Supporting Top-k *Join* Queries in Relational Databases

Ihab F. Ilyas Walid G. Aref Ahmed K. Elmagarmid
Department of Computer Sciences, Purdue University
West Lafayette IN 47907-1398
{ilyas,aref,ake}@cs.purdue.edu

Abstract

Ranking queries produce results that are ordered on some computed score. Typically, these queries involve joins, where users are usually interested only in the top-k join results. Current relational query processors do not handle ranking queries efficiently, especially when joins are involved. In this paper, we address supporting top-k join queries in relational query processors. We introduce a new rank-join algorithm that makes use of the individual orders of its inputs to produce join results ordered on a user-specified scoring function. The idea is to rank the join results progressively during the join operation. We introduce two physical query operators based on variants of ripple join that implement the rank-join algorithm. The operators are non-blocking and can be integrated into pipelined execution plans. We address several practical issues and optimization heuristics to integrate the new join operators in practical query processors. We implement the new operators inside a prototype database engine based on PREDATOR. The experimental evaluation of our approach compares recent algorithms for joining ranked inputs and shows superior performance.

1 Introduction

Rank-aware query processing has become a vital need for many applications. In the context of the Web, the main applications include building meta-search engines, combining ranking functions and selecting documents based on multiple criteria [6]. 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 ranking of the media objects similar to the query, hence the need to combine these rankings, usually, through joining and aggregating the individual feature rankings to produce a global ranking. Similar applications exist in the context of information retrieval and data mining.

Most of these applications have queries that involve joining multiple inputs, where users are usually interested in the top-k join results based on some score function. Since most of these applications are built on top of a commercial relational database system, our goal is to support top-k join queries in relational query processors. The answer to a *top-k join query* is an ordered set of join results according to some provided function that combines the orders on each input.

More precisely, 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, 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)
STOP AFTER k;
```

1.1 Motivation

The join operation can be viewed as the process of spanning the space of Cartesian product of the input relations to get valid join combinations. For example, in the case of a binary join operation, the Cartesian space of the input relations A and B is a two dimensional space. Each point is a tuple pair (A_i, B_j), where A_i is the i^{th} tuple from the first relation and B_j is the j^{th} tuple from the second relation. The join condition needs to be evaluated for all the points in the space. However, only part of this space needs to be computed to evaluate *top-k join queries*. This partial space evaluation is possible if we make use of the individual orderings of the input relations.

Current join operators cannot generally benefit from orderings on their inputs to produce ordered join results. For example, in sort-merge join (MGJN) only the order on the join column can be preserved. In nested-loops join (NLJN), only the orders on the outer relations are preserved through the join, while in hash join (HSJN), orders from both inputs are usually destroyed after the join, when hash tables do not fit in memory. The reason is that these join operators decouple the join from sorting the results. Consider the following example ranking query:

Q1:
```
SELECT A.1,B.2
FROM A,B,C
WHERE A.1 = B.1 and B.2 = C.2
ORDER BY (0.3*A.1+0.7*B.2)
STOP AFTER 5;
```

where `A, B` and `C` are three relations and `A.1,B.1,B.2` and `C.2` are attributes of these relations. The *Stop After* operator, introduced in [3, 4], limits the output to the first five tuples. In *Q1*, the only way to produce ranked results on the expression $0.3 * A.1 + 0.7 * B.2$ is by using a sort operator on top of the join. Figure 1 (a) gives an example query execution plan for *Q1*. Following the concept of *interesting orders* [16] introduced in system R, the optimizer may already have plans that access relations `A` and `B` ordered on `A.1` and `B.2`, respectively. Interesting orders are those that are useful for later operations (e.g., sort-merge joins), and hence, need to be preserved. Usually, interesting orders are on the join column of a future join, the grouping attributes (from the *group by* clause), and the ordering attributes (from the *order by* clause).

Despite the fact that individual orders exist on `A.1` and `B.2`, current join operators cannot make use of these individual orders in producing the join results ordered on the expression $0.3 * A.1 + 0.7 * B.2$. Hence, the optimizer ignores these orders when evaluating the *order by* clause. Therefore, a sort operator is needed on top of the join. Moreover, consider replacing $B.2$ by $B.3$ in the *order by* clause. According to current query optimizers, $B.3$ is not an *interesting order* since it does not appear (by itself) in the *order by* clause. Hence, generating a plan that produces an order on $B.3$ is not beneficial for later operations. On the other hand, $B.3$ is definitely *interesting* if we have a rank-join operator that uses the orders on $A.1$ and $B.3$ to produce join results ordered on $0.3 * A.1 + 0.7 * B.3$. Having a rank-join operator will probably force the generation of base plans for B that has an order on $B.3$.

Two major problems arise when processing the previous rank-join query using current join implementations: (1) sorting is an expensive operation that produces a total order on all the join results while the user is only interested in the first few tuples. (2) Sorting is a blocking operator and if the inputs are from external sources, the whole process may stall if one of the sources is blocked.

1.2 Our Contribution

The two aforementioned problems result from decoupling the sorting (ranking) from the join operation and losing the advantage of having already ranked inputs. We need a ranking-aware join operator that behaves in a smarter way in preserving the *interesting* orders of its inputs. We need the new rank-join operator to: (1) perform the basic join operation under general join conditions. (2) conform with the current query operator interface so it can be integrated with other query operators (including ordinary joins) in query plans. (3) make use of the individual orders of its inputs to avoid the unnecessary sorting of the join results. (4) produce the first ranked join results as quickly as possible. (5) adapt to input fluctuations; a major characteristic in the applications that depend on ranking. We summarize our contribution in this paper as follows:

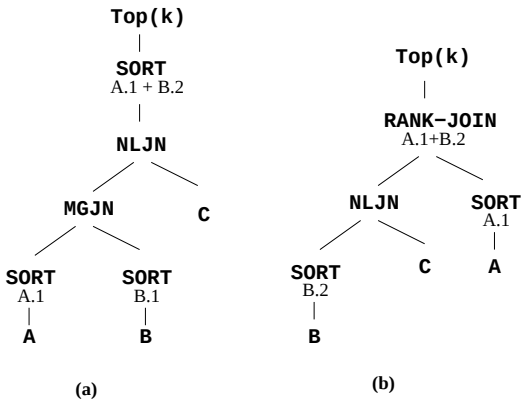


Figure 1: Alternative plans for Query **Q1**.

- We propose a new rank-join algorithm, with the above desired properties, along with its correctness proof.
- We implement the proposed algorithm in practical pipelined rank-join operators based on ripple join, with better capabilities of preserving orders of their inputs. The new operators can be integrated in query plans as ordinary join operators and hence give the optimizer the chance to produce better execution plans. Figure 1 (b) gives an example execution plan for **Q1**, using the proposed rank-join operator (RANK-JOIN). The plan avoids the unnecessary sort of the join results by utilizing the base table access plans that preserve interesting orders. Moreover, the plan produces the top-k results incrementally.
- We propose a novel score-guided join strategy that minimizes the range of the Cartesian space that needs to be evaluated to produce the top-k ranked join results. We introduce an adaptive join strategy for joining ranked inputs from external sources, an important characteristic of the applications that use ranking.
- We experimentally evaluate our proposed join operators and compare them with other approaches to join ranked inputs. The experiments validate our approach and show a superior performance of our algorithm over other approaches.

The remainder of this paper is organized as follows. Section 2 describes relevant previous attempts and their limitations. Section 3 gives some necessary background on ripple join. Section 4 describes the query model for answering top-k join queries. Also, in Section 4, we introduce the new rank-join algorithm along with its correctness proof. We present two physical rank-join operators in Section 5. In Section 6, we generalize the rank-join algorithm to exploit any available random access to the input relations. Section 7 gives the experimental evaluation of the new rank-join operator and compares it with alternative techniques. We conclude in Section 8 with a summary and final remarks.

2 Related Work

A closely related problem is supporting top-k *selection* queries. In top-k selection queries, the goal is to apply a scoring function on multiple attributes of the same relation to select tuples ranked on their combined score. The problem is tackled in different contexts. In middleware environments, Fagin [7] and Fagin et al. [8] introduce the first efficient set of algorithms to answer ranking queries. Database objects with m attributes are viewed as m separate lists, each supports sorted and, possibly, random access to object scores. The TA algorithm [8] assumes the availability of random access to object scores in any list besides the sorted access to each list. The NRA algorithm [8] assumes only sorted access is available to individual lists. Similar algorithms are introduced (e.g., see [9, 10, 15]). In [2], the authors introduce an algorithm for evaluating

top-k selection queries over web-accessible sources assuming that only random access is available for a subset of the sources. Chang and Hwang [5] address the expensive probing of some of the object scores in top-k selection queries. They assume a sorted access on one of the attributes while other scores are obtained through probing or executing some user-defined function on the remaining attributes.

A more general problem is addressed in [14]. The authors introduce the J^* algorithm to join multiple ranked inputs to produce a global rank. J^* maps the rank-join problem to a search problem in the Cartesian space of the ranked inputs. J^* uses a version of the A^* search algorithm to guide the navigation in this space to produce the ranked results. Although J^* shares the same goal of joining ranked inputs, our approach is more flexible in terms of join strategies, is more general in using the available access capabilities, and is easier to be adopted by practical query processors. In our experimental study, we compare our proposed join operators with the J^* and show significant enhancement in the overall performance. The top-k join queries are also discussed briefly in [5] as a possible extension to their algorithm to evaluate top-k selection queries.

Top-k selection queries over relational databases can be mapped into range queries using high dimensional histograms [1]. In [13], top-k selection queries are evaluated in relational query processors by introducing a new pipelined join operator termed *NRA-RJ*. NRA-RJ modifies the NRA algorithm [8] to work on ranges of scores instead of requiring the input to have exact scores. NRA-RJ is an efficient rank-join query operator that joins multiple ranked inputs based on a key-equality condition and cannot handle general join conditions. In [13], it is shown both analytically and experimentally that NRA-RJ is superior to J^* for equality join conditions on key attributes.

3 An Overview on Ripple Join

Ripple join is a family of join algorithms introduced in [11] in the context of online processing of aggregation queries in a relational DBMS. Traditional join algorithms are designed to minimize the time till completion. However, ripple joins are designed to minimize the time till an acceptably precise estimate of the query result is available. Ripple joins can be viewed as a generalization of nested-loops join and hash join. We briefly present the basic idea of ripple join below.

In the simplest version of a two-table ripple join, one previously-unseen random tuple is retrieved from each table (e.g., R and S) at each sampling step. These new tuples are joined with the previously-seen tuples and with each other. Thus the Cartesian product $R \times S$ is swept out as depicted in Figure 2.

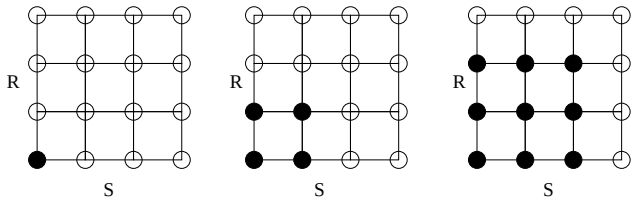


Figure 2: Three steps in Ripple Join

The *square* version of ripple join draws samples from R and S at the same rate. However, in order to provide the shortest possible confidence intervals, it is often necessary to sample one relation at a higher rate. This requirement leads to the general *rectangular* version of the ripple join where more samples are drawn from one relation than from the other. Variants of ripple join are: (1) *Block Ripple Join*, where the sample units are blocks of tuples of size b (In classic ripple join, $b = 1$), (2) *Hash Ripple Join*, where all the sampled tuples are kept in hash tables in memory. In this case, calculating the join condition of a new sampled tuple with previously sampled tuples is very fast (saving I/O). The second variant is exactly the *symmetric hash join* [12, 19] that allows a high degree of pipelining in parallel databases. When the hash tables grow in size and exceed memory size, the hash ripple join falls back to block ripple join.

4 Supporting Top-k Join Queries

In this section we address the problem of supporting *top-k join queries*. We start by defining the query model and present our approach to support evaluating this type of queries in relational query engines.

4.1 Query Model

In traditional relational systems the answer to a join query is a set of $m - tuple$ records, where m is the number of joined relations and each join result is a new tuple that consists of the concatenation of the tuples from the joined relations. There is no order requirement imposed on the join results although the join technique may be able to preserve partial orders of the inputs. In contrast, the answer to a *top-k join query* is an ordered set of join results according to some provided function that combines the orders on each input.

4.2 The New Rank Join Algorithm

Current implementations of the join operator do not make use of the fact that the inputs may be already ordered on their individual scores. Using these individual orderings, we can perform much better in evaluating the top-k join queries by eliminating the need to sort the join results on the combined score.

The join operation can be viewed as the process of spanning the space of Cartesian product of the input relations to get valid join combinations. An important observation is that, only part of this space needs to be computed to evaluate *top-k join queries*, if we have the inputs ordered individually.

In this section we describe a new join algorithm, termed *rank-join*. The algorithm takes m ranked inputs, a join condition, a *monotone* combining ranking function f and the number of desired ranked join results k. The algorithm reports the top k ranked join results in descending order of their combined score. The rank-join algorithm works as follows:

- Retrieve objects from the input relations in a descending order of their individual scores. For each new retrieved tuple:
  1. Generate new valid join combinations with all tuples seen so far from other relations, using some join strategy.
  2. For each resulting join combination, J, compute the score $J.score$ as $f(O_1.score, O_2.score, \ldots, O_m.score)$, where $O_i.score$ is the score of the object from the i^{th} input in this join combination.
  3. Let the object $O_i^{(d_i)}$ be the last object seen from input i, where d_i is number of objects retrieved from that input, $O_i^{(1)}$ be the first object retrieved from input i, and T be the maximum of the following m values: $f(O_1^{(d_1)}.score, O_2^{(1)}.score, \ldots, O_m^{(1)}.score)$, $f(O_1^{(1)}.score, O_2^{(d_2)}.score, \ldots, O_m^{(1)}.score)$, $\ldots$, $f(O_1^{(1)}.score, O_2^{(1)}.score, \ldots, O_m^{(d_m)}.score)$.
  4. let L_k be a list of the k join results with the maximum combined score seen so far and let $score_k$ be the lowest score in L_k, halt when $score_k \geq T$.
- Report the join results in L_k ordered on their combined scores.

The value T is an upper-bound of the scores of any join combination not seen so far. An object O_i^p, where $p > d_i$, not seen yet from input i, cannot contribute to any join combination that has a combined score greater than or equal $f(O_1^{(1)}.score, \ldots, O_i^{(d_i)}.score, \ldots, O_m^{(1)}.score)$. The value T is continuously updated with the score of the newly retrieved tuples.

Theorem 4.2.1: *Using a monotone combining function, the described rank-join algorithm correctly reports the top k join results ordered on their combined score.*

Proof: For simplicity, we prove the algorithm for two inputs l and r. The proof can be extended to cover the m inputs case. We assume that the algorithm access the same number of tuples at each step, i.e., $d_1 = d_2 = d$. The two assumptions do not affect the correctness of the original algorithm.

The proof is by contradiction. Assume that the algorithm halts after d sorted accesses to each input and reports a join combination $J_k = (O_l^{(i)}, O_r^{(j)})$, where $O_l^{(i)}$ is the i^{th} object from the left input and $O_r^{(j)}$ is the j^{th} object from the right input. Since the algorithm halts at depth d, we know that $J_k.score \geq T^{(d)}$, where $T^{(d)}$ is the maximum of $f(O_l^{(1)}.score, O_r^{(d)}.score)$ and $f(O_l^{(d)}.score, O_r^{(1)}.score)$. Now assume that there exists a join combination $J = (O_l^{(p)}, O_r^{(q)})$ not yet produced by the algorithm and $J.score > J_k.score$. That implies $J.score > T^{(d)}$, i.e.,

$$f(O_l^{(p)}.score, O_r^{(q)}.score) > f(O_l^{(1)}.score, O_r^{(d)}.score) \tag{1}$$

and

$$f(O_l^{(p)}.score, O_r^{(q)}.score) > f(O_l^{(d)}.score, O_r^{(1)}.score) \tag{2}$$

Since each input is ranked in descending order of object scores, then $O_l^{(p)}.score \leq O_l^{(1)}.score$. Therefore, $O_r^{(q)}.score$ must be greater than $O_r^{(d)}.score$. Otherwise, Inequality (1) will not hold because of the monotonicity of the function f. We conclude that $O_r^{(q)}$ must appear before $O_r^{(d)}$ in the right input, i.e.,

$$q < d \tag{3}$$

Using the same analogy, we have $O_r^{(q)}.score \leq O_r^{(1)}.score$. Therefore, $O_l^{(p)}.score$ must be greater than $O_l^{(d)}.score$. Otherwise, Inequality (2) will not hold because of the monotonicity of the function f. We conclude that $O_l^{(p)}$ must appear before $O_l^{(d)}$ in the left input, i.e.,

$$p < d \tag{4}$$

From (3) and (4), if valid, the combination $J = (O_l^{(p)}, O_r^{(q)})$ must have been produced by the algorithm, which contradicts the original assumption. ∎

Theorem 4.2.1: *The buffer maintained by the rank-join algorithm to hold the ranked join results is bounded and has a size that is independent of the size of the inputs.*

Proof: Other than the space required to perform the join, the algorithm needs only to remember the top k join results independent of the size of the input. ∎

Following this abstract description of the rank-join algorithm, we show how to implement the algorithm in a binary pipelined join operator that can be integrated in commercial query engines. Theoretically, any current join implementation can be augmented to support the previously described algorithm. Practically, the join technique greatly affects the performance of the ranking process. We show the effect of the selection of the join strategy on the stopping criteria of the rank-join algorithm.

4.3 The Effect of Join Strategy

The order in which the points in the Cartesian space are checked as a valid join result has a great effect on the stopping criteria of the rank-join algorithm. Consider the two relations in Figure 3 to be joined with the join condition $L.A = R.A$. The join results are required to be ordered on the combined score of $L.B + R.B$.

id	A	B
1	1	5
2	2	4
3	2	3
4	3	2

L

id	A	B
1	3	5
2	1	4
3	2	3
4	2	2

R

Figure 3: Two example relations

Following the new rank-join algorithm, described in Section 4.2, a threshold value will be maintained as the maximum between $f(L^{(1)}.B, R^{(d_2)}.B)$ and $f(L^{(d_1)}.B, R^{(1)}.B)$, where $L^{(d_1)}$ and $R^{(d_2)}$ are the last tuples accessed from L and R, respectively. Figure 4 shows two different strategies to produce join results.

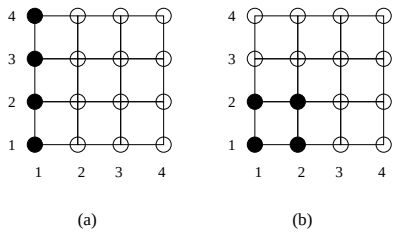


Figure 4: Two possible join strategies.

Strategy (a) is a nested-loops evaluation while Strategy (b) is a symmetric join evaluation that tries to balance the access from both inputs. To check for possible join combinations, Strategy (a) accesses four tuples from L and one tuple from R while Strategy (b) accesses two tuples from each relation. The rank-join algorithm at this stage computes a different threshold value T in both strategies. In Strategy (a), $T = max(5 + 2, 5 + 5) = 10$, while in Strategy (b) $T = max(5 + 4, 5 + 4) = 9$. At this stage, the only valid join combination is the tuple pair $[(1, 1, 5), (2, 1, 4)]$ with a combined score of 9. In Strategy (a), this join combination cannot be reported because of the threshold value of 10 while the join combination is reported as the top-ranked join result according to Strategy (b).

The previous discussion suggests using join strategies that reduce the threshold value as quickly as possible to be able to report top ranked join results early on. In the next section, we present different implementations of the rank-join algorithm by choosing different join strategies.

5 New Physical Rank Join Operators

The biggest advantage of encapsulating the rank-join algorithm in a real physical query operator is that rank-join can be adopted by practical query engines. The query optimizer will have the opportunity to optimize a ranking query by integrating the new operator in ordinary query execution plans. The only other alternative to develop a query operator is to implement the rank-join algorithm as a user defined function. This approach will lose the efforts of the query optimizer to produce a better overall query execution plan. Figure 5 gives alternative execution plans to rank-join three ranked inputs.

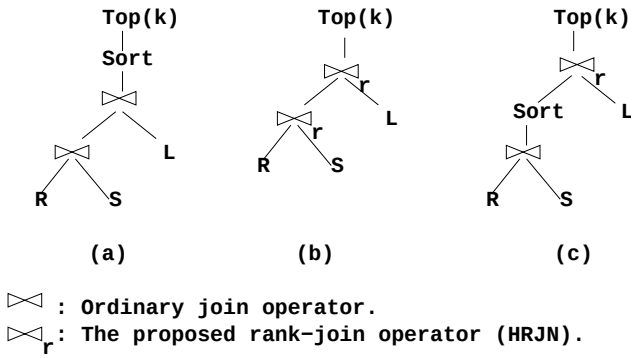


Figure 5: Alternative execution plans to rank-join three ranked inputs.

In this section, we present two alternatives to realize the new rank-join algorithm as a physical join operator. The main difference between the two alternatives is in the join strategy that is used in order to produce valid join combinations. Reusing the current join strategies (nested-loops join, merge join and hash join) results in a poor performance. Nested-loops join will have a high threshold value because we access all the tuples of the inner relation for only one tuple from the outer relation. Merge join requires sorting on the join columns (not the scores) of both inputs and hence cannot be used in the rank-join algorithm. Similarly,

```
Open(L,R,C,f)
input L,R: Left and right ranked input
       C: join condition.
       f: monotone combining ranking function.
begin
  Allocate a priority queue Q;
  Build two hash tables for L and R;
  Set the join condition to C;
  Set the combining function to f;
  Threshold = 0;
  L.Open();
  R.Open();
end
```

Table 1: The HRJN *Open* operation

.

hash join destroys the order through the use of hashing when hash tables exceed memory size. The join strategies presented here depend on balancing the access of the underlying relations.

Since the join operation is implemented in most systems as a dyadic (2-way) operator, we describe the new operators as *binary* join operators. Following common query execution models, we describe the new physical join operators in terms of the three basic interface methods *Open*, *GetNext* and *Close*. The *Open* method initializes the operator and prepares its internal state, the *GetNext* method reports the next ranked join result upon each call, and the *Close* method terminates the operator and performs the necessary clean up.

In choosing the join strategy, the discussion in Section 4.3 suggests sweeping the Cartesian space in a way that reduces the threshold value. We depend on the idea of ripple join as our join strategy. Instead of randomly sampling tuples from the input relations, the tuples are retrieved in order to preserve ranking. One challenge is to determine the rate at which tuples are retrieved from each relation. We present two variants of our rank-join algorithm. The two variants are based on adopting two ripple join variants: the hash ripple join and the block ripple join.

5.1 Hash Rank Join Operator (HRJN)

HRJN can be viewed as a variant of the *symmetrical hash join* algorithm [12, 19] or the hash ripple join algorithm [11]. The *Open* method is given in Table 1. The HRJN operator is initialized by specifying four parameters: the two inputs, the join condition, and the combining function. Any of the two inputs or both of them can be another HRJN operator [1]. The join condition is a general equality condition to evaluate valid join combinations. The combining function is a monotone function that computes a global score from the scores of each input. The *Open* method sets the state and creates the operator internal state which 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. The *Open* method also calls the initialization methods of the inputs.

The *GetNext* method encapsulates the rank-join algorithm and is given in Table 2. The algorithm maintains a *threshold* value that gives an upper-bound of the score of all join combinations not yet seen. To compute the threshold, the algorithm remembers the two top scores and the two bottom scores (last scores seen) of its inputs. These are the variables L_{top}, R_{top}, L_{bottom} and R_{bottom}, respectively. L_{bottom} and R_{bottom} are continuously updated as we retrieve new tuples from the input relations. At any time during execution, the threshold upper-bound value (T) is computed as the maximum of $f(L_{top}, R_{bottom})$ and $f(L_{bottom}, R_{top})$.

The algorithm starts by checking if the priority queue holds any join results. If exists, the score of the top join result is checked against the computed threshold. A join result is reported as the next *GetNext* answer if the join result has a combined score greater than or equal the threshold value. Otherwise, the algorithm

[1] Because HRJN is symmetric, we can allow pipelined bushy query evaluation plans.

```
GetNext()
output : Next ranked join result.
begin
  if (Q is not empty)
    tuple = Q.Top;
    if (tuple.score ≥ T)
      return tuple;
  Loop
   Determine next input to access, I; (Section 5.3)
   tuple= I.GetNext();
   if (I_firstTuple)
     I_top = tuple.score;
     I_firstTuple = false;
   I_bottom = tuple.score;
   T = MAX(f(L_top, R_bottom), f(L_bottom, R_top));
   insert tuple in I Hash table;
   probe the other hash table with tuple;
   For each valid join combination
     Compute the join result score using f;
     Insert the join result in Q;
   if (Q is not empty)
     tuple = Q.Top;
     if (tuple.score ≥ T)
       break loop;
  End Loop;
  Remove tuple from Q;
  return tuple;
end
```

Table 2: The HRJN *GetNext* operation.

continues by reading tuples from the left and right inputs and performs a symmetric hash join to generate new join results. For each new join result, the combined score is obtained and the join result is inserted in the priority queue. In each step, the algorithm decides which input to poll. This gives the flexibility of optimizing the operator to get faster results depending on the joined data. A straight forward strategy is to switch between left and right input at each step.

5.2 Local Ranking in HRJN

Implementing the rank-join algorithm as a binary pipelined query operator raises several issues. We summarize the differences between HRJN and the logical rank-join algorithm as follows:

- The total space required by HRJN is the sum of two hash tables and the priority queue. In a system that supports symmetrical hash join, the extra space required is only the size of the priority queue of join combinations. As shown in Section 4.2, in the proposed rank-join algorithm (with all inputs processed together), the queue buffer is bounded by k, the maximum number of ranked join results that the user asks for. In this case, the priority queue will hold only the top-k join results. Unfortunately, in the implementation of the algorithm as a pipelined query operator, we can only bound the queue buffer of the top HRJN operator since we do not know in advance how many partial join results will be pulled from the lower-level operators. The effect of pipelining on the performance is addressed in the experiments in Section 7.

- Realizing the algorithm in a pipeline introduces a computational overhead as the number of pipeline stages increases. To illustrate this problem, we elaborate on how HRJN works in a pipeline of three input streams, say L_1, L_2 and L_3. When the top HRJN operator, OP_1, is called for the next top

ranked join result, several *GetNext* calls from the left and right inputs are invoked. According to the HRJN algorithm, described in Table 2, at each step, OP_1 gets the next tuple from its left and right inputs. Hence, OP_2 will be required to deliver as many top partial join results of L_2 and L_3 as the number of objects retrieved by L_1. These excessive calls to the ranking algorithm in OP_2 result in retrieving more objects from L_2 and L_3 than necessary, and accordingly larger queue sizes and more database accesses. We call this problem the *Local Ranking* problem.

Solving The Local Ranking Problem Another version of ripple join is the blocked ripple join [11]. At each step, the algorithm retrieves a new block of one relation, scans all the old tuples of the other relation, and joins each tuple in the new block with the corresponding tuples there. We utilize this idea to solve the local ranking problem by unbalancing the retrieval rate of the inputs. We issue less expensive *GetNext* calls to the input with more HRJN operators in its subtree of the query plan. For example, in a left-deep query execution plan, for each p tuples accessed from the right input, one tuple is accessed from the left input. The idea is to have less expensive *GetNext* calls to the left child, which is also an HRJN operator. This strategy is analogous to the block ripple join algorithm, having the left child as an outer and the right child as inner with a block of size p. Using different depths in the input streams does not violate the correctness of the algorithm, but will have a major effect on the performance. This optimization significantly enhances the performance of the HRJN operator as will be demonstrated in Section 7. Through the rest of this paper, we call p the *balancing factor*. Choosing the right value for p is a design decision and depends on the generated query plan, but a good choice of p boosts the performance of HRJN.

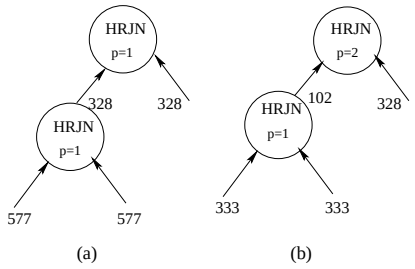


Figure 6: The effect of applying the heuristic to solve the local ranking problem in HRJN.

For example, in a typical query with three ranked inputs, we compare between the total number of accessed tuples by the HRJN operator before and after applying the heuristic. Figure 6 shows the number of retrieved tuples for each case. In the plan in Figure 6 (a), p is set to 1 for both HRJN operators. This query pipeline is applied on real data to retrieve the top 50 join results. The top HRJN operator retrieves 328 tuples from both inputs, hence the top 328 partial join results are requested from the HRJN child operator. The child HRJN operator has to retrieve 577 tuples from each of its inputs, for a total of 1482 tuples. In the plan in Figure 6 (b), p is set to 2 for the top HRJN operator. While retrieving the same answers, the total number of tuples retrieved is 994 tuples, which is much less than that of the HRJN before applying the heuristic, since the top HRJN operator requested only 102 tuples from its left child.

5.3 $HRJN^*$: Score-Guided Join Strategy

As discussed in Section 4.3, the way the algorithm schedules the next input to be polled can affect the operator response time significantly. One way is to switch between the two inputs at each step. However, this balanced strategy may not be the optimal. Consider the two relations L and R to be rank-joined. The scores from L are $100, 50, 25, 10 \ldots$ while the scores from R are $10, 9, 8, 5, \ldots$. After 6 steps using a balanced strategy (three tuples from each input) we will have the threshold of $max(108, 35) = 108$. On the other hand, favoring R by retrieving more tuples from R than L (four tuples from R and two tuples from L) will give a threshold of $max(105, 60) = 105$.

One heuristic is to try to narrow the gap between the two terms in computing the threshold value. Recall that the threshold is computed as the maximum between two virtual scores: $T_1 = f(L_{top}, R_{bottom})$ and

$T_2 = f(L_{bottom}, R_{top})$, where f is the ranking function. If $T_1 > T_2$ more inputs should be retrieved from R to reduce the value of T_1 and hence the value of the threshold, leading to possible faster reporting of ranked join results.

This heuristic will cause the join strategy to adaptively switch between the hash join and nested-loops join strategies. Consider the previous example, since $T_1 > T_2$, more tuples will be retrieved from R till the end of that relation. In this case, L_{top} can be reduced to 50. In fact, because all the scores in L are significantly higher than R, the strategy will behave exactly like a nested-loops join. On the other extreme, if the scores from both relations are close, the strategy will behave as a symmetric hash join with equal retrieval rate. Between the two extremes, the strategy will gracefully switch between nested-loops join and hash join to reduce the threshold value as quickly as possible. Of course, this heuristic does not consider the I/O and memory requirements that may prefer one strategy on the other. In the experimental evaluation of our approach, discussed in Section 7, we implement the new join strategy using the HRJN operator. We call the enhanced operator $HRJN^*$. $HRJN^*$ shows better performance than those of other rank-join operators including the original HRJN.

5.4 An Adaptive Join Strategy

When inputs are from external sources, one of the inputs may stall for some time. An adaptive join algorithm makes use of the tuples retrieved from the other input to produce valid join results. This processing environment is common in applications that deal with ranking, e.g., a mediator over web-accessible sources and distributed multimedia repositories.

In these variable environments, the join strategy of the rank-join operators may use input availability as a guide instead of the aforementioned score-guided strategy. If both inputs are available, the operator may choose the next input to process based on the retrieved scores. Otherwise, the available input is processed. *HRJN* can be easily adapted to use XJoin [18]. *XJoin* is a practical adaptive version of the symmetric hash join operator. The same *GetNext* interface will be used with the only change that the next input to poll is determined by input availability and rate. The adaptive version of *HRJN* will inherit the adaptability advantage of the underlying XJoin strategy with the added feature of supporting top-k join queries over external sources.

6 Generalizing Rank-Join to Exploit Random Access Capabilities

The new rank-join algorithm and query operators assume only sorted access to the input. Random access to some of these inputs is possible when indexes exist. Making use of these indexes may give better performance depending on the type of the index and the selectivity of the join operation. We would like to give the optimizer the freedom to choose whether to use indexes given the necessary cost parameters.

In this section, we generalize the rank-join algorithm to make use of the random access capabilities of the input relations. The main advantage to using random access is to further reduce the upper-bound of the score of unseen join combinations, and hence being able to report the top-k join results earlier. For simplicity, we present the algorithm by generalizing the HRJN operator to exploit the indexes available on the join columns of the ranked input relations. Consider two relations L and R, where both L and R support sorted access to their tuples. Depending on index existence, we have two possible cases. The first case is when we have an index on only one of the two inputs, e.g., R. Upon receiving a tuple from L, the tuple is first inserted in L's hash table and is used to probe the R index. This version can be viewed as a hybrid between hash join and index nested-loops join. The second case is when we have an index on each of the two inputs. Upon receiving a tuple from $L(R)$, the tuple is used to probe the index of $R(L)$. In this case, there is no need to build hash tables.

On-the-fly Duplicate Elimination The generalization, as presented, may cause duplicate join results to be reported. We eliminate the duplicates on-the-fly by checking the combined score of the join result against the upper-bound of the scores of join results not yet produced. Consider the two relations L and R with an index on the join column of R. A new tuple from L, with score L_{bottom}, is used to probe R's index and generate all valid combinations. A new tuple from R, with score R_{bottom}, is used to probe the L's hash table of all *seen* tuples from L. A key observation is that any join result, not yet produced, cannot have

a combined score greater than $U = f(L_{bottom}, R_{bottom})$. Notice that L_{bottom} is an upper-bound of all the scores from L not yet seen. All join combinations with scores greater than U were previously generated by probing R's index. Hence, A duplicate tuple can be detected and eliminated on-the-fly if it has a combined score greater than U. A similar argument holds for the case when both L and R have indexes on the join columns. One special case is when the two new tuples from L and R can join. In this case, only one of them is used to probe the other relation.

Faster Termination Although index probing looks similar to hash probing in the original HRJN algorithm in Table 2, it has a significant effect on the threshold values. The reason is that since the index contains all the tuples from the indexed relation (e.g., L), the tuple that probes the index from the other relation (e.g., R) cannot contribute to more join combinations. Consequently, the top value of Relation R should be decreased to the score of the next tuple. For example, for the two ranked relations L and R in Figure 3, assume that relation R has an index on the join column to be exploited by the algorithm. In the first step of the algorithm, the first tuple from L is retrieved:$(1, 1, 5)$. We use this tuple to probe the index of R, the resulting join combination is $[(1, 1, 5), (2, 1, 4)]$. Since the tuple from L cannot contribute to other join combinations, we reduce the value L_{top} to be that of the next tuple $(2, 2, 4)$, i.e., 4. In this case we always have $L_{top} = L_{bottom}$ which may reduce the threshold value $T = max(L_{top} + R_{bottom}, L_{bottom} + R_{top})$. Note that if no indexes exists, the algorithm behaves exactly like the original HRJN algorithm.

7 Performance Evaluation

In this section, we compare the two rank-join operators, HRJN and HRJN* introduced in Section 5, with another rank-join operator based on the J^* algorithm. The experiments are based on our research platform for a complete video database management system (VDBMS) running on a Sun Enterprise 450 with 4 UltraSparc-II processors running SunOS 5.6 operating system. The research platform is based on PREDATOR [17], the object relational database system from Cornell University. The database tables have the schema *(Id, JC, Score, Other Attributes)*. Each table is accessed through a sorted access plan and tuples are retrieved in a descending order of the *Score* attribute. *JC* is the join column (not a key) having D distinct values.

We use a simple ranking query that joins four tables on the non-key attribute JC and retrieves the join results ordered on a simple function. The function combines individual scores which in this case a weighted sum of the scores (w_i is the weight associated with input i). Only the top k results are retrieved by the query. The following is a SQL-like form of the query:

Q:
```
SELECT T1.id, T2.id, T3.id, T4.id
FROM T1, T2, T3, T4
WHERE T1.JC=T2.JC and
      T2.JC=T3.JC and
      T3.JC=T4.JC
ORDER BY w1*T1.Score + w2*T2.Score +
         w3*T3.Score + w4*T4.Score
STOP AFTER k;
```

One pipelined execution plan for the query **Q** is the left-deep plan, *Plan A*, given in Figure 7. We limit the number of reported answers to k by applying the *Stop-After* query operator [3, 4]. The operator is implemented in the prototype as a physical query operator *Scan-Stop*, a straightforward implementation of Stop-After and appears on top of the query plan. *Scan-Stop* does not perform any ordering on its input.

7.1 A Pipelined Bushy Tree

Plan A is a typical pipelined execution plan in current query optimizers. *Plan B* is a bushy execution plan given in Figure 8. Note that bushy plans are not pipelined in current query processors because of the current join implementations. Because rank-join is a symmetric operation, a bushy execution plan can also be pipelined. The optimizer chooses between these plans depending on the associated cost estimates.

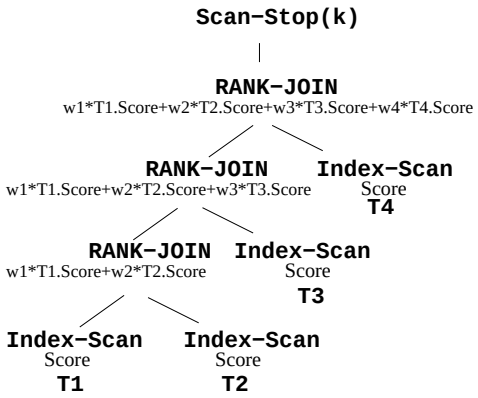


Figure 7: *Plan A*: A left-deep execution plan for **Q**.

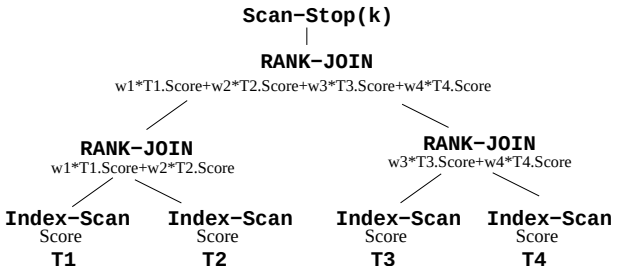


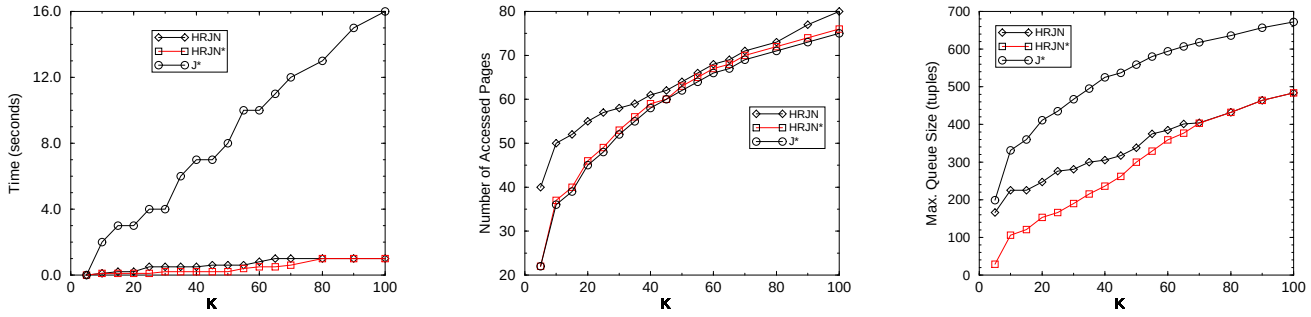
Figure 8: *Plan B*: A bushy execution plan for **Q**.

Plan B does not suffer from the local ranking problem, described in Section 5.2, because each operator has almost the same cost for accessing both of its inputs (same number of plan levels). However, having large variance of the score values between inputs, retrieving more inputs from one side may result in a faster termination. This is a typical case where the operator $HRJN^*$ can perform better, because $HRJN^*$ uses input scores to guide the rate at which it retrieves tuples from each input.

7.2 Comparing the Rank-Join Operators

In this section, we evaluate the performance of the introduced operators by comparing them with each other and with a rank-join operator based on the J^* algorithm [14]. We limit our presentation to comparing three rank-join operators: the basic HRJN operator, the $HRJN^*$ operator and the J^* operator. HRJN applies the basic symmetric hash join strategy; at each step one tuple is retrieved from each input. The local ranking minimization heuristic, proposed in Section 5.2, is applied in HRJN. The $HRJN^*$ operator uses the score-guided strategy, proposed in Section 5.3, to determine the rate at which it retrieves tuples from both inputs. The J^* operator is an implementation of the J^* algorithm. We do not compare with the naïve approach of joining the inputs then sorting since all the rank-join algorithms give a better performance by orders of magnitude. We choose four performance metrics: the total time to retrieve k ranked results, the total number of accessed disk pages, the maximum queue size, and the total occupied space. In the following experiments, we use *Plan A* as the execution plan for **Q**. Using *Plan B* gave similar performance results.

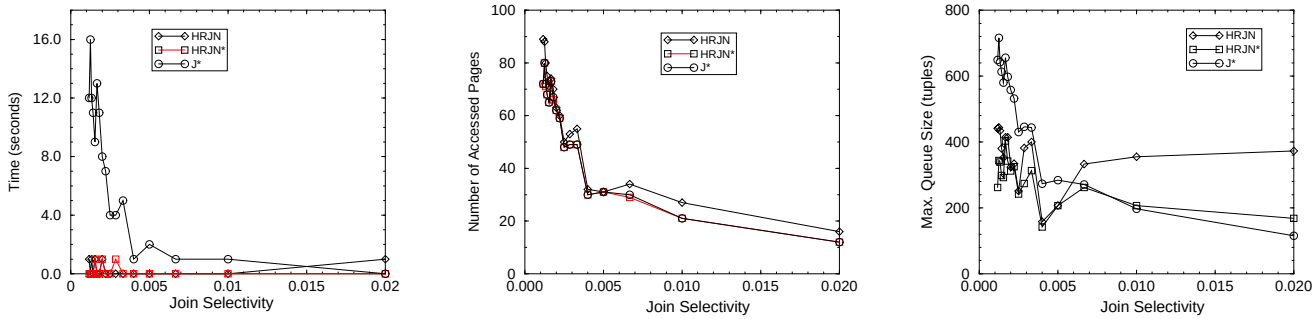
Changing the number of required answers In this experiment, we vary the number of required answers, k, from 5 to 100 while fixing the join selectivity to 0.2%. Figure 9 (a) compares the total time to evaluate the query. HRJN and $HRJN^*$ show a faster execution by an order of magnitude for large values of k. The high CPU complexity of the J^* algorithm is because it retrieves one join combination in each step. In each step, J^* tries to determine the next optimal point to visit in the Cartesian space. Since both HRJN and $HRJN^*$ use symmetric hash join to produce valid join combinations, more join combinations are ranked at each



(a) (b) (c)

Figure 9: Comparing $HRJN$, J^* and $HRJN^*$ for $m = 4$ and selectivity = 0.2%.

step. Figure 9 (b) compares the number of accessed disk pages. The three algorithms have a comparable performance in terms of the number of pages retrieved. J^* and $HRJN^*$ achieve better performance because retrieving a new tuple is guided by the score of the inputs, which makes both algorithms retrieve only the tuples that makes significant decrease in the threshold value and hence less I/O. Figure 9 (c) compares the number of maintained buffer space. HRJN and $HRJN^*$ have low space overhead because they use the buffer only for ranking the join combinations, while J^* maintains all the retrieved tuples in its buffer. Had we also included the space of the hash tables, J^* will have a lower overall space requirement. In most practical systems the hash space is already reserved for hash join operations. Hence, the space overhead is only the buffer needed for ranking.

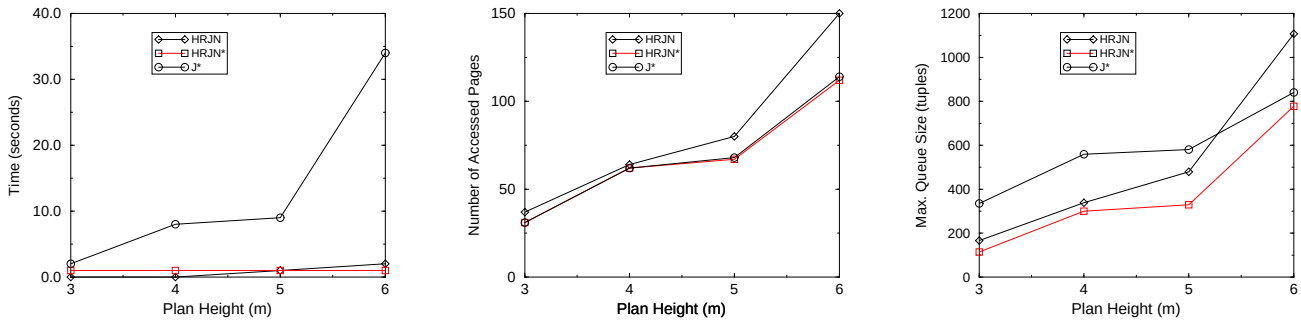


(a) (b) (c)

Figure 10: The effect of selectivity on $HRJN$, J^* and $HRJN^*$ for $m = 4$ and $K = 50$.

Changing the join selectivity In this experiment, we fix the value of k to 50 and vary the join selectivity from 0.12% to 2%. Figure 10 (a) compares the total time to report 50 ranked results, while Figures 10 (b) and 10 (c) compare the number of accessed disk pages and the extra space overhead, respectively. For all selectivity values, $HRJN^*$ shows the best performance. J^* has a better performance than $HRJN$ for high selectivity values while HRJN performs better for low selectivity values. The reason is that $HRJN^*$ combines the advantages of J^* and $HRJN$. While $HRJN^*$ uses a score-guided strategy to navigate in the Cartesian space for a faster termination (similar to J^*), it also uses the power of producing fast join results by using the symmetric hash join technique (similar to $HRJN$).

The effect of pipelining In this experiment, we evaluate the scalability of the rank-join operators. We vary the number of join inputs, m, from 3 to 6 and fix $k = 50$ and the join selectivity to 0.2%. Figure 11 (a)



(a) (b) (c)

Figure 11: The effect of pipelining on $HRJN$, J^* and $HRJN^*$ for selectivity 0.2% and $K = 50$.

gives the effect of pipelining on the total query time. HRJN and $HRJN^*$ show much better scalability than that of J^* by orders of magnitude. The CPU complexity of J^* increases significantly as m increases. On the other hand, J^* and $HRJN^*$ show better performance in terms of the number of accessed pages compare to HRJN (Figure 11 (b)), because of the score-guided strategy they are using. $HRJN^*$ is the most scalable in terms of the space overhead as shown in Figure 11 (c).

8 Conclusion

In this paper, we address supporting top-k join queries in practical relational query processors. We introduce a new rank-join algorithm that is independent of the join strategy, along with its correctness proof. The proposed rank-join algorithm makes use of the ranking on the input relations to produce ranked join results on a combined score. The ranking is performed progressively during the join and hence, there is no need for a blocking sort operation after join. We present a physical query operator to implement rank-join based on ripple join; the hash rank join (HRJN). We propose a new join strategy that is guided by the input score values. We apply the new strategy on the original HRJN algorithm and call the new operator $HRJN^*$. We address exploiting available indexes on the join columns. We propose a general rank-join algorithm that utilizes these indexes for faster termination of the ranking process. We experimentally evaluate the proposed join operators and compare their performance with a recent algorithm to join ranked inputs. We conduct several experiments varying the number of required answers, the join selectivity, and the number of inputs in the pipeline. The experiments prove the concept and show a significant performance enhancement, especially for low values of join selectivity.

References

[1] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *TODS*, 27(2), 2002.

[2] Nicolas Bruno, Luis Gravano, and Amelie Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, 2002.

[3] Michael J. Carey and Donald Kossmann. On saying "Enough already!" in SQL. In *SIGMOD, Tucson, Arizona*, May 1997.

[4] Michael J. Carey and Donald Kossmann. Reducing the braking distance of an SQL query engine. In *VLDB, New York, August*, 1998.

[5] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*, 2002.

[6] Cynthia Dwork, S. Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the web. In *World Wide Web*, 2001.

[7] Ronald Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences (JCSS)*, 58(1), Feb 1999.

[8] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS, Santa Barbara, California*, May 2001.

[9] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In *VLDB, September 10–14, Cairo, Egypt*, 2000.

[10] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, 2001.

[11] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD, Philadelphia, Pennsylvania, USA*, june 1999.

[12] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, 1(1), Jan. 1993.

[13] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Joining ranked inputs in practice. In *VLDB, Hong Kong, China*, 2002.

[14] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In *VLDB, Rome, Italy*, 2001.

[15] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE, Sydney, Austrialia*, 1999.

[16] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path election in a relational database management system. In *SIGMOD*, 1979.

[17] Praveen Seshadri and Mark Paskin. Predator: An or-dbms with enhanced data types. In *SIGMOD, May 13-15, Tucson, Arizona, USA*, 1997.

[18] T. Urhan and M. J. Franklin. XJoin: A reactively- scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2), February 2000.

[19] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1), 1993.