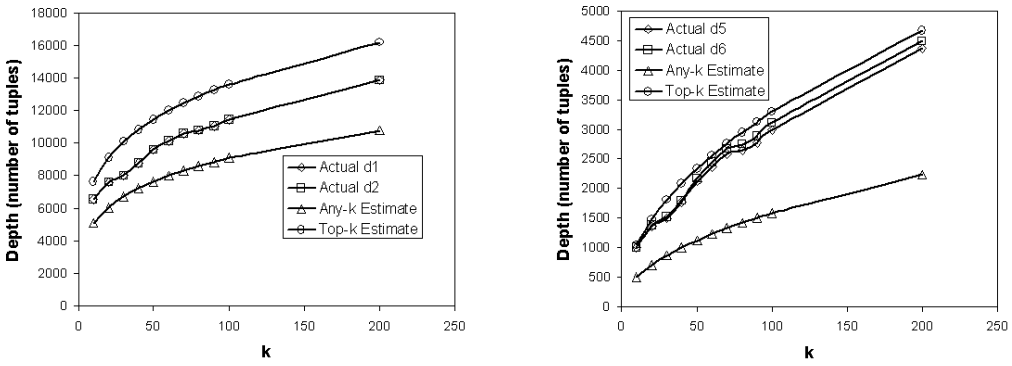
—*Score Correlation Monitors*: A correlation monitor is placed in each rank-join operator in the query plan. The monitor computes the number of tuples retrieved from each input to report one top-k result. If the number of tuples retrieved from the operator inputs exceeds the expected input cardinality (computed according to the cost estimation model), we raise a *need-to-reoptimize* flag.

We avoid thrashing by limiting the number of reoptimizations during query execution. In this set of experiments, the user query joins 3 to 10 ranked inputs and asks for the top-k join results with respect to a monotone aggregating function. We use the *sum* of the tuples' scores as the aggregate function. The data is generated synthetically to simulate web site popularity search. In this real scenario, a highly ranked object in one input (relation), is likely to be highly ranked in other inputs as well. We simulate this scenario by picking the difference between the rank of an object in one relation and the rank of the same object in another relation from a Zipfian distribution ($\alpha = 0.9$). Each input provides a ranked list of objects with several attributes, e.g., (id, $JoinAttribute$, $score$). The difference between the positions of an object in two lists is randomly drawn from a Zipfian distribution . Each input can be accessed only through a sorted access path (a $GetNext()$ interface), i.e., no random access or indexes are allowed on the inputs. Hence, our inputs act as external ranking sources. In the following sections, we elaborate on a representative sample of the conducted experiments.

6.2 Experiments Set 1: Verifying Input Cardinality Estimation

In this experiment, we evaluate the accuracy of the depth estimates of rank-join operators. The results shown here represent the estimates for Plan P in Figure 15. 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 16 (a) compares the actual values of d_1 and d_2 (refer to Figure 15) 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 as in Section 4. The actual values of d_1 and d_2 are obtained by running the query and counting the number of retrieved input tuples by each operator. Figure 16 (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. 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 a *lower-bound* on the depths required by a rank-join operator.

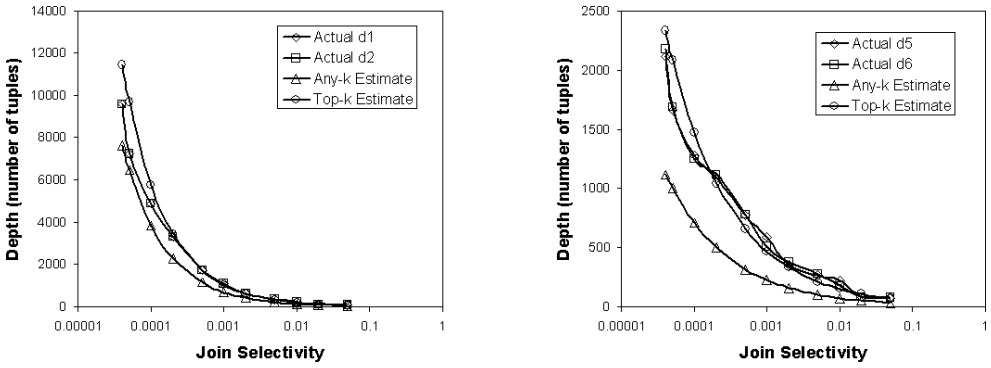


(a) (b)

Fig. 16. Estimating the input cardinality for different values of k.

Varying the Join Selectivity Figure 17 compares the actual and estimated values for the depths of Plan P in Figure 15 for various join selectivities. For low selectivity values, the required depths increase as the rank aggregation algorithm needs to retrieve more tuples from each input to produce the top ranked join results. The maximum estimation error is less than 30% of the actual depth values.

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 important in estimating the total cost of



(a) (b)

Fig. 17. Estimating the input cardinality for different values of join selectivity.

a rank-join operator. In this experiment, we use Plan P in Figure 15. 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 18 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 18 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.

The Effect of Rank Correlation We conduct a simple experiment on a real data set to show the effect of rank correlation on our estimation model. We used data generated from a multimedia database system that ranks objects based on feature similarity to a given query image. We used two correlated features (color histogram and color layout). We apply the rank-join algorithm to get the top-k objects with respect to both features (using sum of ranks as the aggregation function). Let d_1 and d_2 be the depths in the two ranked inputs, respectively. Figure 19 gives the actual monitored values f d_1 and d_2 varying the number of required results k. Note that in this simple rank-join scenario $d_1 = d_2$. Figure 19 also shows our estimated value of the depths using two estimation models:(1) assuming join uniformity; and (2) assuming a simple rank correlation following the model described in Section 4.5, with $\theta = 50$. Because of the strong positive correlation, we tend to overestimate the

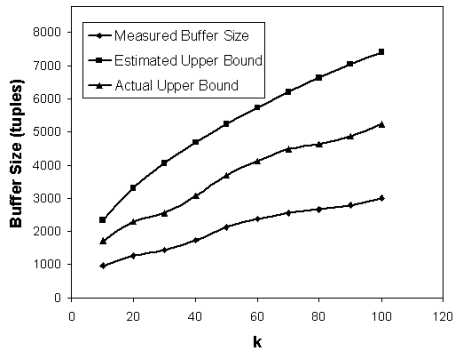


Fig. 18. Estimating the buffer size of Rank-join.

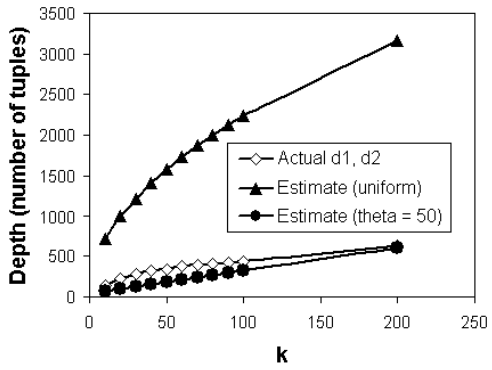


Fig. 19. The effect of rank correlation.

depths if we assumed the uniformity of the join; however, much better estimates can be obtained applying our simple rank correlation model.

6.3 Experiments Set 2: Adaptive Mid-query Reoptimization

In this section, we conduct a set of experiments to evaluate two issues: First, we examine the overhead of the proposed mid-query reoptimization algorithm. We would like to answer the question: When is it beneficial to reoptimize a running query upon discovering a change in the optimality condition? Second, we evaluate the efficiency of our mid-query plan alteration algorithms.

Overhead of Mid-query Reoptimization In this experiment, we start execution using a default sub-optimal plan (e.g., because of faulty estimates). Figure 20 compares the performance speedup achieved by reoptimizing the running ranking query during execution using: (1) our adaptive plan alteration strategy to switch to a new plan (*Plan B*) in run-time; and (2) by restarting the query using the new plan (*Plan B*). We assume that we can restart the execution of a running pipelined query although this may not always be possible (e.g., in the case of input streams). The speedup is computed as the ratio of execution time of the original plan to that of the new plan. The cost of plan alteration is included in the execution time of our adaptive strategy. Several experiments with different data sets showed the same behavior in Figure 20. Figure 20 shows that:

—Our adaptive execution strategy can achieve significant speedup over re-executing the query using the new evaluation plan. This superiority is due to the aggressive reuse of internal state information to minimize the amount of the repeated work.

—The benefit of reoptimizing the query diminishes as execution progresses due to the overhead of state migration in our adaptive strategy, or due to repeating work in the re-execution strategy. In fact, as the query gets closer to completion, re-executing the query can result in a significant performance degradation. Our aggressive reuse of old state information significantly limits the performance degradation with query progress. These results suggest adopting cost-based

mid-query reoptimization. For example, the following costs can be estimated: (1) $cost_A$, the cost to complete execution using the current plan (after updating the statistics); (2) $cost_B$, the cost to resume execution using the new generated plan; and (3) $cost_{state}$, the cost to transform the state between plans according to the algorithms in Section 5.1. $cost_{state}$ can be easily computed since accurate cardinality information is available. A simple cost-based decision is to reoptimize whenever $cost_A$ is significantly greater that $cost_B + cost_{state}$.

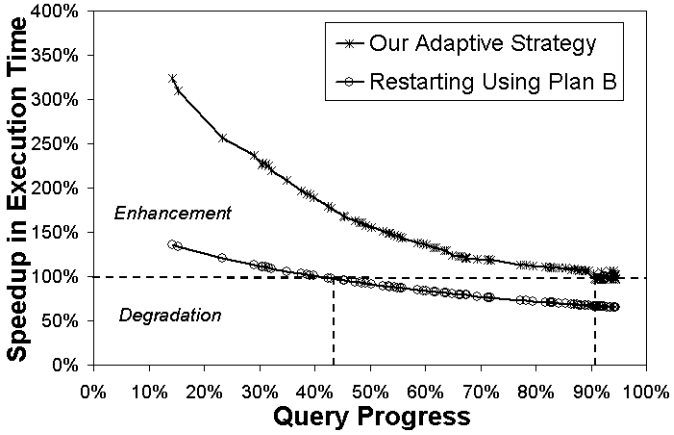


Fig. 20. Mid-query reoptimization overhead.

Mid-query Plan Alteration PostgreSQL query optimizer is not sensitive to correlations among input scores, and is not sensitive to source delays and disconnections. Since the focus of this paper is to provide a mid-query reoptimization algorithm for ranking queries, the evaluation is independent from the source of generating the new execution plan (by re-invoking the optimizer or by using an optimization heuristic). In this section, we conduct two experiments: The first experiment introduces delays and disconnections in the data sources; when the *Delay Monitors* trigger the need to reoptimize the query, we apply the heuristic in Section 5.2.2 by pushing the delayed (disconnected) source as close as possible to the root of the rank-join tree. In the second experiment, we start executing the query following a default join order (this is not uncommon in the absence of statistics information in traditional optimizers). Upon discovering a bad execution strategy using the *Score Correlation Monitors*, we use the rank-join order heuristic in [Ilyas et al. 2004] to form a new plan strategy that takes into account the statistics on score similarities that are now available on the inputs.

In both experiments, our plan transformation algorithm (described in Section 5.1) transforms the internal state of the old execution plan to a valid state of the new plan, allowing the executor to resume query evaluation with the new strategy.

Figure 21 gives an evaluation of the heuristic proposed in Section 5.2.2. The plan used in Figure 21 is a left-deep tree with 10 ranked inputs. A large delay is introduced in the farthest input from the root. The performance improvement

is evaluated for every possible swapping of the delayed input with other inputs. The figure shows that better performance can be achieved by pushing delayed or disconnected inputs as close as possible to the root in the rank-join plan.

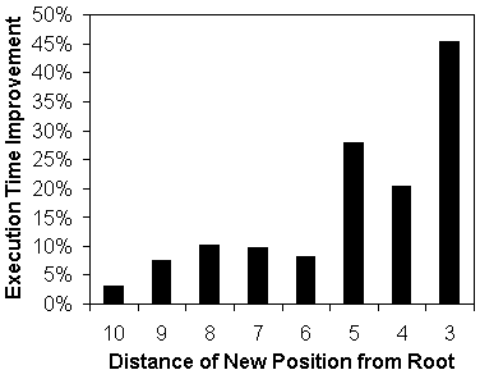


Fig. 21. Heuristic for dealing with source delays.

Figure 22 compares the performance of two ranking queries in three different scenarios: (1) Using *Plan A*, the default plan that is used when we do not have prior knowledge about source delays or availability; (2) Using *Plan B*, the plan that takes delay information into account by pushing the slowest source near the root; and (3) By *Changing Plan A to Plan B*, which is our proposed strategy to start execution using *Plan A* and to progressively switch to *Plan B* upon detecting large source delays. Figures 22(a) and (b) give the execution time of two ranking queries (normalized by dividing the execution time by the total query time) with respect to the three different scenarios. The transition was decided around the point where 5 top results have been already reported to the user. After the transition, we follow closely the performance of *Plan B*. The execution time includes the overhead of performing the state transition. Notice that in Figure 22(b) the total execution time according to our strategy is even less than starting execution using *Plan B*. Although this is not always guaranteed (see Figure 22(a)), the reason can be explained as follows. During a reuse of the state information to transform the execution strategy to *Plan B*, we may be able to report many valid top-k results depending on how many inputs we have already read from all inputs, which may exceed the number of results that can be reported if we use *Plan B* to retrieve inputs from the beginning of the execution. After the transition, the time to report each of the next top-k results using our strategy is the same as that of *Plan B*. To further illustrate this transition, Figure 23 gives the difference in the execution time per reported result between our strategy and both *Plan A* and *Plan B*. Similar results are observed in the second experiment, where we reoptimize the query upon detecting errors in deciding the rank-join order. For space limitation, we only show a comparison of the total number of retrieved tuples in Figure 22(c).

To test the effectiveness of our adaptive strategy in handling source disconnections and plan stalls, we conduct the following experiment: In a ranking query with

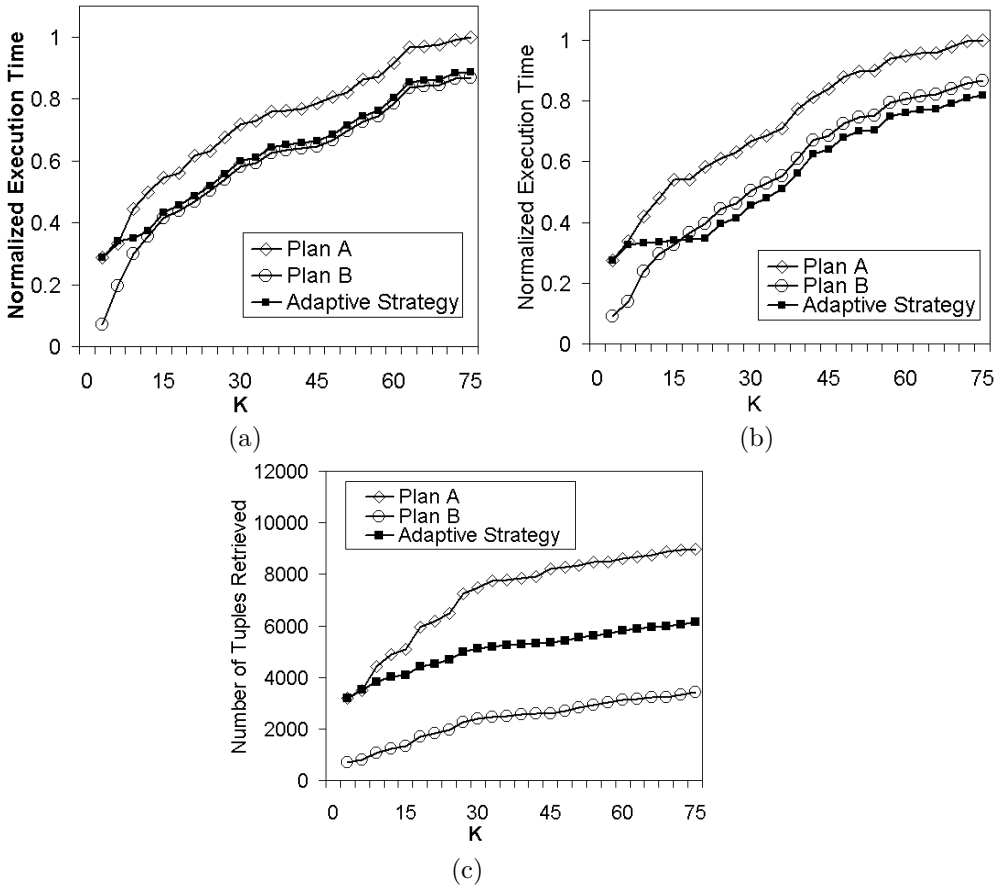


Fig. 22. Mid-query alteration of execution plan.

4 ranked inputs shown in Figure 24(a) as *Plan A*, we disconnect source S during the execution of *Plan A*, which causes the plan to stall. Upon discovering the plan stall, we alter the execution plan to *Plan B* as shown in Figure 24(a). In Figure 24(b), we show the number of reported top-k results. Plan alteration gives the opportunity for more "useful" work to be done by reusing the old state information and by retrieving tuples from other inputs. As a result, 26 more top-k results are reported, an 86% increase. The ranking plans in this experiment use the adaptive implementation of the rank-join operator described in Section 5.1.

7. CONCLUSION

We introduced a framework for integrating rank-aware operators in real-world query optimizers. Our framework was based on three key aspects: First, we extended the enumeration phase of the 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

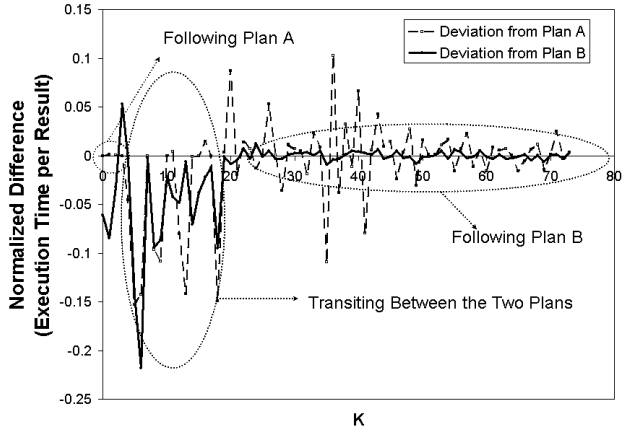


Fig. 23. Transiting between plans.

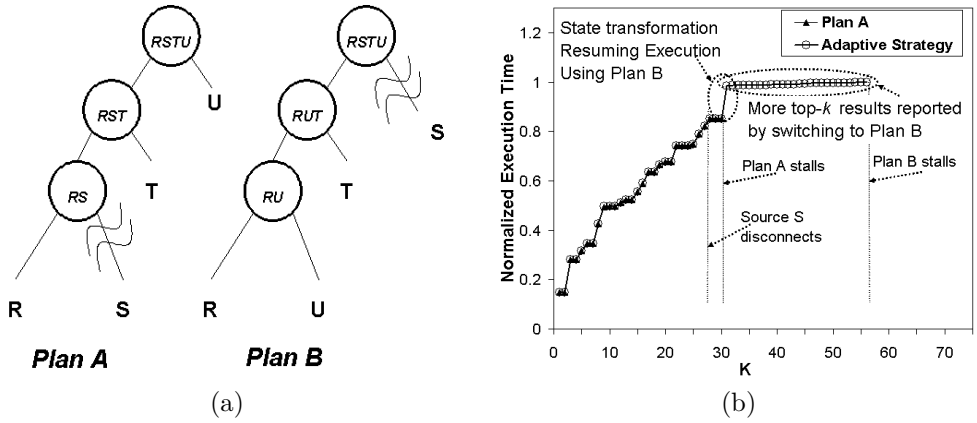


Fig. 24. Mid-query alteration for source disconnections.

the generation of a new space of ranking plans for efficient top-k processing.

Second, we provided a probabilistic technique to estimate the minimum required input cardinalities by rank-join operators to produce top k join results. Unlike traditional join operators, rank-join operators do not need to consume all their inputs. Hence, estimating the cost of rank-join operators depends on estimating the number of tuples required from the input.

Third, we addressed adaptive processing of ranking queries in modern ubiquitous computing environments. We proposed several techniques and progressive optimization algorithms for ranking query plans. We also outlined general adaptive processing frameworks based on the type of changes in the optimality conditions of the current executing plan. We distinguished between two types of changes: (1) changes and errors in the optimizer cost parameters (optimizer-sensitive changes) that require re-invoking the query optimizer during execution to produce a new optimal

ranking plan; and (2) changes and fluctuations in the computing environments that are not factored in the optimizer cost model (optimizer-insensitive changes), where heuristic techniques may be used to produce a better execution strategy. In the core of these frameworks, we introduce a novel algorithm to alter the current pipelined ranking plan in run-time and to resume with the new optimal (or better) execution strategy. The plan alteration mechanism employs an aggressive reuse of old ranking state from the current plan in building the state of the new plan. We have also studied the application of other non-traditional adaptive query processing techniques such as query scrambling and eddies in the context of ranking queries.

We conducted an extensive experimental study and we showed that our proposed estimation model captured the "early-out" 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%. The experiments also showed significant performance gain by changing sub-optimal execution strategies in run-time (more than 300% speedup and 86% more results in the case of source disconnection). The experiments demonstrated the significant superiority over current reoptimization techniques of pipelined query plans based on re-executing the whole query.

REFERENCES

Amsaleg, L., Franklin, M. J., Tomasic, A., and Urhan, T. 1996. Scrambling query plans to cope with unexpected delays. In *Distributed and Parallel Database Systems*. 208–219.

Avnur, R. and Hellerstein, J. M. 2000. Eddies: Continuously adaptive query processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 261–272.

Bruno, N. and Chaudhuri, S. 2002. Exploiting statistics on query expressions for optimization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 263–274.

Bruno, N., Chaudhuri, S., and Gravano, L. 2002. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Sys. 27,* 2, 369–380.

Bruno, N., Gravano, L., and Marian, A. 2002. Evaluating top-k queries over web-accessible databases. In *Proc. 18th Int. Conf. on Data Engineering*. 153–187.

Carey, M. J. and Kossmann, D. 1997. On saying "Enough already!" in SQL. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 219–230.

Carey, M. J. and Kossmann, D. 1998. Reducing the braking distance of an SQL query engine. In *Proc. 24th Int. Conf. on Very Large Data Bases*. 158–169.

Chakrabarti, K., Ortega-Binderberger, M., Mehrotra, S., and Porkaew, K. 2004. Evaluating refined queries in top-k retrieval systems. *IEEE Transactions on Knowledge and Data Engineering 16,* 2, 256–270.

Chang, K. C.-C. and Hwang, S.-W. 2002. Minimal probing: supporting expensive predicates for top-k queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 346–357.

Deshpande, A. and Hellerstein, J. M. 2004. Lifting the burden of history from adaptive query processing. In *Proc. 30 Int. Conf. on Very Large Data Bases*. 948–959.

Donjerkovic, D. and Ramakrishnan, R. 1999. Probabilistic optimization of top N queries. In *Proc. 25th Int. Conf. on Very Large Data Bases*.

Dwork, C., Kumar, S. R., Naor, M., and Sivakumar, D. 2001. Rank aggregation methods for the web. In *Proc. 10th Int. World Wide Web Conference*. 613–622.

Fagin, R. 1999. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences (JCSS) 58,* 1 (Feb), 216–226.

Fagin, R., Lotem, A., and Naor, M. 2001. Optimal aggregation algorithms for middleware. In *Proc. 20th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*. 102–113.

Graefe, G. and DeWitt, D. J. 1987. The EXODUS optimizer generator. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*.

Graefe, G. and McKenna, W. J. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proc. 9th Int. Conf. on Data Engineering*. 209–218.

Güntzer, U., Balke, W.-T., and Kiessling, W. 2000. Optimizing multi-feature queries for image databases. In *Proc. 26th Int. Conf. on Very Large Data Bases*. 419–428.

Güntzer, U., Balke, W.-T., and Kiessling, W. 2001. Towards efficient multi-feature queries in heterogeneous environments. In *International Symposium on Information Technology (ITCC)*. 622–628.

Haas, P. J. and Hellerstein, J. M. 1999. Ripple joins for online aggregation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 287–298.

Hong, W. and Stonebraker, M. 1993. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Database Systems 1,* 1 (Jan.), 9–32.

Hristidis, V., Gravano, L., and Papakonstantinou, Y. 2003. Efficient IR-style keyword search over relational databases. In *Proc. 29th Int. Conf. on Very Large Data Bases*.

Ilyas, I. F., Aref, W. G., and Elmagarmid, A. K. 2002. Joining ranked inputs in practice. In *Proc. 28th Int. Conf. on Very Large Data Bases*. 950–961.

Ilyas, I. F., Aref, W. G., and Elmagarmid, A. K. 2003. Supporting top-k join queries in relational databases. In *Proc. 29th Int. Conf. on Very Large Data Bases*. 754–765.

Ilyas, I. F., Aref, W. G., and Elmagarmid, A. K. 2004. Supporting top-k join queries in relational databases. *The VLDB Journal 13,* 3, 207–221.

Ilyas, I. F., Shah, R., Aref, W. G., Vitter, J. S., and Elmagarmid, A. K. 2004. Rank-aware query optimization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 203–214.

Kabra, N. and DeWitt, D. J. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 106–117.

Li, C., Chang, K. C.-C., Ilyas, I. F., and Song, S. 2005. Query algebra and optimization for relational top-k queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*.

Lohman, G. M. 1988. Grammar-like functional rules for representing query optimization alternatives. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*.

Markl, V., Raman, V., Simmen, D. E., Lohman, G. M., and Pirahesh, H. 2004. Robust query processing through progressive optimization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 659–670.

Natsev, A., Chang, Y.-C., Smith, J. R., Li, C.-S., and Vitter, J. S. 2001. Supporting incremental join queries on ranked inputs. In *Proc. 27th Int. Conf. on Very Large Data Bases*. 281–290.

Nepal, S. and Ramakrishna, M. V. 1999. Query processing issues in image (multimedia) databases. In *Proc. 15th Int. Conf. on Data Engineering*. 22–29.

Raman, V., Deshpande, A., and Hellerstein, J. M. 2003. Using state modules for adaptive query processing. In *Proc. 19th Int. Conf. on Data Engineering*. 353–387.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. 1979. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*.

Stillger, M., Lohman, G. M., Markl, V., and Kandil, M. 2001. LEO - DB2's learning optimizer. In *Proc. 27th Int. Conf. on Very Large Data Bases*. 19–28.

Urhan, T., Franklin, M. J., and Amsaleg, L. 1998. Cost-based query scrambling for initial delays. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 130–141.

Wilschut, A. N. and Apers, P. M. G. 1993. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Database Systems 1,* 1, 68–77.

Zhu, Y., Rundensteiner, E. A., and Heineman, G. T. 2004. Dynamic plan migration for continuous queries over data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*.

A. DERIVATIONS OF USED FORMULAS

A.1 Estimating Any-k Depth with Rank Correlation

Since we have not relaxed the join independence assumption, we still can view the join with window rank correlation (described in Section 4.5) as a set of independent Bernoulli trials. However, the probability of success is not s, rather it is $\frac{1}{\theta}$ within the window and 0 elsewhere.

For simplicity, assume that $c_L = c_R = c$. We can visualize the process by plotting depth in L along x axis and depth in R along y axis. Let Area A be the area that falls between the lines $y = x + \theta/2$, $y = x - \theta/2$, $x = c$, and $y = c$. The area of A represents the number of possible join combinations (trials) between $c_L = c$ and $c_R = c$.

We apply our analysis in Section 4.2 on Area A: The probability of success (a valid join) $= \frac{1}{\theta}$. Hence, $area(A)$ has to be greater that $k\theta$. For $\theta < 2c$, $area(A) = c\theta - \frac{\theta^2}{4}$, therefore, to get k valid join results, c has to be $\geq k + \theta/4$. This analysis changes when θ grows bigger than $2c$ and approaches the case of uniform join discussed in Theorem 1.

A.2 Estimating d_L and d_R in a Join Hierarchy

Let u_p be the sum of p distributions, each is a uniform distribution on $[0, n]$. Let X_i's, for $i = 1, ..., j$, be identical and independent random variables from u_1. Then, $X = \sum_1^j X_i$ is a random variable from u_j, which ranges from $[0, jn]$. Let $Y_i = n - X_i$ and $Y = \sum_1^j Y_i = jn - X$. By symmetry, the score decrement random variable Y follows the same distribution of X. If we select m items from u_j and select the ith highest-score item, then score decrement of i, $\delta(i) = jn - S(i)$ satisfies the following property: $prob(Y \leq \delta(i)) = i/m$. The left-hand term $prob(Y_1 + Y_2 + ... + Y_j) \leq \delta(i)$ is actually the ratio of volume of simplex $Y_1 + Y_2 + ... + Y_j \leq \delta(i)$, $\delta(i) \leq n$ to the volume of hypercube with length n. The volume of the simplex is $\delta(i)^j/j!$, thus, $(\delta(i)^j/j!)/n^j = i/m$. This implies

$$S(i) = jn - (j! i n^j/m)^{1/j}, \text{ when } i \leq m/j! \tag{7}$$

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 6 (b). Let L be the output of rank-joining l ranked relations and let R be the output of rank-joining r ranked relations. For simplicity, assume that each of L and R has n tuples. Let k be the number of output ranked results required from the subplan, and s be the join selectivity. Using Equation 7, we set $j = l$, $m = n$, and $i = c_L$ to get $S_L(c_L)$. Similarly, we set $j = r$, $m = n$, and $i = c_R$ to get $S_R(c_R)$. Maximizing $S_L(c_L) + S_R(c_R)$ (according to Theorem 2) amounts to minimizing $(l! c_L n^{l-1})^{1/l} + (r! c_R n^{r-1})^{1/r}$. The minimization yields:

$$c_L^{r+l} = \frac{(r!)^l k^l n^{r-l} l^{rl}}{s^l (l!)^r r^{rl}} \quad , \quad c_R^{r+l} = \frac{(l!)^r k^r n^{l-r} r^{rl}}{s^r (r!)^l l^{rl}}$$

$$d_L = c_L[1 + r/l]^l \quad , \quad d_R = c_R[1 + l/r]^r$$