The VLDB Journal (2004) 13: 207–221 / Digital Object Identifier (DOI) 10.1007/s00778-004-0128-2

Supporting top-k *join* queries in relational databases*

Ihab F. Ilyas[1], **Walid G. Aref**[2], **Ahmed K. Elmagarmid**[2]

[1] School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada (e-mail : ilyas@uwaterloo.ca)
[2] Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, USA (e-mail: {aref,ake}@cs.purdue.edu)

Edited by S. Abiteboul. Received: December 23, 2003 / Accepted: March 31, 2004
Published online: August 12, 2004 –

Abstract. Ranking queries, also known as top-k 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. Top-k queries are dominant in many emerging applications, e.g., multimedia retrieval by content, Web databases, data mining, middlewares, and most information retrieval applications. 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 nonblocking and can be integrated into pipelined execution plans. We also propose an efficient heuristic designed to optimize a top-k join query by choosing the best join order. 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.

Keywords: Ranking – Top-k queries – Rank aggregarion – Query operators

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 metasearch engines, combining ranking functions and selecting documents based on multiple criteria [8]. 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 scoring 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.

The following examples illustrate possible scenarios for top-k join queries and highlight their challenges to current relational database systems.

Example 1. Consider a video database system where several visual features are extracted from each video object (frames or segments). Example features include color histograms, color layout, texture, and edge orientation. Features are stored in separate relations and are indexed using high-dimensional indexes that support similarity queries. Suppose that a user is interested in the top 10 video frames most similar to a given query image based on color. The top-k query is translated into a similarity query (a nearest-neighbor query) using the high-dimensional index on the color feature. Only the top 10 results are presented to the user.

We call the above query a *single-feature* or a *single-criterion* ranking. Answering a single-criterion ranking query does not require any join. A database system that supports approximate matching ranks the tuples depending on how well they match the query according to some similarity measure.

Example 2. In example 1, suppose that the user is interested in the top 10 video frames most similar to the given query image based on color and texture combined. The user also provides a function for how to combine the ranking according to each feature in an overall ranking. For example, the global rank of

* Extended version of the paper published in the Proceedings of the 29th International Conference on Very Large Databases, VLDB 2003, Berlin, Germany, pp 754–765

a frame $= 0.5 \times rank(color) + 0.5 \times rank(texture)$, where $rank(F)$ of a frame (v) is the rank of v among all video frames with respect to the similarity of v to the query image, based on feature F.

We refer to the query in example 2 as a *multicriteria* ranking query, or simply a top-k join query. Unlike single-criterion ranking, in top-k join queries, the database query processor combines (joins) the individual rankings into one global ranking by applying the provided rank-combining function.

Example 3. Consider a user interested in finding a location (e.g., city) where the combined cost of buying a house and paying school tuition in that location is minimum. The user is interested in the top five least expensive places. Assume that there are two external sources (databases) – $Houses$ and $Schools$ – that can provide information on houses and schools, respectively. The $Houses$ database can provide a ranked list of the cheapest houses and their locations. Similarly, the $Schools$ database can provide a ranked list of the least expensive schools and their locations.

A naïve way to answer the user query in example 3 is to retrieve two lists: a list of the cheapest houses from $Houses$ and a list of the cheapest schools from $Schools$. The user then "joins" the two lists based on location. A valid pair is a house and a school in the same location. For all the join results, the user computes the total cost of each pair, e.g., by adding the house price and the school tuition for 5 years. The five cheapest pairs constitute the final answer to the user query. Unfortunately, the user has to "guess" the size of the input lists that will produce five valid matches. If, after the join operation, there are fewer than five join results, the whole process needs to be restarted with larger input sizes.

In all previous examples, answering the top-k join query can be prohibitively expensive and requires complex *join* and *sorting* operations on large amounts of input data.

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, say, f, that combines individual scores. Note that the score attached to 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 R1, R2, ..., Rm
WHERE join_condition(R1, R2, ..., Rm)
ORDER BY f(R1.score, R2.score, ..., Rm.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 a sort-merge join (MGJN) only the order on the join column can be preserved. In a nested-loops join (NLJN), only the orders on the outer relations are preserved through the join, while in a 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 1a gives an example query execution plan for *Q1*. Following the concept of *interesting orders* [18] 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 and (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.

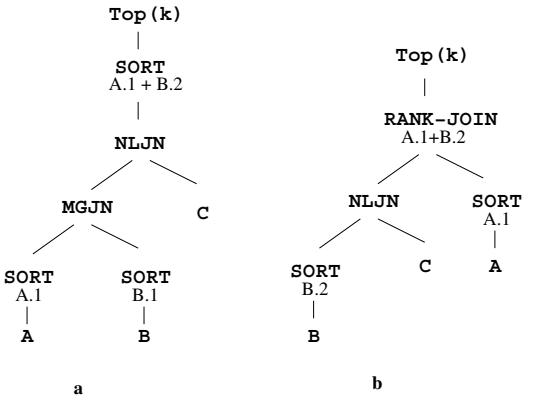


Fig. 1a,b. Alternative plans for query **Q1**

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, and (5) adapt to input fluctuations, a major characteristic in the applications that depend on ranking. We summarize our contribution in this paper as follows:

- We propose a new rank-join algorithm having the above desired properties, along with its correctness proof.
- We analyze the I/O cost of the proposed algorithm and prove its optimality. Our analysis provides strong performance guarantee with respect to the number of required database accesses.
- 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 1b gives an example execution plan for **Q1**, using the proposed rank-join operator (RANK-JOIN). The plan avoids the unnecessary sorting 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 introduce an efficient mechanism for optimizing a top-k join query by determining the best order to perform the binary rank-join operations.
- 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 Sect. 4 we introduce the new rank-join algorithm along with its correctness and optimality proofs. We present two physical rank-join operators in Sect. 5. We introduce an efficient optimization heuristic for rank-join execution plans in Sect. 6. In Sect. 7, we generalize the rank-join algorithm to exploit any available random access to the input relations. Section 8 gives the experimental evaluation of the new rank-join operator and compares it with alternative techniques. We conclude in Sect. 9 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 [9] and Fagin et al. [10] introduce the first efficient set of algorithms to answer ranking queries. Database objects with m attributes are viewed as m separate lists, and each supports sorted and, possibly, random access to object scores. The TA algorithm [10] assumes the availability of random access to object scores in any list besides the sorted access to each list. The NRA algorithm [10] assumes only sorted access is available to individual lists. Similar algorithms are introduced (e.g., see [11,12,17]). 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.

The more general problem of the top-k join is addressed in [16]. 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, more general in using the available access capabilities, and more easily adopted by practical query processors. In our experimental study, we compare our proposed join operators with the J^* algorithm 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. Importance-based join processing [21] is another related problem of producing "important" join results as early as possible; the importance value of a tuple is provided from some sorting attribute. In [21],

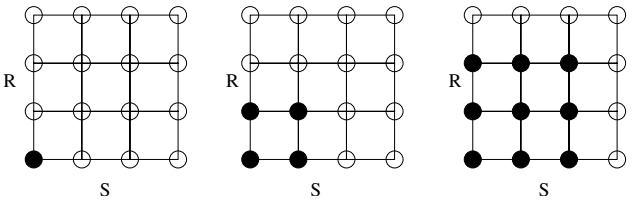


Fig. 2. Three steps in ripple join

important tuples are given higher priority to proceed through the query evaluation plan.

Top-k selection queries over relational databases can be mapped into range queries using high-dimensional histograms [1]. In [15], 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 [10] 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 [15], it is shown both analytically and experimentally that NRA-RJ is superior to J^* for equality join conditions on key attributes.

3 An overview of ripple join

Ripple join is a family of join algorithms introduced in [13] 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 Fig. 2.

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: (i) *block ripple join*, where the sample units are blocks of tuples of size b (in classic ripple join, $b = 1$); (ii) *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* [14,22] 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 supporting evaluation of this type of query 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, and 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 the number of objects retrieved from that input, $O_i^{(1)}$ is the first object retrieved from input i, and T is 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 yet seen from input i, cannot contribute to any join result that has a combined score greater than or equal to $f(O_1^{(1)}.score, \ldots, O_i^{(d_i)}.score, \ldots, O_m^{(1)}.score)$. The value T is continuously updated with the score of the newly retrieved tuples.

Theorem 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 accesses 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.,

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

and

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

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 function f. We conclude that $O_r^{(q)}$ must appear before $O_r^{(d)}$ in the right input, i.e.,

$$q < d\,. \quad (3)$$

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 function f. We conclude that $O_l^{(p)}$ must appear before $O_l^{(d)}$ in the left input, i.e.,

$$p < d\,. \quad (4)$$

From Eqs. 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. □

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

Fig. 3. Two example relations

Theorem 2. *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 need only 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. In practical terms, 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 Effect of the 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 Fig. 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 $L.B + R.B$.

Following the new rank-join algorithm, described in Sect. 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 for producing join results.

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), $T = max(5 +$

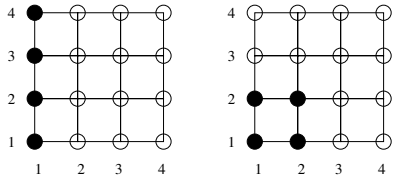


Fig. 4a,b. Two possible join strategies

$2, 5 + 5) = 10$, 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.

4.4 Optimality of algorithm rank-join

In this section, we analyze the I/O cost of the proposed rank-join algorithm. The notion of *instance optimality* is defined by Fagin et al. [10]. Formally, instance optimality is defined as follows. Let $\mathcal{A}$ be a class of algorithms and let $\mathcal{D}$ be a class of databases. For an algorithm $A \in \mathcal{A}$ and a database $D \in \mathcal{D}$, let $cost(A, D)$ be the total number of I/O accesses incurred by applying A on D. An algorithm B is instance optimal over $\mathcal{A}$ and $\mathcal{D}$ if $B \in \mathcal{A}$ and for every $A \in \mathcal{A}$ and $D \in \mathcal{D}$ we have

$$cost(B, D) = O(cost(A, D)).$$

Hence, there exist constants $c, c' > 0$ such that $cost(B, D) \leq c.cost(A, D) + c'$ for every choice of $A \in \mathcal{A}$ and $D \in \mathcal{D}$. The constant c is referred to as the *optimality ratio*.

Theorem 3. *Let $\mathcal{D}$ be the class of all databases consisting of m sorted relations (ranked lists) and let $\mathcal{A}$ be the class of all correct algorithms that produce the top-k ranked join results from these lists. The rank-join algorithm is instance optimal over $\mathcal{A}$ and $\mathcal{D}$.*

Proof. We present the proof in the case of two lists L and R. The proof can be easily generalized to m lists by adjusting the optimality ratio. Refer to Fig. 5 for illustration. Let l be the top element in L with a score s_l and let r be the top element in R with a score s_r.

Assume that the rank-join algorithm, when run on $D \in \mathcal{D}$, halts at depth d. Let $A \in \mathcal{A}$ be an arbitrary algorithm. We shall show that algorithm A must get to depth d in at least one of the lists. It then follows that the rank-join algorithm is instance optimal with an optimality ratio at most 2 (assuming two lists). Assume that algorithm A does not get to depth d in either list; we shall show that algorithm A makes a mistake on some database.

Let $\mathcal{T}$ be the threshold value at depth $d - 1$, where $\mathcal{T}$ is computed as $\mathcal{T} = MAX(f(s_l, q), f(s_r, p))$, where p is the score at depth $d-1$ in L, q is the score at depth $d-1$ in R, and f is the scoring function. Without loss of generality, assume that $f(s_l, q) \geq f(s_r, p)$, hence $\mathcal{T} = f(s_l, q)$.

Since rank-join did not halt at depth $d - 1$, there are less than k joinable pairs (a, b) that have been seen by depth $d-1$ whose overall score is at least $\mathcal{T}$. We now construct a database D' on which algorithm A errs.

Let D' have exactly d elements in each list (so that D' goes only to depth d). Let D' be identical to the original database, D, up to depth $d-1$ in both lists. Hence, algorithm A performs exactly the same on both D and D'. At depth d, we put a new element a' with score p in the first list and a new element b' with score q in the second list such that b' joins with l (the top element in L). Hence, the join results (l, b') has score $f(s_l, q) = \mathcal{T}$.

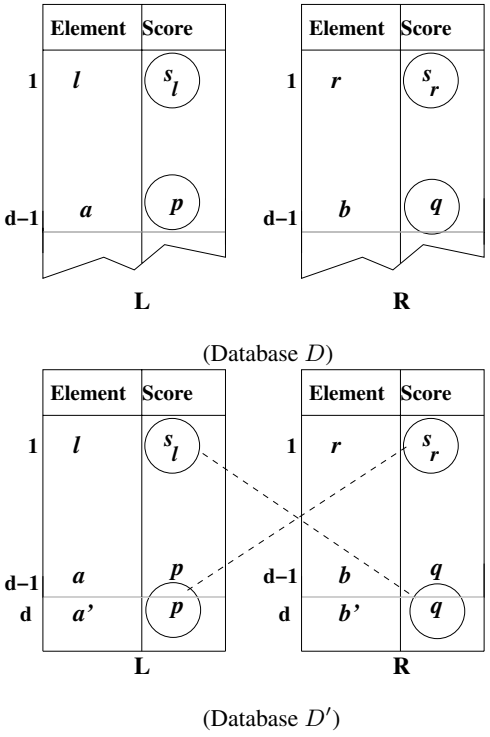


Fig. 5. Instance optimality of rank-join algorithm

Clearly, (l, b') is not on the output list of algorithm A since algorithm A never sees b' before it stops. However, the output list of algorithm A contains k joinable pairs with less than k having score $\geq \mathcal{T}$. So algorithm A made a mistake on D'. □

Since database accesses are the dominant cost factor in querying large databases, the instance optimality of the rank-join algorithm plays an important role in optimizing top-k queries. In practical terms, instance optimality of the rank-join algorithm establishes a strong performance guarantee for the rank-join algorithm when compared with any other way to evaluate top-k queries.

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 developing 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 6 gives alternative execution plans to rank-join three ranked inputs.

In this section, we present two alternatives to realizing the new rank-join algorithm A algorithm as a physical join operator. The main difference between the two alternatives is in the

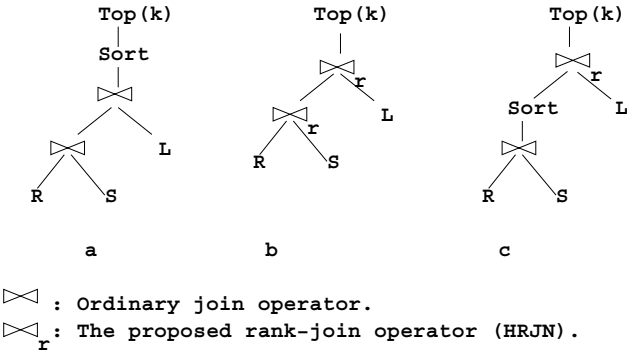


Fig. 6a–c. Alternative execution plans to rank-join three ranked inputs

join strategy that is used 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, 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 (two-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 Sect. 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 [14,22] or the hash ripple join algorithm [13]. 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 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

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

Table 1. The $HRJN$ *Open* operation

```
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 2. The $HRJN$ *GetNext* operation

```
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; (Sect. 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
```

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 one 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 to 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. 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 straightforward 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 Sect. 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 Sect. 8.
- 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 [13]. 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 tuple 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 relation and the right child as an inner relation 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 Sect. 8. For 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 $HRJN$.

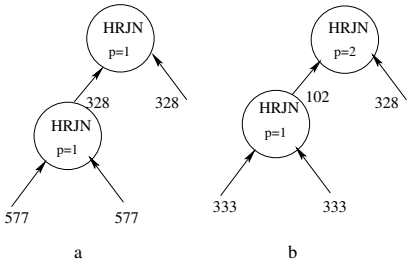


Fig. 7a,b. Effect of applying heuristic to solve local ranking problem in $HRJN$

For example, in a typical query with three ranked inputs, we compare the total number of tuples accessed by the $HRJN$ operator before and after applying the heuristic. Figure 7 shows the number of retrieved tuples for each case. In the plan in Fig. 7a, 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 Fig. 7b, p is set to 2 for the top $HRJN$ operator. While retrieving the same answers, the total number of tuples retrieved is 994, 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 Sect. 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 one. 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 a 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 possibly 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 those in R, the strategy will behave exactly like a nested-loops join. At 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 over the other. In the experimental evaluation of our approach, discussed in Sect. 8, 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 [20]. *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 Choosing the best join order

As described in Sect. 5, the join operation is implemented in most systems as a dyadic (two-way) operator for flexibility and practical implementation reasons. Hence, rank-join operators, e.g., $HRJN$, are implemented as *binary* join physical operators. To rank-join n ranked inputs, the inputs are organized in a pipelined query evaluation plan in the form of a binary tree. The evaluation plan determines the order at which we carry out the rank-join operations. In this section we highlight the effect of join order on the overall performance of top-k join queries. We propose an optimization heuristic to choose the best join order based on sampling.

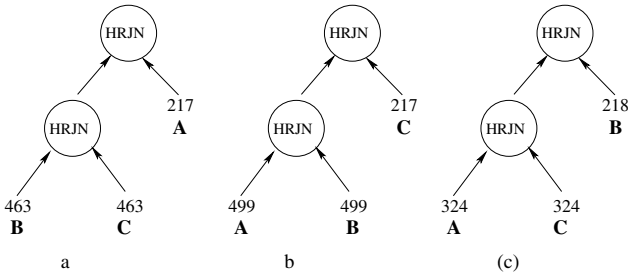


Fig. 8a–c. Effect of join order on size of required input

Consider the following example to join three ranked inputs A, B, and C. Figure 8 gives the number of retrieved records from each input to report the top ten join results. Figure 8 also gives three different join orders (evaluation plans) for the rank-join operations among A, B, and C. The figure shows that the join order significantly affects the number of retrieved records from the inputs. For example, we save a significant number of I/O accesses by changing the join order from plan (b) to plan (c). For plan (b), the number of records retrieved from A, B, and C is 499, 499, and 217, respectively, with a total of 1215 records. For plan (c), the number of records retrieved from A, B, and C is 324, 324, and 218, respectively, with a total of 866 records, which is 70% of the inputs required for plan (b).

The main reason for the effect of the join order on the size of required input – and hence the performance of the rank-join operation – is the *correlation* or the *similarity* among the input rankings. In Fig. 8, the degree of similarity between the rankings of A and C is higher than that between the rankings of A and B. Hence, a rank-join between A and C is likely to require fewer records than a rank-join between A and B to produce the same number of ranked results. Like traditional query optimization, the main goal of optimizing rank-join queries is to choose the best join order. Unlike traditional optimization, the size of the inputs involved in the rank-join operation is not known a priori. Hence it is hard to estimate the cost of a rank-join operation. Figure 9 gives actual total execution time and the number of I/O accesses to rank-join four ranked inputs with six different join orders. The figure shows the significant impact of the order on the overall performance of the rank-join operation.

An optimal query execution plan is the plan with the cheapest overall cost, where the cost includes various components, e.g., the I/O complexity and the memory usage. Since the number of retrieved input records greatly affects the I/O and time complexities of the rank-join operation, we give a definition of an optimal rank-join order. For simplicity, the definition assumes that the I/O cost to retrieve a record is the same for all the inputs. This simplification can be easily relaxed by using a different weight for each input.

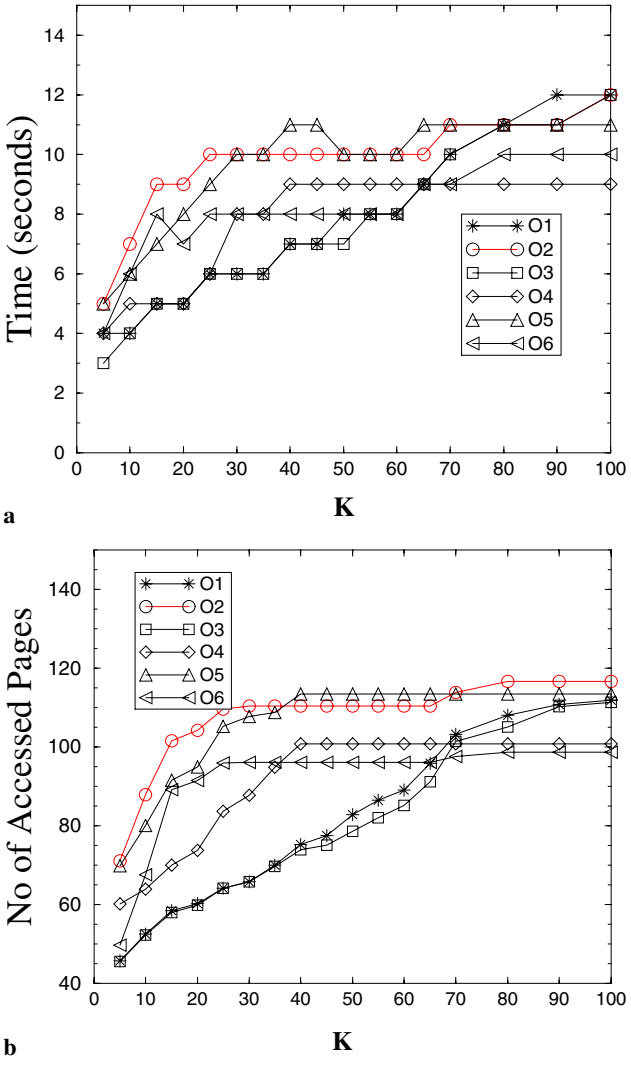


Fig. 9a,b. Effect of join order on performance of rank-join operators

Definition 4 *Optimal rank-join order: A rank-join order is optimal if it requires the least number of input records to produce the same number of ranked join results.*

Computing the optimal rank-join order is hard and even impossible in certain situations for the following reasons:

- Determining the number of input records needed to produce the top-k join results requires a complete knowledge of the score distribution of each input and a well-defined notion of similarity measure among input rankings.
- There is no clear way of estimating the number of required join results, k, when pushed down in a plan pipeline. For example, in plan (c) of Fig. 8, while $k = 10$ for the top $HRJN$ operator, $k = 111$ for the left child $HRJN$ operator. The value of k in each rank-join operator depends on the final value of k (specified in the user query), the rank-join strategy, and the score distribution of the input.
- Assuming that there is a way to compute and use these statistics to estimate the input size, the input itself may not be available offline. For example, the input rankings may be computed as the output of single-feature similarity subqueries (refer to examples 1 and 2).

In the following discussion, we propose a simple yet efficient heuristic to choose a "good" rank-join order based on sampling.

6.1 Rank-join order heuristic

The main idea of the proposed heuristic is to push the rank-join of *similar* rankings as early as possible in the query evaluation plan. We describe the intuition behind this heuristic as follows:

- Since the number of required results from each rank-join operator increases as we go down in the query pipeline, we aim at making early rank-join operations (deep in the query plan) as fast as possible.
- The best-case scenario for the rank-join algorithm occurs when joining identical ranked inputs. We can easily show that the algorithm performance deteriorates, i.e., requires more input before termination, as the similarity between the input rankings decreases.

The proposed technique depends on two main steps: first, obtaining a *ranked* sample of size S from each input, and second, having a well-defined notion for the similarity between two rankings. The first step depends on the type of the input rankings. In general, input rankings can be in one of the following two forms:

- *Available offline as regular database relations*: In this case the ranked sample is the top S records from the inputs and is available statically without the need to run the whole (or part) of the top-k query.
- *Dynamically computed input*: Examples of this category are the output of single-feature similarity queries or through-pulling ranked results from an external source (e.g., a Web site). In this case we need to run *warmup* subqueries on the inputs. A warmup subquery is a single-feature top-S query on each individual input.

In both cases, the ranked samples are ranked "lists" of the top S objects from each individual ranking.

We define a similarity measure between two rankings based on the *footrule distance* [6,7] between those two rankings. The footrule distance between two ranked lists L and R *over the same set of objects* is defined as $F(L,R) = \sum_i |L(i) - R(i)|$, where $L(i)$ and $R(i)$ are the *rank* of object i in L and R, respectively. For two input rankings (possibly on different sets of objects), with a join condition to join objects from the first input with objects from the second input, we generalize the distance metric $F(L,R)$ to $F(L,R) = \sum_{i,j} |L(i) - R(j)|$, where (i,j) is a valid join result that joins object i from L with object j from R.

Using the ranked sample and the definition of the distance metric, $F(L,R)$, we lay out the rank-join order technique in Table 3. The technique is an adaptation of Kruskal's minimum spanning tree algorithm to build the final rank-join evaluation plan.

The algorithm in Table 3 starts by building a graph structure that represents the similarity measure among all inputs. An edge in the graph connects two vertices, each representing an input ranking list, where a join condition exists between these two inputs. The edge is labeled by the value of the distance metric $F(L,R)$, described earlier. The final rank-join

Table 3. The rank-join order algorithm

Rank-join order ($L_1, L_2, \ldots, L_m$, JCs)
input $L_1, \ldots, L_m$: m input ranked lists of size S
  JCs: a set of join conditions pairs of inputs.
output P: a query plan to rank-join the inputs
begin
 Compute $F(L_i, L_j)$ for each pair of inputs L_i and L_j
 Define a graph $G = (V, E)$ as follows:
  Each input ranked list represents one vertex in V
  An edge (L_i, L_j) exists if
  there exists a join condition in JCs between L_i and L_j
  Each edge (L_i, L_j) is labeled with $F(L_i, L_j)$
 Let $T = \{\}$ be a set of graph vertices
 Loop while E is not empty
  Choose the edge $e = (L_i, L_j)$ with the least $F(L_i, L_j)$
  Remove e from E
  if $L_i \in T$ and $L_j \in T$, then ignore e
  else if $L_i \notin T$ and $L_j \notin T$, then:
    Form a subplan P' that rank-joins L_i and L_j
    if $P = NULL$, then $P = P'$
    else let P'' be a subplan that rank-joins P and P';
    and set $P = P''$
    $T = T \cup \{L_i, L_j\}$
  else if $L_i \notin T$, (same for L_j) then:
    let P'' be a subplan that rank-joins P andL_i;
    and set $P = P''$
    $T = T \cup \{L_i\}$
 End Loop
end

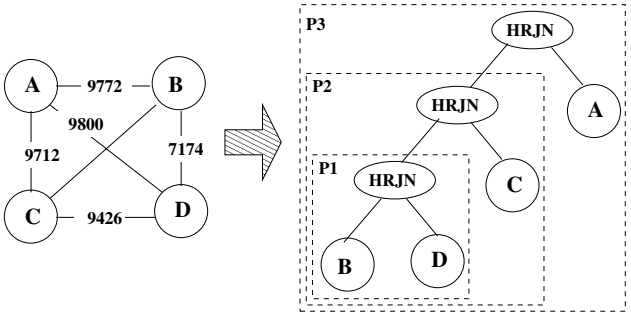


Fig. 10. Example execution of rank-join order algorithm

evaluation plan, say, P, is built bottom-up by choosing the next most similar pair of inputs from the graph. If neither of the two chosen inputs exists in the current plan, a new subplan, P', is formed by providing these two inputs as the inputs to a rank-join operator. P' is joined to the current evaluation plan, P, through building a new rank-join root operator; the inputs of the new root are the current evaluation plan, P, and the newly formed subplan, P'. If only one input from the pair of chosen inputs does not exist in P, this input is joined to the plan using a rank-join operator. Note that, because of the symmetry of rank-join operators, we do not distinguish between the left and right child while building P.

Figure 10 gives a real example of applying the rank-join order algorithm in Table 3 on 4 inputs. The sample size $S = 100$ records from each input. The join condition is an equijoin on the object id from each list. The labels on the graph edges represent the ranking distance as described earlier in this section. First, the algorithm chooses the two inputs B and D since they have the least distance value, 7174. The current evaluation plan $P = P1$ is a rank-join operator that joins B and D. Since the pair of inputs C and D has the next smallest distance value, C is joined to $P1$ to form the evaluation plan $P2$. Finally, the next most similar two inputs are A and C. Hence, A is joined to $P2$ to form the final evaluation plan $P3$. Other edges in the graph are ignored since we consumed all the inputs.

7 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 of using random access is to further reduce the upper bound of the score of unseen join combinations, and hence to be 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 into L's hash table and used to probe the R index. This version can be viewed as a hybrid between a hash join and an 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 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 Fig. 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, and 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 index exists, the algorithm behaves exactly like the original $HRJN$ algorithm.

8 Performance evaluation

In this section, we compare the two rank-join operators, $HRJN$ and $HRJN^*$, introduced in Sect. 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 four UltraSparc-II processors running SunOS 5.6 operating system. The research platform is based on PREDATOR [19], 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 is 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 query **Q** is the left-deep plan, *plan A*, given in Fig. 11. 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.

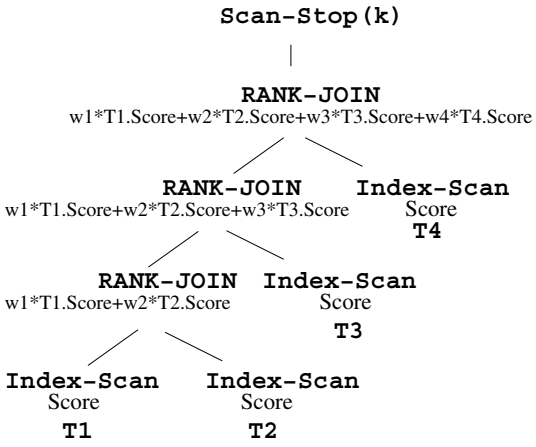


Fig. 11. *Plan A*: A left-deep execution plan for **Q**

8.1 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 Fig. 12. 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.

Plan B does not suffer from the local ranking problem, described in Sect. 5.2, because each operator has almost the same cost for accessing both of its inputs (same number of plan levels). However, with 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.

8.2 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 [16]. 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 Sect. 5.2, is applied in $HRJN$. The $HRJN^*$ operator uses the score-guided

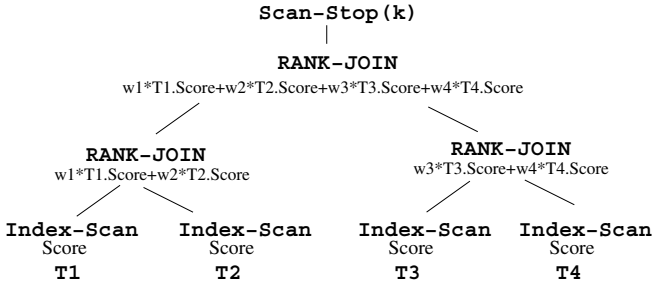


Fig. 12. *Plan B*: A bushy execution plan for **Q**

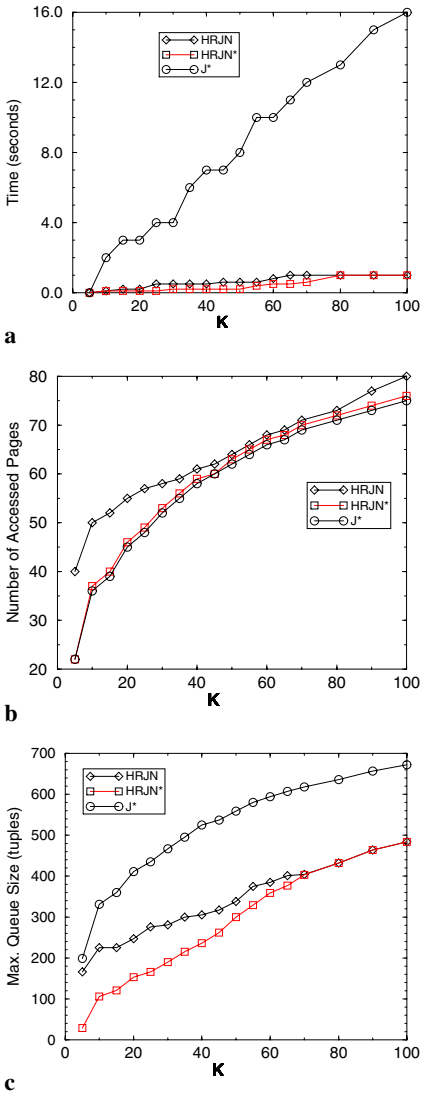


Fig. 13a–c. Comparing $HRJN$, J^*, and $HRJN^*$ for $m = 4$ and selectivity $= 0.2\%$

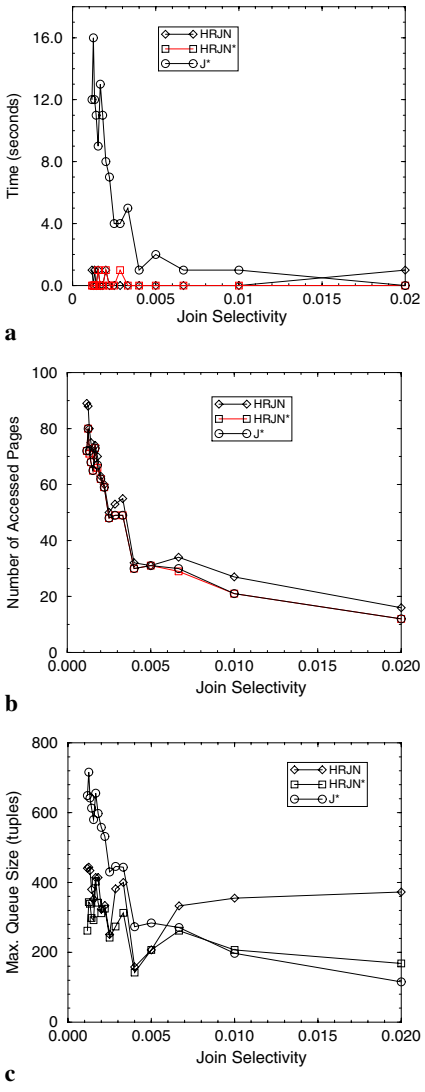
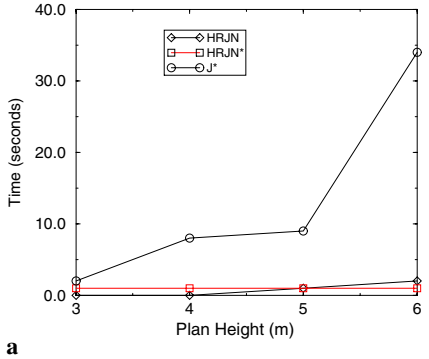


Fig. 14a–c. Effect of selectivity on $HRJN$, J^*, and $HRJN^*$ for $m = 4$ and $K = 50$

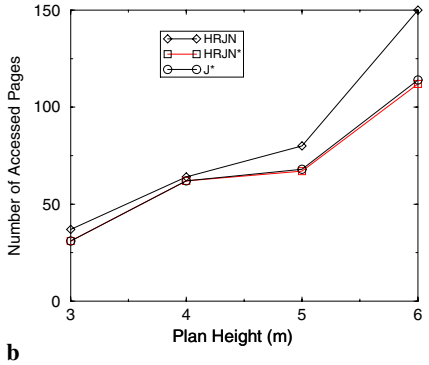
strategy, proposed in Sect. 5.3, to determine the rate at which it retrieves tuples from both inputs. The J^* operator is an implementation of the J^* algorithm. We do not compare with the naïve approach of joining the inputs then sorting since all the rank-join algorithms give a better performance by orders of magnitude. We choose four performance metrics: total time to retrieve k ranked results, total number of accessed disk pages, maximum queue size, and total occupied space. In the following experiments, we use *plan A* as the execution plan for **Q**. Using *plan B* gives 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 13a 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 due to the fact that 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 a symmetric hash join to produce valid join combinations, more join combinations are ranked at each step. Figure 13b 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 causes a significant decrease in the threshold value and hence less I/O. Figure 13c 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^* would have had 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.

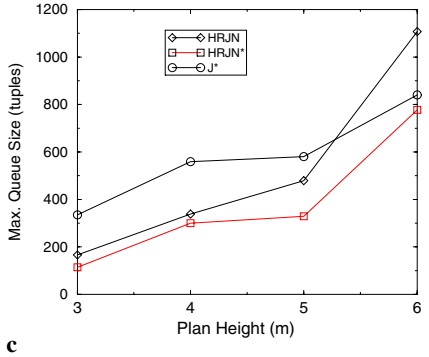
Changing the join selectivity. In this experiment, we fix the value of k at 50 and vary the join selectivity from 0.12 to 2%.



a



b



c

Fig. 15a–c. Effect of pipelining on $HRJN$, J^*, and $HRJN^*$ for selectivity $= 0.2\%$ and $K = 50$

Figure 14a compares the total time to report 50 ranked results, while Figs. 14b and 14c 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 15a 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 $HRJN$ (Fig. 15b) because of the score-guided strategy they use. $HRJN^*$ is the most scalable in terms of the space overhead, as shown in Fig. 15c.

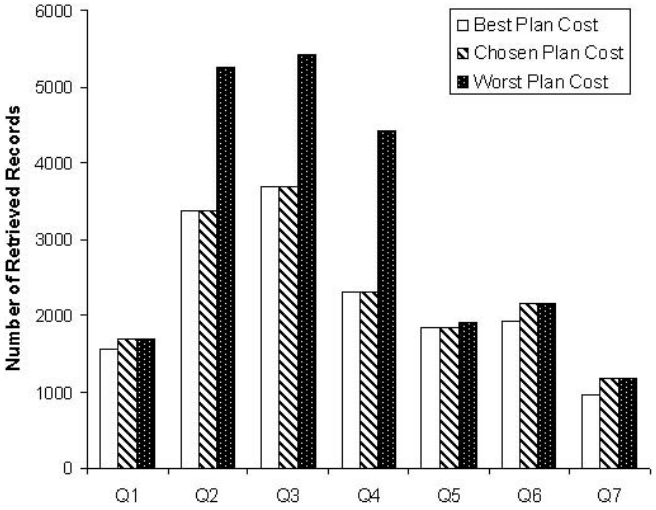


Fig. 16. Evaluation of rank-join order technique

8.3 Evaluating the rank-join order technique

In Sect. 6, we introduced a technique for choosing an order of the rank-join operators based on sampling. In this experiment, we evaluate the efficiency of the proposed heuristic. We experiment with several top-k join queries and choose 7 representative queries that join the rankings according to three visual features. The set of visual features in each query is different. For each query, we compute the cost of all possible evaluation plans by actually executing these plans. In each query, we compare the cost of best plan, the cost of the chosen plan according to our heuristic, and the cost of the worst plan. The cost of a plan is calculated as the total number of retrieved objects from the input rankings.

Figure 16 shows that in most cases, the heuristic is able to avoid "bad" orders and hence expensive execution plans. The figure shows that for the cases where there is a significant difference between the cost of the best and the worst order, our technique is able to choose an order that is (or as close as possible to) the optimal rank-join order, saving up to 48% of the cost. For the cases where there is no significant cost difference between the best and the worst plan (less that 20%), the technique tends to choose a random join order that can range from the best to the worst rank-join order.

9 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 the join. We analyze the I/O performance of the proposed rank-join algorithm and prove its optimality in terms of the number of accessed input tuples. 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 and apply the new strategy to the original $HRJN$ algorithm and call the new operator $HRJN^*$. We study the effect of rank-join order on the performance of rank-join query evaluation pipeline and introduce an efficient rank-join order heuristic to help choose a near-optimal join order. We address the possibility of exploiting available indexes on the join columns, propose a general rank-join algorithm that utilizes these indexes for faster termination of the ranking process, and experimentally evaluate the proposed join operators and compare their performance with a recent algorithm to join ranked inputs. Finally, 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 join selectivity values.

Acknowledgements. We acknowledge the support of the National Science Foundation under Grants Numbers IIS-0093116, EIA-9972883, and IIS-0209120. We would like to thank Ronald Fagin for helping us with the instance optimality analysis of the rank-join algorithm.

References

1. Bruno N, Chaudhuri S, Gravano L (2002) Top-k selection queries over relational databases: mapping strategies and performance evaluation. ACM Trans Database Sys (TODS) 27(2):369–380
2. Bruno N, Gravano L, Marian A (2002) Evaluating top-k queries over web-accessible databases. In: Proceedings of the IEEE 18th international conference on data engineering (ICDE), San Jose, CA, pp 153–187
3. Carey MJ, Kossmann D (1997) On saying "Enough already!" in SQL. In: Proceedings of the ACM SIGMOD international conference on management of data, Tucson, AZ, pp 219–230
4. Carey MJ, Kossmann D (1998) Reducing the braking distance of an SQL query engine. In: Proceedings of the 24th international conference on very large databases (VLDB), New York, August 1998, pp 158–169. Morgan Kaufmann, San Francisco
5. Chen-Chuan Chang K, won Hwang S (2002) Minimal probing: supporting expensive predicates for top-k queries. In: Proceedings of the ACM SIGMOD international conference on management of data, pp 346–357
6. Diaconis P (1988) Group representation in probability and statistics. IMS Lecture Series 11, IMS
7. Diaconis P, Graham R (1977) Spearman's footrule as a measure of disarray. J R Stat Soc 39(2):262–368
8. Dwork C, Ravi Kumar S, Naor M, Sivakumar D (2001) Rank aggregation methods for the web. In: Proceedings of the 10th international conference on the World Wide Web, Hong Kong, pp 613–622
9. Fagin R (1999) Combining fuzzy information from multiple systems. J Comput Sys Sci 58(1):216–226
10. Fagin R, Lotem A, Naor M (2001) Optimal aggregation algorithms for middleware. In: Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems (PODS), Santa Barbara, CA, pp 102–113
11. Güntzer U, Balke W-T, Kießling W (2000) Optimizing multi-feature queries for image databases. In: Proceedings of the 26th international conference on very large databases (VLDB), Cairo, Egypt. Morgan Kaufmann, San Francisco, pp 419–428
12. Güntzer U, Balke W-T, Kießling W (2001) Towards efficient multi-feature queries in heterogeneous environments. In: Proceedings of the IEEE international symposium on information technology (ITCC), Las Vegas, pp 622–628
13. Haas PJ, Hellerstein JM (1999) Ripple joins for online aggregation. In: Proceedings of the ACM SIGMOD international conference on management of data, Philadelphia, pp 287–298
14. Hong W, Stonebraker M (1993) Optimization of parallel query execution plans in XPRS. Distrib Parallel Databases 1(1):9–32
15. Ilyas IF, Aref WG, Elmagarmid AK (2002) Joining ranked inputs in practice. In: Proceedings of the 28th international conference on very large databases (VLDB), Hong Kong. Morgan Kaufmann, San Francisco, pp 950–961
16. Natsev A, Chang Y-C, Smith JR, Li C-S, Vitter JS (2001) Supporting incremental join queries on ranked inputs. In: Proceedings of the 27th international conference on very large databases (VLDB), Rome, pp 281–290. Morgan Kaufmann, San Francisco
17. Nepal S, Ramakrishna MV (1999) Query processing issues in image (multimedia) databases. In: Proceedings of the IEEE 15th international conference on data engineering (ICDE), Sydney, Australia, pp 22–29
18. Selinger PG, Astrahan MM, Chamberlin DD, Lorie Ra, Price TG (1979) Access path election in a relational database management system. In: Proceedings of the ACM SIGMOD international conference on management of data, Boston, pp 23–34
19. Seshadri P, Paskin M (1997) Predator: An or-dbms with enhanced data types. In: Proceedings of the ACM SIGMOD international conference on management of data, Tucson, AZ, pp 568–571
20. Urhan T, Franklin MJ (2000) XJoin: A reactively scheduled pipelined join operator. IEEE Data Eng Bull 23(2):27–33
21. Urhan T, Franklin MJ (2001) Dynamic pipeline scheduling for improving interactive query performance. In: Proceedings of the 27th international conference on very large databases (VLDB), Rome. Morgan Kaufmann, San Francisco, pp 501–510
22. Wilschut AN, Apers PMG (1991) Dataflow query execution in a parallel main-memory environment. Distrib Parallel Databases 1(1):68–77