Ranked Join Indices

Panayiotis Tsaparas
University of Toronto
tsap@cs.toronto.edu

Themistoklis Palpanas
University of Toronto
themis@cs.toronto.edu

Yannis Kotidis
AT&T Labs-Research
kotidis@research.att.com

Nick Koudas
AT&T Labs-Research
koudas@research.att.com

Divesh Srivastava
AT&T Labs-Research
divesh@research.att.com

Abstract

A plethora of data sources contain data entities that could be ordered according to a variety of attributes associated with the entities. Such orderings result effectively in a ranking *of the entities according to the values in the attribute domain. Commonly, users correlate such sources for query processing purposes through join operations. In query processing, it is desirable to incorporate user preferences towards specific attributes or their values. A way to incorporate such preferences, is by utilizing scoring functions that combine user preferences and attribute values and return a numerical score for each tuple in the join result. Then, a target query, which we refer to as* top-k *join query, seeks to identify the k tuples in the join result with the highest scores.*

In this paper, we propose a novel technique, which we refer to as ranked join index, *to efficiently answer* top-k *join queries for arbitrary, user specified, preferences and a large class of scoring functions. Our rank join index requires small space (compared to the entire join result) and provides guarantees for its performance. Moreover, our proposal provides a graceful tradeoff between its space requirements and worst case search performance. We supplement our analytical results, with a thorough experimental evaluation using a variety of real and synthetic data sets, demonstrating that in comparison to other viable approaches, our technique offers significant performance benefits.*

1 Introduction

A plethora of data sources contain data entities that could be ordered according to a variety of attributes associated with the entities. Such orderings result effectively in a *ranking* of the entities according to the values in the attribute domain. Such values could reflect various quantities of interest for the entities, such as physical characteristics, quality, reliability or credibility to name a few. As examples, consider a database of houses ordered (ranked) by price or number of rooms; the list of all airports in the country ranked by average flight delay; parts in a supplier-part database ranked by their availability, or suppliers in the same database ranked by their credibility or quality of service (derived by recording user experience with suppliers over time). We refer to such attributes as *rank* attributes. The domain of rank attributes depends on their semantics. For example, the domain could either consist of categorical values (e.g., service can be excellent, fair or poor) or numerical (e.g., an interval of continuous values from R^+).

The existence of rank attributes along with data entities leads to enhanced functionality and query processing capabilities. Indeed a variety of recent works have addressed several aspects of the problem of enhancing query processing taking into account user preferences towards the values of rank attributes [12, 3, 4]. Of particular importance is query answering with the goal of optimizing functions that capture user preferences towards rank attribute values. For example in a database of rental houses, data entities (i.e., houses) can be ranked according to a variety of attributes such as, distance from a specific location, number of rooms, rental price and building age. Users specify their preferences towards specific attributes. Commonly, preferences are expressed in the form of numerical weights, assigned to rank attributes by the user. Query processors incorporate functions that weight attribute values by user preference deriving scores for individual entities. Specific techniques have been developed to carry out query processing with the goal of identifying results that optimize such functions. A typical instance is a query that seeks to quickly identify k data entities that yield best scores among all entities in the database. At an abstract level such queries can be considered as generalized forms of selection queries.

Of equal importance is the ability to support related query functionality on join queries. Consider Figure 1. It consists of two tables, `parts` and `suppliers`. Table `parts` contains three attributes *availability*, *name* and *supplier id*. Similarly table `suppliers` consists of two attributes *supplier id* and *quality*. Assume for purposes of exposition that all parts correspond to the same piece of a mechanical device possi-

PARTS

availability	name	supplier id
5	PO5	1
2	PO5	2
9	PO5	3

SUPPLIERS

supplier id	quality
1	10
2	3
3	8

Figure 1. **Tables and Rank Attributes**

bly of different brand. Rank attributes *availability* and *quality* determine the availability (e.g., current quantity in stock for this part) and the quality of the supplier (e.g., acquired by user experience reports on this supplier) respectively, having as domain a subset of R^+ as shown in Figure 1. A user interested in purchasing parts from suppliers will have to correlate, through a join on *supplier id*, the two tables. Rank attributes, could provide great flexibility in query specification in such cases. For example, a user looking for a part might be more interested in the availability of the part as opposed to supplier quality. In a similar fashion supplier quality might be of greater importance to another user, than part availability. It is imperative to capture user interest or preference towards rank attributes spanning multiple tables and efficiently support such queries, involving user preferences and table join results. User preference towards rank attributes is captured by allowing users to specify numerical values (weights), for any rank attribute. The larger the weights the greater the preference of the user towards the rank attributes. Assuming the existence of *scoring functions* that combine user preferences and rank attribute values, returning a numerical *score*, our target queries, which we refer to as *top-k* join queries, seek to identify the k tuples in the join result of `parts` and `suppliers` with higher scores.

In this paper, we propose a novel technique, which we refer to as *ranked join index*, to efficiently answer *top-k* join queries for arbitrary, user specified, preferences and a large class of scoring functions, namely *monotone linear scoring functions*. Our ranked join index requires small space (compared to the entire join result) and provides guarantees for its performance. It can exist separately from the joined relations and utilized in a variety of query processing scenarios, since like traditional join indices [16] it is compatible with relational operations like selection and union.

In particular, we make the following contributions:

- Our *ranked join index* design provides guaranteed *worst case* search performance for *top-k* join queries for a large class of scoring functions parameterized by arbitrary user preferences.
- We show that for a large class of scoring functions of interest, the space required by our ranked join index is much smaller than that required for materializing the entire join result. For a maximum value of K (provided at construction time) denoting an upper bound on the number of results requested by *top-k* join queries, we show that the worst case space requirements of our index are small and independent of user preferences. We propose an algorithm to identify the join result tuples that are necessary to include and maintain in our ranked join index.
- We provide an efficient algorithm to construct the ranked join index and derive its properties with respect to worst case search performance for any *top-k* join query, $k \leq K$ involving arbitrary user preferences. We show that its performance is logarithmic in the size of the index and the size of the query result.
- We demonstrate that our design provides a graceful tradeoff between space and worst case search performance and we quantify this tradeoff.
- We propose and implement alternate solutions for answering *top-k* join queries based on R-trees.
- We experimentally demonstrate the performance benefits our approach offers when compared to an approach based on R-trees.

This paper is organized as follows: Section 2 reviews related work. Section 3 formally defines the class of scoring functions and the problem we consider in this paper. In section 4 we present an algorithm that identifies the tuples in the join result that should be further indexed and processed with our ranked join index. Sections 5 and 6 present our design for the ranked join index, proposing an algorithm to construct it and analytically derive its worst case search properties. Section 8 presents the results of a thorough experimental evaluation using real and synthetic datasets, presenting the performance advantages of our approach when compared with other applicable approaches. Finally, section 9 concludes the paper and points to problems of interest for further study.

2 Related Work

Agrawal and Wimmers [1] proposed a framework for preference based query processing. Various works considered realizations of a specific instance of this framework, namely *top-k* selection queries, that is, quickly identifying k tuples that optimize scores assigned by monotone linear scoring functions on a variety of ranked attributes and user specified preferences [10, 6, 12, 3, 5]. Most of these techniques for answering *top-k* selection queries [10, 12, 6, 3] are not based on indexing. Instead, they are geared towards optimizing the number of tuples examined in order to identify the answer under various cost models of interest. Such optimizations include, minimization of tuples read sequentially from the input [12, 10, 6] or minimization of random disk access [3, 4]. Chang et al. [5] propose an indexing technique for answering *top-k* selection queries. This technique does not provide guarantees for its performance and in the worst case, the entire data set has to be examined in order to identify the correct answer to a *top-k* selection query.

A significant volume of work in multimedia databases addresses issues of incorporating user preferences into query processing [9, 8, 7]. The optimization objectives and the nature of solutions are not directly related to our framework as these works do not address indexing.

Natsev et al. [14] proposed techniques to answer *top-k* queries over the join of two relations. They assume no preprocessing and compute the join of the relations from scratch for each join condition and user supplied preference values. The techniques provide no performance guarantees for general data distributions and arbitrary user preferences. However, since the entire join is computed from scratch, they offer the flexibility of incorporating arbitrary join conditions between the two relations. A recent study presents an efficient implementation of a pipelined operator for ranked joins [13]. Our work presents the *first* solution providing performance guarantees for *top-k* join queries over two relations, when preprocessing to construct a ranked join index for a specific join condition is permitted.

We note that our work is also applicable in the case of a single relation in the spirit of the works in [10, 12, 5]. In this case, our work extends these approaches in the sense that it provides the *first* solution to the *top-k* selection problem with monotone linear functions, having *guaranteed* worst case search performance for the case of two ranked attributes and arbitrary preference vectors.

3 Problem Definition

Let R, S be two relations, with attributes $A_1, \ldots, A_n$ and $B_1 \ldots B_m$ respectively. Without loss of generality, assume that A_1, B_1 are rank attributes with domain a subset of R^+ and θ an arbitrary join condition defined between (sub)sets of $A_2, \ldots, A_n, B_2, \ldots B_m$ ($R \bowtie_\theta S$). For a tuple $t \in R \bowtie_\theta S$, $A_i(t)$ (similarly $B_i(t)$), corresponds to the value of attribute A_i of tuple t. Let $f : R^+ \times R^+ \rightarrow R^+$ be a scoring function that takes as input the pair of rank attribute values $(s_1, s_2) = (A_1(t), B_1(t))$ of tuple $t \in R \bowtie_\theta S$, and produces a score value $f(s_1, s_2)$ for the tuple t.

Definition 1 (Monotone Functions) *A function $f : R^+ \times R^+ \rightarrow R^+$ is* monotone *if the following holds: if $x_1 \leq x_2$, and $y_1 \leq y_2$, then $f(x_1, y_1) \leq f(x_2, y_2)$.*

Let $e = (p_1, p_2)$ denote the user defined preferences towards rank attributes A_1, B_1. We define a *linear* scoring function $f_e : R^+ \times R^+ \rightarrow R^+$ as a scoring function that maps a pair of score values (s_1, s_2) to the value $f_e(s_1, s_2) = p_1 s_1 + p_2 s_2$. We assume that user preferences are positive (belonging to R^+); this is an intuitive assumption as it provides monotone semantics to preference values (the greater the value the larger the preference towards that attribute value). In such a case, the linear function f_e is monotone as well. We use $\mathcal{L}$ to denote the class of monotone linear functions. Note that the pair of user defined preferences e uniquely determines a function $f \in \mathcal{L}$.

Definition 2 (*top-k* join query) *Given relations R, S, a join condition θ and a scoring function $f_e \in \mathcal{L}$, a* top-k *query returns a collection $T_k(e) \subseteq R \bowtie_\theta S$ of k tuples ordered by $f_e(A_1(t), B_1(t))$, such that for all $t \in R \bowtie_\theta S$, $t \notin T_k(e) \Rightarrow f_e(A_1(t), B_1(t)) \leq f_e(A_1(t_i), B_1(t_i))$, for all $t_i \in T_e(k), 1 \leq i \leq k$.*

Thus, a *top-k* join query returns as a result k tuples from the join of two relations with the highest score, for a user specified scoring function f_e, among all tuples in the join result. We are now ready to formally define the main problem we address in this paper.

Problem 1 (Ranked Join Index) *Given relations R, S, a join condition θ, a class of scoring functions $\mathcal{L}$ and an upper bound K for the maximum requested result size of any* top-k *join query, preprocess $R \bowtie_\theta S$ and construct an index, providing answers with guaranteed performance on any* top-k *join query, $k \leq K$, issued using any scoring function $f \in \mathcal{L}$.*

We will demonstrate that our solution offers search performance logarithmic to the size of the index. We present our solutions in the following steps. First we show that only a subset of $R \bowtie_\theta S$ is necessary to be represented in our join index. This subset is the same for all scoring functions in $\mathcal{L}$. Then we present the construction of the join index and show its properties. All the proofs of lemmas and theorems are omitted due to space constraints. They are available in the full version of this paper.

4 Pruning the Join Result

If the relations R, S to be joined consist of $O(n)$ tuples, the size of the join relation $R \bowtie_\theta S$ may be as large as $O(n^2)$. We demonstrate that most of the tuples of the join relation, $R \bowtie_\theta S$, are not necessary for answering *top-k* join queries. In particular, we will show that for a fixed value K and for the entire class of linear functions $\mathcal{L}$, in the worst case, a number of tuples much smaller than $O(n^2)$ is sufficient to provide the answer to any *top-k* join query, $k \leq K$. We present algorithms that successively prune the relation $R \bowtie_\theta S$ to produce the number of required tuples.

We first note that we do not need to generate the complete join result $R \bowtie_\theta S$. Let C denote the subset of $R \bowtie_\theta S$ necessary to generate, in the worst case, in accordance to Problem 1. We make a simple observation that limits the size of C substantially. Note that although each tuple t of R could join in the worst case with $O(n)$ tuples of S, for a fixed value of K, we only join t with at most K tuples in S, the ones that have the highest rank values. Therefore, among the possible $O(n)$ tuples in the join that are produced for each tuple $t \in R$, only the K tuples with the highest rank values are required. Due to the monotonicity property of functions in $\mathcal{L}$ these K tuples will have the highest scores for any $f \in \mathcal{L}$. This is formalized by the following lemma.

Lemma 1 *For relations of size $O(n)$ and a value K in accordance to Problem 1, the worst case size of C is $O(nK)$.*

Note that this worst case size is query independent, i.e., using the same set of tuples C of worst case size $O(nK)$ one can answer any *top-k* join query, $k \leq K$, for any $f \in \mathcal{L}$. In a preprocessing step, C can be determined by joining R and S and selecting for each tuple $t \in R$ the K (worst case) tuples contributed by t to the join result, that have the highest rank values in S. Such a preprocessing step can be carried out in a fully declarative way using SQL. We seek ways to reduce the size of C further.

Definition 3 *Let t and t' denote two tuples of $R \bowtie_\theta S$. Let (s_1, s_2) and (s'_1, s'_2) denote the pairs of rank values associated with each tuple. We say that tuple t'* dominates *tuple t, if $s_1 \leq s'_1$, and $s_2 \leq s'_2$.*

The domination property provides a basic means to prune C further. The intuition is as follows. Lemma 1 prunes the join result, by restricting the number of the tuples contributed to the join by a single tuple of a relation. The domination property of Definition 3 enables pruning by examining the tuples contributed to the join by multiple tuples of a relation. This intuition is formalized by the following lemma.

Lemma 2 *For a value of K in accordance to Problem 1, if some tuple $t \in C$ is dominated by at least K other tuples, then t can safely be excluded from C as it cannot be in the answer set of any* top-k *join query, $k \leq K$.*

A proof for this lemma follows from the monotonicity properties of the scoring functions. It is evident from Lemma 2 that a viable strategy to reduce the size of C is to identify all tuples in C dominated by at least K tuples. We formalize this with the following definition.

Definition 4 *Given a set C, the* dominating set *$\mathcal{D}_K$ is the* minimal *subset of C with the following property: for every tuple $t \notin \mathcal{D}_K$ with rank values (s_1, s_2), there are at least K tuples $t_i \in \mathcal{D}_K$, that dominate tuple t.*

We present the algorithm for computing the dominating set, $\mathcal{D}_K$ for any value of K in Figure 2. Every tuple t_i in C is associated with a pair of rank values (s^i_1, s^i_2). The algorithm maintains a priority queue Q (supporting insertions/deletions in logarithmic time) storing the K largest s^i_1 rank values encountered so far. It first sorts the tuples in the join result in non-increasing order with respect to the s^i_2 rank values. It then considers the tuples one at a time in that order. For every tuple t_i, if its s^i_1 rank value is less than the minimum rank value present in Q we discard it. Otherwise the tuple is included in the dominating set, and the priority queue Q is updated. It can be shown that this algorithm correctly computes the dominating set $\mathcal{D}_K$, for a *top-k* join query, for $k \leq K$.

Algorithm *DominatingSet* (shown in Figure 2), requires time $O(|C| \log |C|)$ for sorting and computes the dominating set $\mathcal{D}_K$ in $O(|C| \log K)$ time. The number of tuples pruned by algorithm *DominatingSet* depends on the distribution of the rank value pairs in the join result. In practice we expect the size of $\mathcal{D}_K$ to be much smaller than $O(nK)$. In the worst case, however, no tuple is dominated by K other tuples and as a result algorithm *DominatingSet* does not achieve any pruning. Consider the following example.

```
**DominatingSet**($C, K$)
Initialize priority queue $Q$ and $\mathcal{D}_K = \emptyset$.
Sort the join result in non-increasing order of the $s_2$ rank values.
For the $i$th tuple $t_i$ with rank values $(s^i_1, s^i_2)$
    if ($|Q| < K$)
        include $t_i$ in $\mathcal{D}_K$
        insert $s^i_1$ in $Q$
    else
        if $s^i_1 \leq \min\{Q\}$ discard $t_i$
    else
        include $t_i$ in $\mathcal{D}_K$
        insert $s^i_1$ in $Q$
        if $|Q| > k$ delete the minimum element of $Q$
Output $\mathcal{D}_K$
```

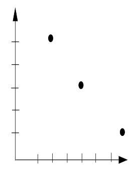
Figure 2. **The *DominatingSet* Algorithm**

PARTS

availability	name	supplier id
5	PO5	1
3	PO5	2
1	PO5	3

SUPPLIER

supplier id	quality
1	2
2	4
3	6
4	1



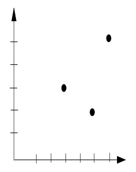
(a)

PARTS

availability	name	supplier id
5	PO5	1
3	PO5	2
2	PO5	3

SUPPLIER

supplier id	quality
1	5
2	3
3	4
4	1



(b)

Figure 3. **Examples of set $\mathcal{D}_K$ for different join results**

Example 1 *Figure 3 presents two pairs of relations with different rank attribute values. For both pairs of relations, the size of the join result is the same (equal to 3). For the tuples of each join result in Figure 3 we draw a geometric analogy and represent the tuple by the rank attribute pair* (quality,availability) *as a point in two dimensional space. For the rank attribute value distributions in Figure 3(a) the set $\mathcal{D}_1$ has size 3 (worst case) since no tuple is dominated by any other tuple. Thus, in this case algorithm* DominatingSet *will produce $\mathcal{D}_1$ having a size equal to the theoretically predicted worst case. In contrast, in Figure 3(b), algorithm* DominatingSet *will produce a set $\mathcal{D}_1$ with size 1 (containing the tuple whose rank attribute pair dominates the other two).*

In Section 8 we experimentally evaluate the amount of pruning achieved by the algorithm for several data distributions.

The following lemma establishes the relationship among the sets $\mathcal{D}_k$ associated with each *top-k* join query possible with $k \leq K$.

Lemma 3 *Consider two* top-k *join queries requesting k_1, k_2 results and $k_1 \leq k_2 \leq K$. For the dominating sets $\mathcal{D}_{k_1}$, $\mathcal{D}_{k_2}, \mathcal{D}_K$, the following property holds: $\mathcal{D}_{k_1} \subseteq \mathcal{D}_{k_2} \subseteq \mathcal{D}_K$.*

Lemma 3 shows that it is sufficient to identify and materialize only set $\mathcal{D}_K$, since the answers to *any top-k* join query, $k \leq K$ are contained in this set. Also the lemma holds for any scoring function $f \in \mathcal{L}$. Executing algorithm *DominatingSet* on C, using K as provided in Problem 1 identifies all the tuples necessary to answer any *top-k* join query with $k \leq K$. In the next section we will consider the issues that arise when one is interested to index set $\mathcal{D}_K$ in order to provide answers to *top-k* join queries with guaranteed worst case access time.

5 Constructing the Ranked Join Index

We now present an algorithm to preprocess set $\mathcal{D}_K$ and construct an index structure, called *RJI*, providing answers to *top-k* join queries in an efficient way.

Every function $f \in \mathcal{L}$ is completely defined by a pair of preference values (p_1, p_2). The value of f on a tuple $t \in \mathcal{D}_K$ with rank values (s_1, s_2) is $p_1 s_1 + p_2 s_2$. We will present our construction by representing members of $\mathcal{L}$ and rank value pairs for each $t \in \mathcal{D}_K$ as vectors in a 2-dimensional space. Since every $f_e \in \mathcal{L}$ is completely defined by the pair $e = (p_1, p_2)$ we can think of every function f to be represented by the vector $e = \langle(0,0)(p_1,p_2)\rangle$ on the plane. Similarly, rank value pairs, can be represented as a vector $s = \langle(0,0)(s_1,s_2)\rangle$. In light of this geometric representation the value of a function f on a tuple $t \in \mathcal{D}_K$ with rank values (s_1, s_2) is the inner product of the vectors e, and s. The intuition behind representing members of $\mathcal{L}$ as vectors, is as follows. Assume that $||e|| = 1$, that is, e is a unit vector. Then, the value of $f_{(p_1,p_2)}(s_1, s_2)$ is the length of the projection of vector s on vector e, as shown in Figure 4(a). The assumption that e is a unit vector is solely for simplifying our presentation; it is not required for the correctness of our approach. The result of any *top-k* join query $T_k(e)$ is the same, independent of the magnitude of e. Let $u = \alpha e$ be some vector in the direction of e with length α. $T_k(e)$ is exactly the same as $T_k(u)$ since the lengths of the projected vectors change only by a scaling factor, and thus, their relative order is not affected.

The set of tuples $\mathcal{D}_K$ can be represented as points in two dimensional space using the rank values of each tuple as shown in Figure 4(b). Given a unit vector e, we define the *angle* $a(e)$ of the vector to be the angle of e with the axis representing (without loss of generality) the s_1 rank values as shown in Figure 4(b). For a set of ℓ tuples $\{t_1, t_2, ..., t_\ell\}$, we define $Ord_e(\{t_1, t_2, ..., t_\ell\})$ to be the ordering of the tuples $\{t_1, t_2, ..., t_\ell\}$ when the rank value pairs associated with each tuple are projected on the vector e, and sorted by non-increasing order of their projection lengths. We use $\overline{Ord_e}(\{t_1, t_2, ..., t_\ell\})$ to denote the *reverse* of that ordering. $T_k(e)$ contains the top k tuples in the ordering $Ord_e(\{t_1, t_2, ..., t_\ell\})$. Figure 4(b) presents such an ordering.

Let the vector e sweep the plane defined by the domains of rank attributes ($R^+ \times R^+$). In particular assume that the sweep starts from the s_1-axis moving towards the s_2-axis (i.e., counter-clockwise). Thus e ranges from $e = \langle(0,0)(1,0)\rangle$ to $e = \langle(0,0),(0,1)\rangle$. We are interested to examine how the ordering $Ord_e(\mathcal{D}_K)$ varies as e sweeps the plane, thus considering every possible scoring function.

Let us first consider two tuples and examine their relative ordering. Let $s^1 = (s^1_1, s^1_2)$, and $s^2 = (s^2_1, s^2_2)$ be the rank value pairs for two tuples $t_1, t_2 \in \mathcal{D}_K$. Since rank value pairs are represented as vectors, let $\langle s^1 s^2\rangle = s^2 - s^1$ denote the vector defined by the difference of s^2 and s^1, and let b denote the angle of the vector $\langle s^1 s^2\rangle$ with the s_1-axis. Figure 4(c) presents an example. The ordering of t_1 and t_2 as e sweeps the plane is governed by the following lemma.

Lemma 4 *Let $s^1 = (s^1_1, s^1_2)$ and $s^2 = (s^2_1, s^2_2)$ be two vectors formed by the rank value pairs corresponding to two tuples $t_1, t_2 \in \mathcal{D}_K$. Depending on the angle b that vector $\langle s^1 s^2\rangle$ forms with the s_1-axis, as e sweeps the plane one of the following holds:*

(a) if $b \in [0, \frac{\pi}{2}]$, $Ord_e(\{t_1, t_2\})$ is the same for all e.

(b) if $b \in [-\frac{\pi}{2}, 0] \cup [\frac{\pi}{2}, \pi]$, let e_s be the vector perpendicular to $\langle s^1 s^2\rangle$. We have:

  (i) $f_{e_s}(s^1_1, s^1_2) = f_{e_s}(s^2_1, s^2_2)$,

  (ii) $Ord_{e_1}(\{t_1, t_2\}) = Ord_{e_2}(\{t_1, t_2\})$, for all vectors e_1, e_2 with $a(e_1), a(e_2) > a(e_s)$, or $a(e_1), a(e_2) < a(e_s)$,

  (iii) $Ord_{e_1}(\{t_1, t_2\}) = \overline{Ord_{e_2}}(\{t_1, t_2\})$, for all e_1, e_2, such that $a(e_1) < a(e_s) < a(e_2)$. Moreover, as a vector e sweeps the positive quadrant, tuples t_1, t_2 are adjacent in the ordering $Ord_e(\mathcal{D}_K)$ immediately before e crosses vector e_s, and remain adjacent in $Ord_e(\mathcal{D}_K)$ immediately after e crosses e_s.

Lemma 4 indicates that as e sweeps the plane, the ordering of tuples t_1 and t_2 changes only when e crosses e_s, which is defined as the vector perpendicular to $\langle s^1 s^2\rangle$. If the vector $\langle s^1 s^2\rangle$ has positive slope, then the ordering of the tuples t_1, t_2 remains the same for all e. We call the vector e_s the *separating vector* of tuples t_1 and t_2, and $a(e_s)$ the *separating point*. Figure 5 presents an example of this behavior. We note that more than two tuples may share the same separating vector. For example, if t_1, t_2, t_3 are three tuples such that their corresponding rank value pairs are co-linear, they all share the same separating vector. We generalize Lemma 4 as follows.

Lemma 5 *If $t_1, t_2, ..., t_\ell$ are ℓ tuples with co-linear rank value pairs that share the same separating vector e_s, then $Ord_{e_1}(\{t_1, t_2, ..., t_\ell\}) = \overline{Ord_{e_2}}(\{t_1, t_2, ..., t_\ell\})$, for all $a(e_1), a(e_2)$ such that $a(e_1) < a(e_s) < a(e_2)$.*

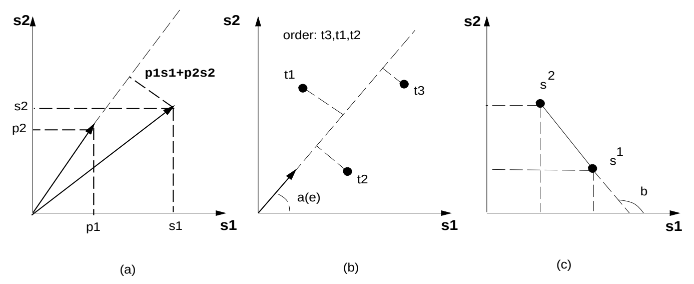


Figure 4. **Vector representation of scoring functions and rank attribute values**

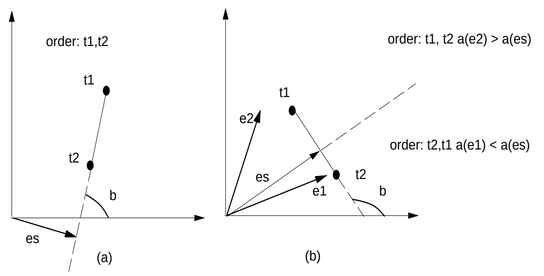


Figure 5. **The two cases of Lemma 4**

Lemma 5 demonstrates that each separating vector corresponds to the reversal of two or more adjacent points. Lemmas 4 and 5 demonstrate properties of the relative ordering of the elements of $\mathcal{D}_K$ for all possible members of $\mathcal{L}$. We will now utilize these properties to efficiently index $\mathcal{D}_K$.

6 Algorithm *ConstructRJI*

We present algorithm *ConstructRJI* which preprocesses $\mathcal{D}_K$ and constructs an index on its elements. We let a vector e sweep the plane, and we keep track of the composition of $T_K(e)$. Every time vector e crosses a separating vector, $Ord_e(\mathcal{D}_K)$ changes by swapping two (or more if they are colinear) *adjacent* tuples as shown by Lemmas 4 and 5. A key observation is that this swap is of interest for indexing purposes only if it causes the composition of $T_K(e)$ to change.

```
ConstructRJI($\mathcal{D}_K$)
For all $(t_i, t_j)$, $t_i, t_j \in \mathcal{D}_K$
    $V \leftarrow$ Compute separating vectors $e_{s_{ij}}$ and separating points $a(e_{s_{ij}})$
Sort $V$ in non-decreasing order of $a(e_{s_{ij}})$
Form $R$ consisting of top-$K$ tuples in $\mathcal{D}_K$ with respect to $f_{(1,0)}$
Set $\ell$ = 0; $R_\ell = R$;
For each element $(t_i, t_j)$ of $V$
    if $t_i, t_j \in R$ or $t_i, t_j \notin R$
        No change in $R$'s composition by $e_{s_{ij}}$; discard $e_{s_{ij}}$
    if $t_i \in R$ and $t_j \notin R$
        Materialize $a(e_{s_{ij}})$,$R$; replace $t_i$ with $t_j$ in $R$
    if $t_i \notin R$ and $t_j \in R$
        Materialize $a(e_{s_{ij}})$,$R$; replace $t_j$ with $t_i$ in $R$
When $V$ is exhausted, materialize $R$
```

Figure 6. **Algorithm *ConstructRJI***

The algorithm is shown in Figure 6. Assuming that $\mathcal{D}_K$ contains tuples of the form (tid_i, s_1^i, s_2^i), where tid_i a tuple identifier, and s_1^i, s_2^i the associated rank values, the algorithm starts by first computing the set V of all separating vectors. This involves considering each pair of tuples in $\mathcal{D}_K$ and computing their separating vector and the associated separating point. Let $e_{s_{ij}}$ ($a(e_{s_{ij}})$) be the separating vector (separating point) for each pair of tuples $t_i, t_j, 1 \leq i, j \leq |\mathcal{D}_K|$. Each pair (tid_i, tid_j) along with the associated separating point $a(e_{s_{ij}})$ is computed and materialized as set V. Then set V is sorted in non-decreasing order of $a(e_{s_{ij}})$. If two separating vectors have the same $a(e_{s_{ij}})$ value, we sort them according to their projection in the s_1-axis.

The algorithm then sweeps the (positive quadrant of the) plane, going through the separating vectors in V in sorted order. The algorithm maintains also a set R that stores (unsorted) the K tuples with highest score according to the function f_e, where e is the current position of the sweeping vector. We initialize R to hold the *top-k* tuples with respect to the initial position of vector e, namely $e = \langle(0,0)(1,0)\rangle$ (function $f_{(1,0)}$). Initializing R is easy, since the set $\mathcal{D}_K$ computed at the end of algorithm *DominatingSet* is sorted by s_1^i.

Each $a(e_{s_{ij}})$ in V (and the corresponding vector $e_{s_{ij}}$) is associated with two tuple identifiers (t_i, t_j). When e crosses the vector $e_{s_{ij}}$ during the sweep it causes the ordering of tuples t_i, t_j to change according to Lemmas 4 and 5. In case both tuple identifiers belong to R, or neither belongs to R, we can safely discard the vector $e_{s_{ij}}$ under consideration, since it does not affect the composition of R. Otherwise, we materialize $a(e_{s_{ij}})$ together with the composition of R, and we update R, to reflect the new tuple identifiers. We also materialize the last value of R, after the sweep is completed.

At the end of the algorithm we have accumulated M separating vectors $e_1, e_2, \ldots, e_M$ (represented by their separating points $a(e_i), 1 \leq i \leq M$). The collection of vectors $e_i, 1 \leq i \leq M$ partitions the quadrant into $M+1$ regions. Each region i, $0 \leq i \leq M$, is defined by vectors e_i, e_{i+1}, where $e_0 = \langle(0,0)(1,0)\rangle$, and $e_{M+1} = \langle(0,0)(0,1)\rangle$. Region i is associated with a set of K points $R_i \subseteq \mathcal{D}_K$, such that for any vector e, with $a(e_i) \leq a(e) \leq a(e_{i+1})$, uniquely identifying a function $f_e \in \mathcal{L}$, $T_K(e)$ is equal to a permutation of R_i. This permutation is derived by evaluating f_e on every element of R_i and then sorting the result in nondecreasing order. That is, R_i contains (up to a permutation) the answer to any *top-k* query, $k \leq K$, for any function defined by a

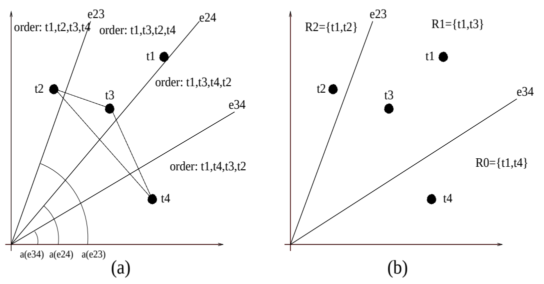


Figure 7. Example operation of algorithm *ConstructRJI*

vector in region i. We illustrate the operation of the algorithm with the following example.

Example 2 *Consider Figure 7(a). It presents a set $\mathcal{D}_2$ consisting of four tuples t_1, t_2, t_3, t_4. The algorithm starts by computing the separating vector for each pair of tuples. For brevity in Figure 7(a) we present the separating vectors only for pairs of tuples t_2, t_3, t_4. The separating vectors e_{34}, e_{24}, e_{23} are computed for each pair as shown in Figure 7(a). Each pair is stored along with the associated separating point and the collection is ordered based on separating points. Setting $K = 2$, we construct an index answering* top-1 *and* top-2 *join queries. Consider now a vector e sweeping the plane. The first two tuples in $Ord_{(1,0)}(\mathcal{D}_2)$ are $R = \{t_1, t_4\}$. The first vector crossed by e is e_{34}, which corresponds to swapping tuples t_3 and t_4. The swap changes the composition of R. In particular, t_4 is replaced with t_3. At this point, $a(e_{34})$ is stored along with the $R_0 = R = \{t_1, t_4\}$ and the current composition of R becomes $R = \{t_1, t_3\}$. Then $a(e_{24})$ is encountered in the sorted order but the swap of t_2, t_4 does not affect the composition of R. The next vector in the sorted order is e_{23}. The composition of R is affected so $a(e_{23})$ is stored along with $R_1 = R = \{t_1, t_3\}$, and the current composition of R changes to $R = \{t_1, t_2\}$. When the input is exhausted, the current ordering $R_2 = R = \{t_1, t_2\}$ is stored, and the algorithm terminates. Figure 7(b) shows the final partitioning of the plane.*

We organize the separating points along with the associated R_i's in a B-tree, indexed by $a(e_i), 1 \leq i \leq M$, storing in the leaves of the B-tree, the sets R_i of tuple identifiers in $\mathcal{D}_K$. We now proceed with the space and performance analysis of this structure.

6.1 Analyzing Algorithm *ConstructRJI*

Critical to the size of the index is the size of M, the number of separating vectors identified by the algorithm. We provide a worst case bound on M by bounding the number of times that a tuple identifier can move from position $K + 1$ to position K in $Ord_e(\mathcal{D}_K)$. Lemmas 4, 5 guarantee that whenever a swap happens between elements of $Ord_e(\mathcal{D}_K)$, it takes place between two *adjacent* elements in $Ord_e(\mathcal{D}_K)$. Thus, we only index the separating vectors that cause a swap of the elements in positions K and $K+1$ in $Ord_e(\mathcal{D}_K)$, since these are the ones that cause the composition of T to change. For every $t_i \in \mathcal{D}_K$ define $rank_{t_i}(e)$ to be the position of tuple t_i in the ordering $Ord_e(\mathcal{D}_K)$. The following lemma provides the means for bounding the value of M.

Lemma 6 *For every tuple $t_i \in \mathcal{D}_K$, $rank_{t_i}(e)$ can change from $K + 1$ to K at most K times for every vector e.*

We claim the following theorem.

Theorem 1 *Given a set of dominating points $\mathcal{D}_K$, we can construct an index for top-k join queries in time $O(|\mathcal{D}_K|^2 \log |\mathcal{D}_K|)$ using space $O(|\mathcal{D}_K|K^2)$ providing answers to top-k join queries in time $O(\log |\mathcal{D}_K| + k \log k)$, $k \leq K$.*

Lemma 6 guarantees that each element in $\mathcal{D}_K$ contributes at most K changes to $T_K(e)$. This means that each tuple introduces at most K separating vectors and consequently introduces K separating points that need to be stored in the worst case. Therefore, the number M of separating points is at most $O(|\mathcal{D}_K|K)$. After the separating points $a(e_s)$ are identified, they are organized along with the associated sets R_i in a B-tree indexed by $a(e_s)$. The leaf level stores pointers to the sets R_i. Thus, the total space requirement becomes $O(|\mathcal{D}_K|K^2)$. There are $O(nK)$ elements in $\mathcal{D}_K$ in the worst case, so the number M of separating points that require representation in the index is at most $O(nK^2)$. Thus, the total space used by this structure in the worst case is $O(nK^3)$. The worst case time complexity for constructing the ranked join index is: $O(n^2K^2)$ time to compute the separating vectors and separating points; $O(n^2K^2 \log n^2K^2)$ time to sort the separating points. Constructing a B-tree can be performed during the single scan on the sorted separating point collection of algorithm *ConstructRJI*. Thus, the total construction time is $O(n^2K^2 \log(n^2K^2))$. We note that these are the worst case space and construction time requirements for the index *RJI*. In section 8 we will experimentally evaluate the requirements of *RJI* for a variety of data distributions.

At query time given the vector e that defines a function $f_e \in \mathcal{L}$, we compute $a(e)$, and search the B-tree using $a(e)$ as a key. This effectively identifies the region that contains vector e. Then, we retrieve the associated set R_i and evaluate f_e for all elements of R_i, sorting the results to produce $T_k(e)$. Thus, query time is $O(\log(nK^2) + K \log K)$ in the worst case, for any *top-k* join query, $k \leq K$.

6.2 Space/Time Tradeoffs in *RJI*

Our ranked join index design provides a variety of space-time tradeoffs which can be utilized to better serve the performance/space constraints in various settings.

If the space is a critical resource, one could decrease the space requirements significantly, at almost no expense on query time. Note that sets R_i and R_{i+1} associated with two

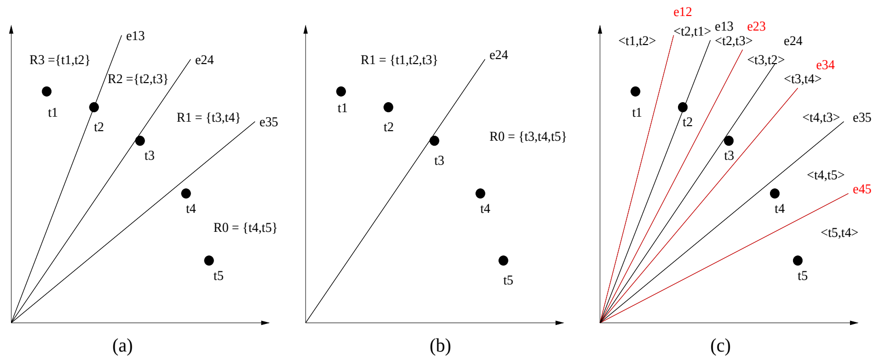


Figure 8. **Space Time Tradeoffs of *RJI***

neighboring regions differ, in the worst case, by only one tuple. Therefore, the set $R_i \cup R_{i+1}$ contains $K+1$ distinct tuples. If we merge m regions, then the resulting region contains at most $K+m-1$ distinct tuples. Note that this is a worst case bound; depending on the distribution, a region may contain less than $K+m-1$ distinct tuples. Therefore, if we initially have M separating vectors, merging every m regions reduces the number of separating vectors to M/m. The space for the index becomes $O(M(K+m)/m)$, and the query time $O(\log(M(K+m)/m)+(K+m)\log(K+m)$. Since $M = O(nK)$ in the worst case, the requirements of the index are $O(nK^2(K+m-1)/m)$ for space, and $O(\log(nK^2(K+m-1)/m)+(K+m-1)\log(K+m-1))$ for query time. An example is shown in Figure 8 ($K=2$). We merge every 2 regions of Figure 8(a) showing the result in Figure 8(b).

Merging m regions does not always result in a region with $K+m-1$ tuples. Depending on the distribution of the rank values, it may be the case that as we cross the vectors that define the m regions, some points move in and out of the top K positions multiple times. In this case, merging m regions, results in a region with far less than $K+m-1$ distinct tuples. Instead of merging every m regions, we can merge so that every region (except possibly the last one) contains *exactly* $K+m-1$ distinct tuples. This allows for more aggressive reduction of space, without affecting the worst case query time. We measure the effects of merging adjacent regions on space, in Section 8.

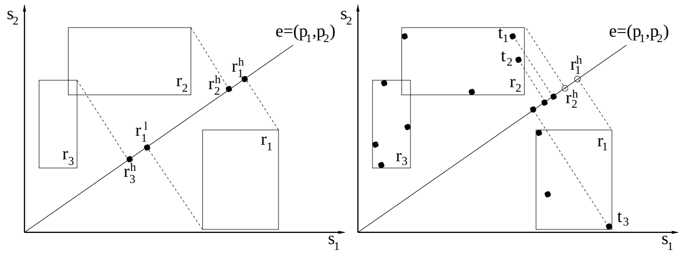
If fast query time is the main concern, one can reduce the query time by storing all separating vectors that cause $T_K(e)$ to change. According to Lemma 6 a tuple can move from position $\ell+1$ to ℓ at most ℓ times, therefore, each tuple can contribute at most $1+2+\ldots+K = K(K+1)/2$ changes to $T_K(e)$. Thus, storing at most $O(nK^3)$ separating vectors one could reduce the query time to $O(\log nK^3)$. Effectively in this case we are storing an ordered sequence of points in each region R_i so there is no need for evaluating f_e on the elements of the region; the ordered sequence (according to f_e) can be returned immediately. Figure 8 presents an example of this tradeoff as well. We materialize the separating points causing a change in ordering for tuples in each region of Figure 8(a). The result is shown in Figure 8(c).

7 A Solution Based on R-trees

In this section we propose a variant of the range search procedure of an R-tree index that is specifically designed to answer *top-k* join queries. This provides a base-case for performance comparison against our solution. The basic idea, is to employ an R-tree index to prune away a large fraction of the tuples that are bound not to be among the top k. We refer to this modified R-tree as the *TopKrtree*.

Consider the 2-dimensional space defined by the 2 rank values associated with each tuple in $\mathcal{D}_K$, returned by the algorithm *DominatingSet*. We build an R-tree on these points using traditional R-tree construction algorithms [11, 2]. A basic observation is that due to the monotonicity property of the functions $f \in \mathcal{L}$, given a Minimum Bounding Rectangle (*MBR*) r at *any* level in that tree, the minimum and maximum score values for *all* tuples inside r are bounded by the value any scoring function in $\mathcal{L}$ gets at the lower left and upper right corners of r. Following this observation we modify the R-tree search procedure as follows. At each node in the R-tree, instead of searching for overlaps between MBRs, the procedure searches for overlaps between the intervals defined by the values of the scoring function in the upper right and lower left corners of the MBRs. The algorithm recursively searches the R-tree and maintains a priority queue collecting k results.

Consider an R-tree with three MBRs, namely r_1, r_2, and r_3, and a *top-k* join query with $e = (p_1, p_2)$. This situation is depicted in Figure 9(a). The largest score that a point in an MBR can possibly achieve is the score given by the projection of the upper right corner of the MBR on vector e. We will refer to this projection as the *maximum-projection* for the MBR, and the MBR that has the largest maximum-projection among all the MBRs of the same R-tree node as the *master* MBR. Similarly, the lowest score is given by the projection of the lower left corner (*minimum-projection*) of the MBR. A simplified version of the algorithm, named *TopKrtreeAnswer*, is presented in Figure 10. For brevity, we will assume that each MBR contains at least K tuples. Therefore, we can present the algorithm guiding the search using



(a) (b)

Figure 9. **A graphical representation for the *TopKrtreeAnswer* algorithm**

```
Input: A number $k$ and a preference vector $e = (p_1, p_2)$.
Output: The answer-set $S$ to the top-k query.

1 procedure TopKrtreeAnswer()
2     let $S = \emptyset$ be a priority queue with space for exactly $k$ values;
3     ProcessRtreeNode(root of rtree, $S$);
4     return($S$);

5 procedure ProcessRtreeNode(node $N$, $S$)
6     if ($N$ is a leaf)
7         for (all tuples $t$ in this node)
8             insert $t$ in $S$;
9     else
10        let $r$ range over all the MBRs in $N$;
11        let $r_{max} = \arg\max_r$ {maximum projection of MBR $r$
                                      on preference vector $e$};
12        let $r_{max}^{low}$ = {minimum projection of MBR $r_{max}$
                              on preference vector $e$};
13        for (each subtree rooted at each MBR $c$ of $N$)
14            if (maximum projection of MBR $c \geq r_{max}^{low}$)
15                ProcessRtreeNode($c$, $S$);
16    return($S$);
```

Figure 10. **The TopKrtreeAnswer algorithm**

only the master MBR at each R-tree level. Accounting for the case where multiple MBR's are required[1] is immediate by maintaining a list of candidate MBRs ordered by their maximum projections at each level. This resembles the type of search performed while answering nearest-neighbor queries using R-trees [15]. In the algorithm presented in Figure 10 the MBR with the largest maximum-projection is always the candidate to search and expand further for obtaining the answer to the *top-k* query. This is rectangle r_1 in Figure 9(a), since its maximum-projection r_1^h is the largest among the three MBRs. In this case, we can safely prune away all MBRs with maximum-projection less than the minimum-projection of the master MBR. In our example we will not examine the tuples in r_3, since all these tuples have scores less than the minimum score of all the tuples in r_1. However, the algorithm will examine all MBRs with maximum-projection greater than the minimum-projection of the master MBR. The range of projections of such MBRs overlap, and the answer to the *top-k* query may be a collection of tuples coming from all those MBRs. Therefore, in order to get the correct answer we must examine all the MBRs whose projections on vector e overlap with the projection of the master MBR.

[1]This could happen if the search using only the master MBR does not yield k results in the leaf R-tree level for a *top-k* join query.

Note that there are many cases in which the *TopKrtree* accesses more MBRs than really necessary. Consider Figure 9(b), showing a *top-2* query with $e = (p_1, p_2)$. Evidently, the answer to this query is the set of tuples $\{t_1, t_2\}$, both contained in r_2. Observe that even though r_1 has the largest maximum-projection (that is r_1^h) none of its tuples (e.g., t_3) are contained in the *top-2* answer. Thus, all the computations involving r_1 are useless in this case.

8 Experimental Evaluation

We implemented our proposal and conducted a series of experiments to evaluate the efficiency of our techniques. We also implemented the *TopKrtree* to compare against *RJI*. We start this section by describing the datasets we used in our evaluation. Then, we experimentally examine the properties of the algorithms proposed herein and assess the efficiency of our solutions in a variety of settings.

8.1 Description of Experiments

We implemented the algorithms described herein in C++ under SunOS v5.8, and run the experiments on a SUN Blade 1000 server with two UltraSPARC-III processors.

In order to test the proposed algorithms we used both synthetic and real datasets. The synthetic ones are generated by sampling uniform, Gaussian, and Zipfian distributions. The size of the join result for all the synthetic datasets was 10,000-1,000,000 tuples. The datasets are generated as follows.

Uniform: Rank values for each rank attribute in the uniform dataset (denoted *unif*) lie in the range [0,100].

Gaussian: Rank values for each rank attribute in the Gaussian dataset (denoted *gauss*) are generated with mean value 400 and standard deviation 5. (In our experiments we varied the standard deviation, but the results were similar, and we omit them for brevity.)

Zipfian: Rank values for each rank attribute in the zipfian data set are produced using a generalized zipfian distribution. The generalized zipfian distribution is defined as $f_r \propto 1/r^\theta$, where f_r is the occurrence frequency of the r-th value (sorted on decreasing frequency of occurrence), and θ is a parameter controlling the skew of the distribution. We produced two datasets, one with skew parameter 0.1 (*Zipf0.1*), and the other one with skew 2 (*Zipf2*).

Real: Our data sets are generated by parsing $HTML$ and XML pages from the web, and constructing two data sets recording various statistics for each page (such as the in and out degree in terms of number of links to/from a page, the size

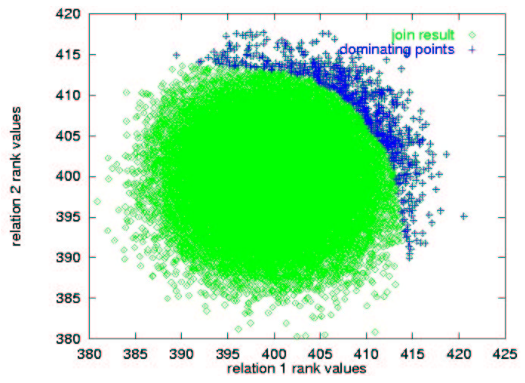
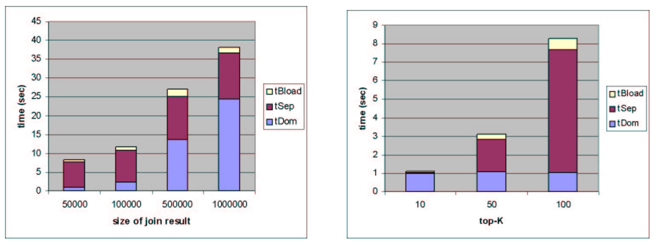


Figure 12. **Rank value distribution for the join result and dominating points for the *gauss* dataset**



(a) varying the join result size (b) varying K

Figure 14. **Breakdown of the time to construct the *RJI* index for the *unif* dataset**

of each page, etc). Our first data set, which we refer to as *real_web* is the outcome of the join of two data sets named *real_web indegree* and *real_web outdegree* recording the in and out degree for a collection of web pages (the join takes place on the page id). This data set consists of 370,000 tuples. Our second data set, *real_xml*, is the outcome of the join of two data sets named *real_xml size* and *real_xml outdegree* recording the size and outdegree of a collection of XML documents collected from the web uniformly at random (the join takes place on document id). This set consists of 160,000 tuples. The statistical properties of the collections we joined to produce the two real datasets are reported in Table 1.

8.2 Evaluating the *RJI* Construction Algorithm

In the first set of experiments we evaluate the effectiveness of the pruning strategies presented in Section 4. We measure the number of elements in set $\mathcal{D}_K$ after the execution of the algorithm (labeled *Dom*) and the number of separating points represented in *RJI* (labeled *Sep*). Figure 11 depicts the sizes of the *Dom* and *Sep* sets as a function of K, for the uniform (Figure 11(a)) and Gaussian (Figure 11(b)) datasets. We also show the same graphs for the zipfian (Figure 11(c)) and real (Figure 11(d)) datasets.

We report the size of each set as a percentage of the size of the join result. Observe that the number of points that our algorithm has to consider, namely the number of dominating points, is significantly smaller than the size of the entire join in all cases. In our experiments this number is less than 6% of the join size. Figure 12 gives a visual representation of the join result (depicted in light color) and the *Dom* set (shown in dark color) for the *gauss* dataset. (For this example the join result has 50,000 tuples, and K=100.) Therefore, this pruning step is extremely effective in reducing the size of the problem. Moreover, the number of separating points *RJI* stores is in most cases only a fraction of the number of dominating points. Consequently, the size of the *RJI* index remains small compared to the join result. The graphs indicate that the sizes of the *Dom* and *Sep* sets grow gracefully with parameter K.

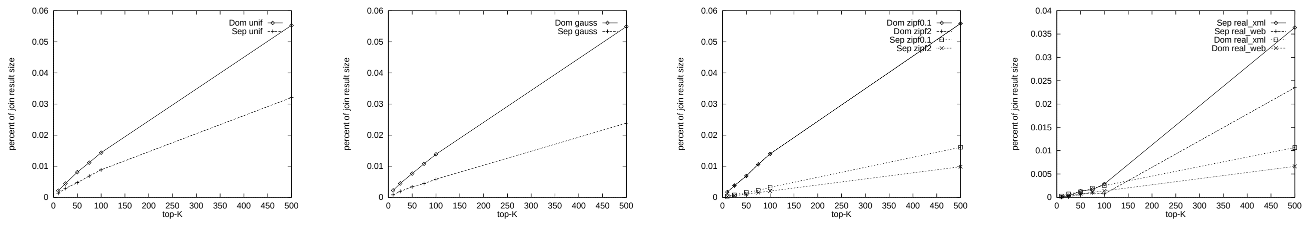
In our second set of experiments we explore the performance of the algorithm when the size of the join result increases. The size of the datasets we use range from 50,000 to 1,000,000 tuples. Figures 13(a)(b) report the size of *Dom* and *Sep*, for the uniform dataset, as a function of the size of the join result. The corresponding numbers for the *Zipf2* dataset are reported in Figures 13(c)(d). One can observe that the sizes of the above sets remain relatively stable as the join result size increases (for the same value of K and data distribution). It is evident that the pruning applied by algorithm *DominatingSet* is effective. Keeping the size of the dominating set small as the join result increases, decouples the time required to build the *RJI* index from the size of the join result.

Figure 14 provides a breakdown of the total time required to build an *RJI* index into three components, namely, the time to compute the *Dom* set (*tDom*), the time to compute the *Sep* set (*tSep*), and finally the time to populate the B-tree index (*tBLoad*). (We only provide the graphs for the uniform dataset. The rest of the datasets exhibit similar behavior and the corresponding graphs are omitted for brevity.)

According to Figure 14(a) as the join result size increases, more time is required to compute the set of dominating points, while the time spent on the other two components increases minimally. This is because, although the number of dominating points remains relatively stable with increasing join result size, the algorithm still has to make a complete pass over the join result to identify the dominating points. This explains the proportional increase in construction time with respect to the join result size. The effect of varying the parameter K in the construction time of our index is illustrated in Figure 14(b). As K increases the index construction time is dominated by the time required to identify separating points. This includes the time to compute all the separating vectors (and points), sort them, and determine which of those to store in the B-tree index. As K increases, more dominating points affect the composition of set R (in Figure 6) and more separating points are introduced, increasing the corresponding time component.

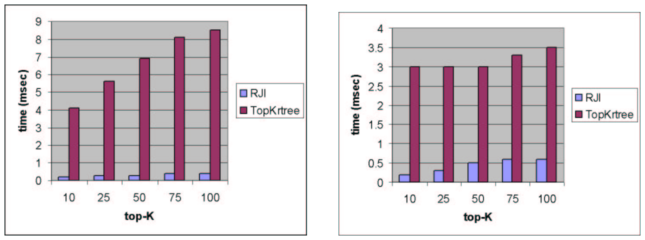
dataset	*min*	*max*	*mean*	*median*	*std.dev.*	*skew*
real_web indegree	1	100288	6.17	1	152.70	520.47
real_web outdegree	1	826	7.02	3	14.92	10.48
real_xml size	10	500608	4641.09	1071	20814.03	12.49
real_xml outdegree	1	5520	13.18	4	46.62	29.89

Table 1. The statistical properties of the *real_web* and *real_xml* datasets



(a) *unif* (b) *gauss* (c) *zipf0.1* and *zipf2* (d) *real_web* and *real_xml*

Figure 11. The effect of K on the size of *Dom* and *Sep*. The size of the join result for the synthetic data is 50,000 tuples.



(a) Dataset *unif* (b) Dataset *real_web*

Figure 15. Time to answer *top-k* queries, varying K

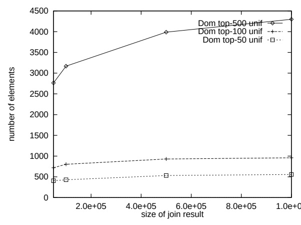
8.3 Answering *top-k* Queries

In the following paragraphs we examine the time required by the *RJI* index to answer *top-k* join queries, and we compare it against the time required by *TopKrtree*. All times are averages of all queries in our workload. We also analyze the space requirements of *RJI* in comparison with that required by the solution using R-trees. Both indices are disk resident. For these experiments, we reduce the space requirements of our index by merging regions as described in Section 6.2, so that each region contains *exactly* $2K$ tuples.

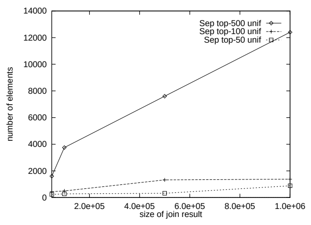
The graphs in Figures 15(a)(b) compare the performance of the two indices as K increases for datasets *unif* and *real_web*. Each point in the graphs represents the average response time of 500 *top-K* queries distributed uniformly at random over the space of all possible queries. The experiments show that *RJI* consistently outperforms *TopKrtree* in these data sets, answering queries up to 17 times faster. The trends in performance gains are similar as K increases beyond the range of values shown. This performance advantage is pronounced as the size of the datasets indexed increases (the graphs presenting response time for *top-k* queries with increasing dominating set size are omitted due to space limitations). The large difference comes from the fact that *TopKrtree* accesses a considerable amount of tuples that prove to be useless. These results, experimentally confirm our analytical expectations. Our *RJI* design provides worst case performance guarantees in contrast with R-trees that in the worst case have to touch every tuple (linear in the size of the indexed data set). Similar behavior is also exhibited by the other synthetic and real datasets.

Figure 16, compares the total size (space occupied by both index nodes and data nodes), in terms of bytes, required by the *RJI* and the R-tree as a function of K. (In these experiments the size of the join result for the synthetic datasets is 50,000 tuples.) In all cases the total size required by the *RJI* index is significantly smaller than that required by the R-tree index. Recall that the R-tree is storing the entire set of dominating points, which in many cases is a superset of the set of points needed to answer the *top-k* join queries. In addition, R-trees require more space due to the overhead they impose by representing MBRs as two dimensional rectangles. In contrast, the *RJI* index only stores the points that are useful in answering the *top-k* join queries. Furthermore, by merging adjacent regions we are able to reduce the storage requirements considerably since adjacent regions have a large number of points in common.

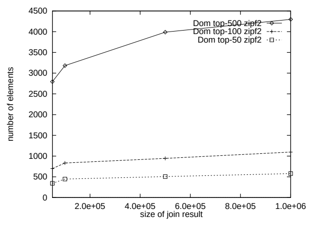
The experiments show that *RJI* consistently requires less space for indexing and at the same time is capable of answering *top-k* join queries much faster than the R-tree solution. For the synthetic datasets, *RJI* can answer queries up to 17 times faster, while occupying 10%-50% of the R-tree space (Figures 16(a)(b)). In the case of the *real_web* and *real_xml* datasets the overall trends are similar. The storage requirements of our index are 3-10 times less than the R-tree approach (Figures 16(c)(d)), and at the same time *RJI* answers queries up to 15 times faster. For all the above datasets, the space and time trends remain consistent as K increases beyond the range shown in the figures.



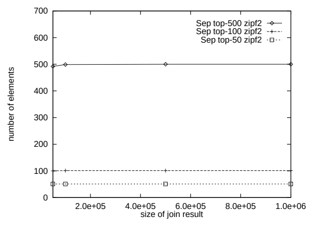
(a) size of Dom



(b) size of Sep

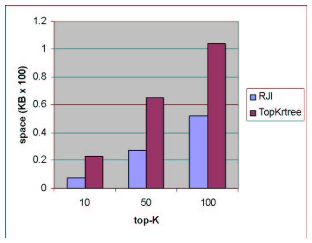


(c) size of Dom

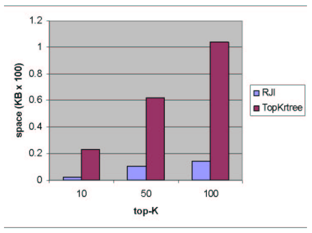


(d) size of Sep

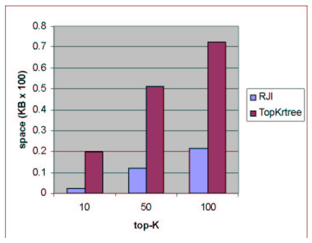
Figure 13. **The *Dom* and *Sep* sizes as a function of the join result size for data sets *unif* and *Zipf***



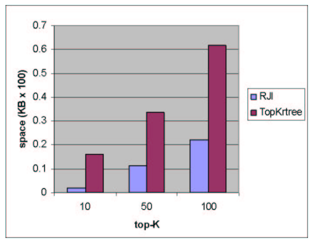
(a) Dataset *unif*



(b) Dataset $Zipf2$



(c) Dataset *real_web*



(d) Dataset *real_xml*

Figure 16. **Overall space (index and data) required to answer *top-k* queries, varying K**

9 Conclusions

We have considered a novel type of join index supporting efficiently queries ranking (based on user specified preferences) the join of two relations. Our index structure, called *RJI*, answers fast *top-k* queries on the *join* of two relations for a large family of functions used to compute the score of tuples in the join result. We showed that only a fraction of the join result requires representation in our index and proposed efficient algorithms to construct an *RJI* providing worst case guarantees on its performance. We have presented an extensive experimental evaluation that proves the validity of our approach, and shows that *RJI* considerably outperforms alternative solutions.

This work raises various questions for further exploration. First, it would be worthwhile to study extensions of our index scheme to more than a pair of relations. This would involve generalizing *RJI* in dimensions more than two. In addition incremental maintenance of our index is important and the center of our current work is in this direction.

Acknowledgements

We would like to thank Davood Rafiei and Laurent Mignet for providing us with the *real_web* and *real_xml* datasets respectively. We would also like to thank Faith Fich for helpful discussions.

References

[1] R. Agrawal and E. Wimmers. A Framework For Expressing and Combining Preferences. *Proceedings of ACM SIGMOD*, pages 297–306, June 2000.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R* - tree: An Efficient and Robust Access Method for Points and Rectangles. *Proceedings of ACM SIGMOD*, pages 220–231, June 1990.

[3] N. Bruno, L. Gravano, and A. Marian. Evaluating Top-k Queries Over Web Accessible Databases. *Proceedings of ICDE*, Apr. 2002.

[4] K. Chang and S.-W. Huang. Minimal Probing: Supporting Expensive Predicates for Top-k Queries. *Proceedings of ACM SIGMOD*, June 2002.

[5] Y. chi Chang, L. Bergman, V. Castelli, C. Li, M. L. Lo, and J. Smith. The Onion Technique: Indexing for Linear Optimization Queries. *Proceedings of ACM SIGMOD*, pages 391–402, June 2000.

[6] D. Donjerkovic and R. Ramakrishnan. Probabilistic Optimization of Top-N Queries. *Proceedings of VLDB*, Aug. 1999.

[7] R. Fagin. Combining Fuzzy Information from Multiple Systems. *PODS*, pages 216–226, June 1996.

[8] R. Fagin. Fuzzy Queries In Multimedia Database Systems. *PODS*, pages 1–10, June 1998.

[9] R. Fagin and E. Wimmers. Incorporating User Preferences in Multimedia Queries. *ICDT*, pages 247–261, Jan. 1997.

[10] L. Gravano and S. Chaudhuri. Evaluating Top-k Selection Queries. *Proceedings of VLDB*, Aug. 1999.

[11] A. Guttman. R-trees : A Dynamic Index Structure for Spatial Searching. *Proceedings of ACM SIGMOD*, pages 47–57, June 1984.

[12] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Efficient Execution of Multiparametric Ranked Queries. *Proceedings of SIGMOD*, June 2001.

[13] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Joining Ranked Inputs in Practice. pages 950–961, Hong Kong, China, Aug. 2003.

[14] A. Natsev, Y.-C. Chang, J. Smith, C.-S. Li, and J. S. Vitter. Supporting Incremental Join Queries on Ranked Inputs. *Proceedings of VLDB*, Aug. 2001.

[15] N. Roussopoulos, S. Kelly, and F. Vincent. Nearest Neighbor Queries. *Proceedings of ACM SIGMOD*, pages 71–79, May 1995.

[16] P. Valduriez. Join Indexes. *ACM TODS, Volume 12, No 2*, pages 218–246, June 1987.