A method and apparatus for ranked join indices includes a 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. The concepts of ranking join indices presented herein are also applicable in the case of a single relation. In this case, the concepts herein provide a 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.

19 Claims, 10 Drawing Sheets
<table>
<thead>
<tr>
<th>PARTS</th>
<th>SUPPLIERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>availability</td>
<td>name</td>
</tr>
<tr>
<td>5</td>
<td>PO5</td>
</tr>
<tr>
<td>2</td>
<td>PO5</td>
</tr>
<tr>
<td>9</td>
<td>PO5</td>
</tr>
</tbody>
</table>

Table and Rank Attributes

Figure 1
DominatingSet($C, K$)
Initialize priority queue $Q$ and $D_K = \emptyset$.
Sort the join result in non-increasing order of the $s_2$ rank value.
For the $i$th tuple $t_i$ with rank values $(s_1^i, s_2^i)$
    if ($|Q| < K$)
        include $t_i$ in $D_K$
        include $s_1^i$ in $Q$
    else
        if $s_1^i \leq \min\{Q\}$ discard $t_i$
        else
            include $t_i$ in $D_K$
            include $s_1^i$ in $Q$
            if $|Q| > k$ delete the minimum element of $Q$

Output $D_K$

The Dominating Set Algorithm

Figure 2
<table>
<thead>
<tr>
<th>PARTS</th>
<th>availability</th>
<th>name</th>
<th>supplier id</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO5</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PO5</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PO5</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUPPLIER</th>
<th>supplier id</th>
<th>quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PARTS</th>
<th>availability</th>
<th>name</th>
<th>supplier id</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO5</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PO5</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PO5</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUPPLIER</th>
<th>supplier id</th>
<th>quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Examples of set $D_K$ for different join result

Figure 3
Algorithm ConstructRJI

Figure 5
Figure 6

Example operation of algorithm Construct RJL

(a) e23

(b) e34

R1 = \{t1, t3\}
R2 = \{t1, t2\}
R0 = \{t1, t4\}
Space Time Tradeoffs of RJL

Figure 7

(a) R3={(1,2)}
R2={(2,1)}
e_{13}

(b) R1={(1,2,1)}
R0={(3,4,5)}
e_{24}

(c) e_{12}, e_{23}, e_{24}, e_{34}, e_{35}, e_{45}, e_{54}
A graphical representation for the Top-kTreeAnswer algorithm

Figure 8
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 TopKtreeAnswer()
2 \hspace{1em} let \( S = \emptyset \) be a priority queue with space for exactly \( k \) values;
3 \hspace{1em} ProcessRtreeNode(root of rtree, \( S \));
4 \hspace{1em} return(\( S \));

5 procedure ProcessRtreeNode(node \( N \), \( S \))
6 \hspace{1em} if \( (N \) is a leaf)
7 \hspace{2em} for (all tuples \( t \) in this node)
8 \hspace{3em} insert \( t \) in \( S \);
9 \hspace{1em} else
10 \hspace{2em} let \( r \) range over all the MBRs in \( N \);
11 \hspace{2em} let \( r_{max} = \arg \max_r \{ \text{maximum projection of MBR } r \) on preference vector \( e \}\};
12 \hspace{2em} let \( r_{low} = \{ \text{minimum projection of MBR } r_{max} \) on preference vector \( e \}\};
13 \hspace{2em} for (each subtree rooted at each MBR \( c \) of \( N \))
14 \hspace{3em} if (maximum projection of MBR \( c \geq r_{low} \))
15 \hspace{4em} ProcessRtreeNode(\( c \), \( S \));
16 \hspace{1em} return(\( S \));

The Top KtreeAnswer algorithm

Figure 9
1

METHOD AND APPARATUS FOR RANKED JOIN INDICES

CROSS-REFERENCES TO RELATED APPLICATIONS

This application is a continuation of U.S. patent application Ser. No. 10/775,056 filed Feb. 9, 2004, now U.S. Pat. No. 7,185,012, which claims the benefit of U.S. Provisional Application No. 60/446,237, filed Feb. 10, 2003, which is herein incorporated by reference in its entirety.

FIELD OF THE INVENTION

The present invention relates generally to the ranking of data entities and, more particularly, to a method and apparatus for ranked join indices.

BACKGROUND OF THE INVENTION

Many data sources contain data entities that may 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 an attribute domain. Such values may reflect various quantities of interest for the entities, such as physical characteristics, quality, reliability or credibility to name a few. Such attributes are referred to 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 values (e.g., an interval of continuous values). The existence of rank attributes along with data entities leads to enhanced functionality and query processing capabilities.

Typically, users specify their preferences toward specific attributes. Preferences are expressed in the form of numerical weights, assigned to rank attributes. Query processors incorporate functions that weight attribute values by user preference, deriving scores for individual entities. Several techniques have been developed to perform query processing with the goal of identifying results that optimize such functions. A typical example 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.

Several prior art techniques disclose a framework for preference-based query processing. Such works consider 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. Most of these techniques for answering top-k selection queries, however, are not based on indexing. Instead, they are directed 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 or minimization of random disk access.

However, the few available techniques that do disclose indexing for answering top-k selection queries do not provide guarantees for performance and in the worst case, an entire data set has to be examined in order to identify the correct answer to a top-k selection query.

SUMMARY OF THE INVENTION

The inventors disclose herein a technique, referred to by the inventors as ranked join index, for efficiently providing solutions to top-k join queries for arbitrary, user specified preferences and a large class of scoring functions. The rank join index technique of the present invention requires small space (i.e., as compared to an entire join result) and provides performance guarantees. Moreover, the present invention provides a tradeoff between space requirements and worst-case search performance.

In one embodiment of the present invention a method of creating a ranked join index for ordered data entries includes determining a dominating set of the ordered data entries, mapping the dominating set of ordered data entries according to rank attributes, determining a separating vector for each set of adjacent mapped data entries, and ordering and indexing the data entries according to a separating point associated with each of the separating vectors.

BRIEF DESCRIPTION OF THE DRAWINGS

The teaching of the present invention can be readily understood by considering the following detailed description in conjunction with the accompanying drawings, in which:

FIG. 1 depicts two tables each comprising a list of attributes and rankings for the attributes;
FIG. 2 depicts an embodiment of an algorithm for computing the dominating set for substantially any value of K, where K depicts an upper bound for the maximum requested result size of any top-k join query;
FIGS. 3a and 3b graphically depict an example of a Dominating Set determined by a Dominating set algorithm for tables and rank attributes having different join results;
FIG. 4a graphically depicts an example of the ordering of two tuples when a vector has a positive slope;
FIG. 4b graphically depicts an example of the ordering of the two tuples for a second case when a vector has an other than positive slope;
FIG. 5 depicts an embodiment of an RJI Construct algorithm of the present invention, which preprocesses a set of tuples and constructs an index on its elements;
FIG. 6a and FIG. 6b graphically depict an example of the operation of the RJI Construct algorithm;
FIGS. 7a, 7b and 7c graphically depict an example of the space-time tradeoffs of the RJI Construct algorithm of FIG. 5;
FIG. 8a and FIG. 8b graphically depict an embodiment of an R-tree with three MBRS and a top-k join query;
FIG. 9 depicts an embodiment of a TopKtree Answer algorithm of the present invention; and
FIG. 10 depicts a high level block diagram of an embodiment of a controller suitable for performing the methods of the present invention.

To facilitate understanding, identical reference numerals have been used, where possible, to designate identical elements that are common to the figures.

DETAILED DESCRIPTION

Although various embodiments of the present invention herein are being described with respect to techniques for providing performance guarantees for top-k join queries over two relations, it will be appreciated by those skilled in the art informed by the teachings of the present invention that the concepts of the present invention may be applied to providing performance guarantees for join queries over substantially any number of relations.

FIG. 1 depicts two tables each comprising a list of attributes and rankings for the attributes. For example, FIG. 1 comprises a first table labeled Parts. The Parts table comprises three attributes, namely; availability, name and supplier id.
FIG. 1 further comprises a second table labeled Suppliers. The Suppliers table comprises two attributes, namely, supplier id and quality. For purposes of explanation, it is assumed that all parts correspond to the same piece of a mechanical device, illustratively part PI85, possibly of different brands. The rank attributes, availability and quality, determine the availability (i.e., current quantity in stock for this part) and the quality of the supplier (i.e., acquired by, for example, user experience reports on a particular supplier) respectively, having as a domain a subset of R+ (i.e., the greater the value the larger the preference towards that attribute value). 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 to 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 each rank attribute (i.e., the larger the weights the greater the preference of the user towards these rank attributes). Assuming the existence of scoring functions that combine user preferences and rank attribute values returning a numerical score, the target queries of the present invention identify the K tuples in the join result of, for example in FIG. 1, Parts and Suppliers with the highest scores.

For example, let R, S depict two relations, with attributes A1-A6 and B1-B6 respectively. A1, B1 are rank attributes with domains a subset of R+ and 0, an arbitrary join condition defined between (sub)sets of the attributes A1-A6, B1-B6 (R⋈A1S). For a tuple, t∈R⋈A1S, A1(t) (and similarly B1(t)) corresponds to the value of attribute A1 (and similarly B1) of tuple t. Furthermore, let f: R×R→R+ be a scoring function that takes as input the pair of rank attribute values (s1,s2)=(A1(t), B1(t)) of tuple t∈R⋈A1S, and produces a score value f(s1,s2) for the tuple t. It should be noted that a function f: R×R→R+ is monotone if the following holds true: x1≤x2 and y1≤y2 then f(x1,y1)≤f(x2,y2).

For further explanation, let e=(p1, p2) denote the user defined preferences towards rank attributes A1, B1. As such, a linear scoring function, f(e): R×R→R+, is defined as a scoring function that maps a pair of score values (s1,s2) to the value f(e)(s1,s2)=p1s1+p2s2. It is assumed 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. The symbol, 'e', is used to denote the class of monotone linear functions. Notice that the pair of user defined preferences, e, uniquely determines a function, f(e).

Given the relations R, S, join condition θ and scoring function f(e), a top-K query returns a collection T_K(e)⊂R⋈A1S of K tuples ordered by f(e)(A1(t), B1(t)), such that for all t∈R⋈A1S, f(e)(A1(t), B1(t))≥f(e)(A1(u), B1(u)), for all u∈T_K(e), 1≤k≤K. Thus, a top-K join query results as a return k tuples from the join of two relations with the highest scores, for a user specified scoring function, f, among all tuples in the join result.

If the relations R, S to be joined consist of O(n) tuples, the size of the join relation R⋈A1S may be as large as O(n^2). The inventors determined and demonstrate herein that most of the tuples of the join relation, R⋈A1S, are typically not necessary for answering top-K join queries. In particular, for a fixed value K<n, where K depicts an upper bound for the maximum requested result size of any top-K join query, and for the entire class of linear functions f, 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∈K.

In addition, it should be noted that there is no need to generate the complete join result R⋈A1S. For example, let C denote the subset of R⋈A1S necessary to generate, in the worst case, an index, providing answers with guaranteed performance on any top-K join query, k∈K, issued using any scoring function f(x). 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, only t is joined 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 determined for each tuple, t∈R, only the K tuples with the highest rank values are required. Due to the monotonicity property of functions in f, these K tuples will have the highest scores for any for each f∈F. As such, the inventors disclose postulate one (1) which follows:

For relations of size O(n) and a value K, 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), any top-K join query, k∈K, for substantially any f∈F may be solved. In a preprocessing step, C may be determined by joining R and S and selecting for each tuple t, t∈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 may be carried out in a fully declarative way using a Structured Query Language (SQL) interface, which is well-known in the art.

For further reduction of the size of C, the inventors disclose letting t and t' denote two tuples of R⋈A1S and (s1,s2) and (s'1,s'2) denote the pairs of rank values associated with the tuples, respectively. Thus, tuple t dominates tuple t' if s1≤s'1 and s2≤s'2. The domination property provides a basic means to reduce C even further.

As such, two methods of reducing the size of C are disclosed herein. That is, the determination that for relations of size O(n) and a value K, the worst case size of C is O(nK), reduces a join result by restricting the number of the tuples contributed to the join by a single tuple of a relation. In addition the domination property described above reduces the size of C by examining the tuples contributed to the join by multiple tuples of a relation. As such, the inventors disclose postulate two (2) which follows:

For a value of K, if some tuple t∈R is dominated by at least K other tuples, then t cannot be in the solution set of any top-K join query, k∈K.

Thus, from the monotonicity properties of the scoring functions, it is evident that a viable strategy to reduce the size of C is to identify all tuples in C dominated by at least K tuples. Formally, given a set C, the dominating set, Dk, is the minimal subset of C with the following property: for every tuple t∉Dk with rank values (s1,s2), there are at least K tuples t∉Dk that dominate tuple t.

FIG. 2 depicts an embodiment of an algorithm for computing the dominating set, Dk, for substantially any value of K in accordance with the present invention. In the algorithm of FIG. 2, every tuple t, t∈C, is associated with a pair of rank values (s1,s2). The algorithm maintains a priority queue, Q (supporting insertions/deletions in logarithmic time) storing the K largest s1 rank values encountered so far. The algorithm first sorts the tuples in the join result in non-increasing order with respect to the s1 rank values. The tuples are then considered one at a time in that order. For every tuple t, after the first K, if its s1 rank value is less than the minimum rank
value present in Q, the tuple is discarded. Otherwise the tuple is included in the dominating set, and the priority queue, Q, is updated. The Algorithm Dominating Set of FIG. 2 requires a time equal to O(C(nlog(C))) for sorting and computes the dominating set D in a time equal to O(C(nlog(K)). The number of tuples reduced by the Dominating Set algorithm depends on the distribution of the rank value pairs in the join result. In practice the size of D is expected to be much smaller than O(nK). In the worst case, however, no tuple is dominated by K other tuples and, as a result, the Dominating Set algorithm does not achieve any additional reduction in the number of tuples.

FIGS. 3a and 3b graphically depict an example of a Dominating Set determined by the Dominating Set algorithm for tables and attributes such as those of FIG. 1, having two different join results. FIG. 3 depicts the two pairs of relations and the 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 FIGS. 3a and 3b, a geometric analogy is drawn and the tuple is represented by the rank attribute pair, (quality, availability), as a point in two-dimensional space. For the rank attribute value distributions in FIG. 3a, the set D has a size of 3 (worst case) since no tuple is dominated by any other tuple. Thus, in this case the Dominating Set algorithm determines the set D having a size equal to the theoretically predicted worst case. In contrast, in FIG. 3b, the Dominating Set algorithm determines a set D with a size of 1 and containing the tuple whose rank attribute pair dominates the other two, for K=1.

The relationship among the sets, Dk, associated with each top-k join query possible with k∈K may be characterized according to the following postulate, number three (3), which follows:

Considering two top-k join queries requesting k1, k2 results and k1<k2, for the dominating sets Dk1, Dk2, Dk then Dk1⊂Dk2⊂Dk.

Thus, it is determined that it is sufficient to identify and determine only the set Dk since the solutions to any top-k join query k∈K are contained in this set. This also holds true for any scoring function, f(x).

The inventors present above an algorithm to preprocess the set Dk and develop an index structure, considered by the inventors as RJI, which provides solutions to top-k join queries with guaranteed worst case access time. Every function, f(x), is completely defined by a pair of preference values (p1, p2). The value of the function, f, on a tuple, teDk, with rank values (s1, s2) is equal to p1s1+p2s2. The index structure, RJI, is constructed by representing members of F and rank value pairs for each teDk as vectors in two-dimensional space. Since every f(x) is completely defined by the pair (p1, p2), every function, f, may be depicted by the vector e=(p1, p2) on the plane. Similarly, the rank value pairs may be characterized as a vector s=(s1, s2) on the plane. In light of the preceding geometric relations, the value of a function, f, on a tuple teDk, with rank values (s1, s2) is inner product of the vectors e and s. The reasoning behind representing members of class of monotone linear functions, F, as vectors may be explained as follows. Assume that |e|1=1 (i.e., the vector, e, is a unit vector), then the value of the function, f(x), is the length of the projection of the vector x on the vector e. It should be noted, however, that the assumption that the vector, e, is a unit vector is solely for the purposes of simplifying the presentation. It should not be interpreted as being required for the correctness of the approach of the present invention. The result of any top-k join query Tk(e) is the same independent of the magnitude of the vector, e. For example, letting u-ze be some vector in the direction of e with length α, Tk(α) is the same as Tk(u) since the lengths of the projected vectors change only by a scaling factor, and thus, their relative order is not affected.

As previously depicted, the set of tuples Dk may be represented as points in two-dimensional space using the rank values of each tuple. Given a unit vector e, the angle α(e) of the vector is defined as the angle of e with the axis representing (without loss of generality) the i-th rank value. For a set of i tuples {t1, t2, ..., ti}, Ord_{i}(t1, t2, ..., ti) is defined as the ordering of the tuples {t1, t2, ..., ti} when the rank value pairs associated with each tuple are projected on the vector e and are sorted by non-increasing order of their projection lengths. Ord_{i}(t1, t2, ..., ti) is used to denote the reverse of that ordering. Tk(e) contains the top k tuples in the ordering Ord_{k}(t1, t2, ..., ti).

Let the vector, e, sweep the plane defined by the domains of rank attributes (R×R+). Specifically, let the sweep start from the s1-axis and move towards the s2-axis (i.e., counter-clockwise). Thus, e ranges from e=(0,0) to e=(1,0) to e=(1,0), (1,0). As such, to examine how the ordering Ord_{i}(Dk) varies as e sweeps the plane, two tuples and their relative order are first considered. That is, let s1=(s1', s2') and s2=(s1'', s2'') be the rank value pairs for two tuples t1, t2, teDk. Since rank value pairs are represented as vectors, let (s1', s2')=s1 and denote the vector defined by the difference of s1 and s1' by e. Let t1 denote the angle of the vector (s1', s2') with the s1-axis. Having done so, the inventors disclose postulate four (4), which follows:

Depending on the angle, b, that vector (s1', s2') forms with the s1-axis as e sweeps the plane, one of the following holds true:

(a) if b∈[0, π/2], Ord_{i}(t1, t2)) is the same for all e.
(b) if b∈[π/2, 0] U [π/2, π] let e be the vector perpendicular to (s1', s2') and as such:
(i) f(s1, s2')=f(s1, s2'),
(ii) Ord_{i}(t1, t3)=Ord_{i}(t1, t2), for all vectors e, e2 with a(e1)=a(e2)=a(e3), or a(e1)=a(e2)≠a(e3).
(iii) Ord_{i}(t1, t3)=Ord_{i}(t1, t2), for all e, e2, such that a(e1)=a(e2)≠a(e3). Moreover, as a vector e sweeps the positive quadrant, tuples t1, t2 are adjacent in the ordering Ord_{i}(Dk) immediately after e crosses vector e1, and vector e2, adjacent in Ord_{i}(Dk) immediately after e crosses vector e2.

The principles presented above indicate that as e sweeps a plane, the ordering of tuples t1 and t2 changes only when e crosses the vector e1, which is defined as the vector perpendicular to (s1', s2'). If the vector <s1', s2'> has a positive slope, then the ordering of the tuples t1, t2 remains the same for all e. The vector e1 is considered the separating vector of tuples t1 and t2, and a(e1) is considered the separating point.

FIG. 4a and FIG. 4b depict a graphical representation of the ordering of two tuples for two different values of the angle, b, that the vector (s1', s2') forms with the s1-axis as e sweeps the plane. In FIG. 4a and FIG. 4b two tuples, t1, and t2, are graphed along with a representation of the separating vector, e1, of the tuples, t1 and t2, and a graphical representation of the angle, b. More specifically, FIG. 4a graphically depicts an example of the ordering of two tuples t1, t2, when the vector <s1', s2'> has a positive slope. As evident in FIG. 4a, the ordering of the tuples t1, t2 remains the same for all e. FIG. 4b graphically depicts an example of the ordering of the two tuples t1, t2, for the second case where the vector <s1', s2'> has an other-than-positive slope. Although only two tuples, t1 and t2 are
depicted in FIG. 4a and FIG. 4b, it should be noted that more than two tuples may share the same separating vector. For example, if \( t_1, t_2, \) and \( t_3 \) are three tuples such that their corresponding rank value pairs are co-linear, the three tuples \( t_1, t_2, \) and \( t_3 \) all share the same separating vector. As such, the inventors disclose postulate five (5), which follows:

\[
\text{If } t_1, t_2, \ldots, t_n \text{ are } \ell \text{ tuples with collinear rank value pairs sharing the same separating vector, } e, \text{ then } \forall \alpha(t_1, t_2, \ldots, t_n), \alpha(e(t_1, t_2, \ldots, t_n)) \text{ for all } \alpha(e), (\alpha(e(t_1, t_2, \ldots, t_n))) \text{ such that } \alpha(e(t_1, t_2, \ldots, t_n)) = \alpha(e). \tag{5}
\]

Briefly stated, each separating vector corresponds to the reversal of two or more adjacent points.

FIG. 5 depicts an embodiment of an RJI Construct algorithm in accordance with the present invention, which pre-processes the set of tuples, \( D_p \), and constructs an index on its elements. In the algorithm of FIG. 5, a vector, \( e \), sweeps the plane and the composition of \( T_p(e) \) is monitored. Every time vector \( e \) crosses a separating vector, \( \text{Ord}(D_p) \) changes by swapping two (or more if they are collinear) adjacent tuples as described above. A key observation is that this swap is of interest for indexing purposes only if it causes the composition of \( T_p(e) \) to change. Assuming that \( D_p \) contains tuples of the form \((t_{id}, s_1, s_2)\), where \( t_{id} \) is a tuple identifier, and \( s_1, s_2 \) are the associated rank values, the algorithm of FIG. 5 initiates by first computing the set \( V \) of all separating vectors. This involves considering each pair of tuples in \( D_p \) and computing their separating vector and the associated separating point. Let \( e_{op} = (t_{id}, s_1, s_2) \) represent the separating vector (separating point) for each pair of tuples, \( t_i, t_j, 1 \leq i < j \leq |D_p| \). Each pair \((t_{id}, t_{id})\), along with the associated separating point \( e_{op} \), is computed and materialized as set \( V \). Then set \( V \) is sorted in non-decreasing order of \( a(e_{op}) \).

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 (un-sorted) the \( K \) tuples with the highest score according to the function \( f_e \), where \( e \) is the current position of the sweeping vector. \( R \) is initialized to hold the top-\( K \) tuples with respect to the initial position of vector, \( e \), namely \( e = (0,0) \). \( f(e) \) (function \( f_{11,0}) \). Initializing \( R \) is easy, since the set \( D_p \) computed at the end of the Dominating Set algorithm is sorted by \( s_1 \).

Each \( a(e_{op}) \) in the set \( V \), and the corresponding vector \( e_{op} \) is associated with two tuple identifiers \( (t_i, t_j) \). When \( e \) crosses the vector \( e_{op} \) during the sweep, it causes the ordering of tuples \( t_i, t_j \) to change according to Postulates 4 and 5 depicted above. In case both tuple identifiers belong to \( R \), or neither belongs to \( R \), the vector \( e_{op} \) can be safely discarded from consideration, since it does not affect the composition of \( R \). Otherwise, \( a(e_{op}) \) is determined together with the composition of \( R \), and \( R \) is updated to reflect the new tuple identifiers. The last value of \( R \) is also determined after the sweep is completed. At the end of the RJI Construct algorithm, \( M \) separating vectors, \( e_1, e_2, \ldots, e_M \) have been accumulated (represented by their separating points \( a(e_i) \), \( 1 \leq i \leq M \)). The accumulation of the vectors, \( e_1, e_2, \ldots, e_M \), partitions the quadrant into \( M+1 \) regions. Each region \( i \) is defined by vectors \( e_i, e_{i+1} \), where \( e_i =<0,0,0,0> \), and \( e_{i+1} =<0,0,1,0> \). Region \( i \) is associated with a set of \( K \) points \( R_{D_p,i} \), such that for any vector, \( e, a(e) =<a(e), a(e_{i+1})> \), uniquely identifying a function \( f_{e', t_{id}} \) on \( T_p(e) \) 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 non-increasing order. That is, \( R' \) contains (up to a permutation) the answer to any top-\( K \) query, \( K \leq K \) for any function defined by a vector in region \( i \).

For example, FIG. 6a and FIG. 6b graphically depict an example of the operation of the RJI Construct algorithm. FIG. 6a and FIG. 6b comprise a set, \( D_2 \), consisting of four tuples, \( t_1, t_2, t_3, t_4 \). The RJI Construct algorithm starts by computing the separating vector for each pair of tuples. For ease of explanation and brevity, in FIG. 6a the separating vectors are presented only for pairs of tuples \( t_1, t_2, t_3, t_4 \). The separating vectors \( e_{12}, e_{13}, e_{14}, e_{23}, e_{24}, \) and \( e_{34} \) are computed for each pair as shown in FIG. 6a. Each pair is stored along with the associated separating point and the collection is ordered based on separating points. Setting \( K = 2 \), an index is created answering the top-1 and top-2 join queries.

Consider now a vector, \( e \), sweeping the plane. The first two tuples in \( Ord_{1,0}(D_2) \) are \( R = \{t_1, t_4\} \). The first vector crossed by \( e \) is \( e_{14} \), which corresponds to swapping tuples \( t_1, t_4 \). The swap changes the composition of \( R \). In particular, \( t_4 \) is replaced with \( t_3 \). At this point, \( a(e_{12}) \) is stored along with the \( R = \{t_1, t_3\} \). The current composition of \( R \) becomes \( R = \{t_1, t_3\} \). Then \( a(e_{34}) \) 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 such that \( a(e_{23}) \) is stored along with the \( R = \{t_1, t_2\} \) and the current composition of \( R \) changes to \( R = \{t_1, t_2\} \). When the input is exhausted, the current ordering \( R = \{t_1, t_2, t_3\} \) is stored, and the algorithm terminates. FIG. 6b depicts the final partitioning of the plane.

Critical to the size of the index is the size of \( M \), the number of separating vectors identified by the RJI Construct algorithm. A worst case bound is provided on \( M \) by bounding the number of times that a tuple identifier can move from position \( K + 1 \) to position \( K \) in \( Ord(D_p) \). Postulates 4, 5 previously presented guarantee that whenever a swap happens between elements of \( Ord(D_p) \), it takes place between two adjacent elements in \( Ord(D_p) \). Thus, only the separating vectors that cause a swap of the elements in positions \( K \) and \( K + 1 \) in \( Ord(D_p) \) are indexed, since these are the ones that cause the composition of \( T \) to change. For every \( t \in D_p \), define \( rank(t) \) to be the position of tuple \( t \) in the ordering \( Ord(D_p) \). As such, the inventors disclose postulate six (6), which follows:

\[
\text{For every tuple } t_i \in D_p, \text{ rank}(t) \text{ can change from } i+1 \text{ to } i \text{ at most } i \text{ times for any vector } e, \text{ } \text{ or } K. \tag{6}
\]

In addition, the inventors disclose the following Theorem:

Given a set of dominating points \( D_p \) an index may be constructed for top-\( K \) join queries in time \( O(D_p + \log(D_p)) \) using space \( O(D_p/K^2) \) providing answers to top-k join queries in time \( O(log(KD_p + K \log K), K \leq K \) in the worst case.

Postulate 6 guarantees that each element in \( D_K \) contributes at most \( K \) changes to \( \text{rank}(t) \). 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 \( O(D_p/K) \). After the separating points \( a(e) \) are identified, they are organized along with the associated sets \( R_k \) in a B-tree indexed by \( a(e) \). The leaf level stores pointers to the sets \( R_k \). Thus, the total space requirement becomes \( O(D_p/K^2) \). There are \( O(nK^2) \) elements in \( D_p \) 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 \( O(nK^2) \). The worst case time complexity for constructing the ranked join index is \( O(nK^2 \log(nK^2)) \) time to compute the separating vectors and separating points and \( O(nK^2 \log(nK^2)) \) time to sort the separating points. Constructing a B-tree may be performed during the single scan on the sorted separating point collection of the RJI Construct algorithm.
Thus, the total construction time is $O(n^2 K^2 \log(n^2 K^2))$. It should be noted that these are the worst case space and construction time requirements for the index R. At query time, given the vector $e$, that defines a function, $f \in F$, a ($a(e)$) is computed and the R-tree is searched using ($a(e)$) as a key. This effectively identifies the region that contains the vector $e$. Then, the associated set $R_1$ is retrieved and $f_0$ evaluated for all elements of $R_1$, sorting the results to produce $T_0(e)$. Thus, the query time is $O(\log(n^2 K)^2 K \log K)$ in the worst case, for any top-k join query, $k \leq K$.

The ranked join index design of the present invention 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, the space requirements could be decreased significantly, at almost no expense on query time. Note that sets $R_1$ and $R_{m-1}$, associated with two neighboring regions differ, in the worst case, by only one tuple. Therefore, the set $R_1 U R_{m-1}$ contains $K + 1$ distinct tuples. If $m$ regions are merged, then the resulting region contains at most $K + m - 1$ distinct tuples. It should be noted that this is a worst case bound. Depending on the distribution, a region may contain less than $K + m - 1$ distinct tuples. Therefore, if there are initially $M$ 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 $O(\log(M/m) + (K + m) \log(K + m))$. Since $M - O(n^2)$ in the worst case, the requirements of the index is $O(n^2(K + m)/m)$ for space, and $O(\log(n^2/m) + (K + m) \log(K + m))$ for query time.

For example, Figs. 7a, 7b and 7c graphically depict an example of the space-time tradeoffs of the RJI Construct algorithm for $K = 2$. Every two regions of Fig. 7a are merged and the result is depicted in Fig. 7b. Merging $m$ regions does not always result in a region with $K + m - 1$ tuples as described above. Depending on the distribution of the rank values, it may be the case that as the vectors that define the $m$ regions are crossed, 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. As such, instead of merging every $m$ regions, the regions may be merged 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. If fast query time is the main concern, the query time may be reduced by storing all separating vectors that cause $T_0(e)$ to change. According to Postulate 6 described above, a tuple may move from position 1 to 1 at most 1 times, therefore, each tuple may contribute at most $1 + 2 + \ldots + K - K(\text{size of each tuple})$ changes to $T_0(e)$. Thus, storing at most $O(n^2 K)$ separating vectors the query time may be reduced to $O(\log(n^2 K))$. Effectively in this case an ordered sequence of points is being stored in each region $R$, so there is no need for evaluating $f_0$ on the elements of the region. The ordered sequence (according to $f_0$) may be returned immediately. Fig. 7c depicts a materialization of the separating points causing a change in ordering for the tuples in each region of Fig. 7a.

The inventors further disclose herein a variant of a 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 a solution provided by the present invention. Briefly stated, an R-tree index is implemented to prune away a large fraction of the tuples that are bound not to be among the top $k$. This modified R-tree is referred to by the inventors as the TopKtree. Consider the two-dimensional space defined by the 2 rank values associated with each tuple in $D_X$ returned by the Dominating Set algorithm. An R-tree on these points is constructed using R-tree construction algorithms know in the art. A basic observation is that due to the monotonicity property of the functions $f_0, f_1, \ldots, f_k$, 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 $f_0$ gets at the lower left and upper right corners of $r$. Following this observation the R-tree search procedure is modified according to the following.

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.

For example, Fig. 8a and Fig. 8b graphically depict an embodiment of an R-tree with three MBRs, namely $r_1, r_2$, and $r_3$, and a top-$k$ join query with $e = (p_1, p_2)$. 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 $e$. This projection is referred to by the inventors as the maximum-projection for the MBR, and the MBR that has the largest maximum-projection among all of 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 embodiment of the algorithm, named TopKtree Answer, is presented in Fig. 9. For brevity, it is assumed that each MBR contains at least $K$ tuples. Therefore, the algorithm guiding the search uses only the master MBR at each R-tree level. Accounting for the case where multiple MBR’s are required 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. In the TopKtree Answer algorithm of Fig. 9, 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 Fig. 8a, since its maximum-projection $r_1^h$ is the largest among the three MBRs. In this case, all MBRs with maximum-projection less than the minimum-projection of the master MBR may be safely pruned away. In this example the tuples in $r_1$ will not be examined 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, all of the MBRs whose projections on vector $e$ overlap with the projection of the master MBR must be examined. It should be noted, however, that there are many cases in which the TopKtree accesses more MBRs than really necessary. For example, Fig. 8b, depicts 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 $r_2$ has the largest maximum-projection (e.g., $r_2^h$) none of its tuples (e.g., $t_1$) are contained in the top-2 answer. Thus, all the computations involving $r_2$ are useless in this case.

Fig. 10 depicts a high level block diagram of an embodiment of the controller suitable for performing the methods (i.e., algorithms) of the present invention. The controller 1000 of Fig. 10 comprises a processor 1010 as well as a memory 1020 for storing the algorithms and programs of the present invention. The processor 1010 cooperates with conventional support circuitry 1030 such as power supplies, clock circuits, cache memory and the like as well as circuits that assist in
executing the software routines stored in the memory 1020. As such, it is contemplated that some of the process steps discussed herein as software processes may be implemented within hardware, for example, as circuitry that cooperates with the processor 1010 to perform various steps. The controller 1000 also contains input-output circuitry 1040 that forms an interface between the various functional elements communicating with the controller 1000.

Although the controller 1000 of FIG. 10 is depicted as a general purpose computer that is programmed to perform various methods and operations in accordance with the present invention, the invention may be implemented in hardware, for example, as an application specified integrated circuit (ASIC). As such, the process steps described herein are intended to be broadly interpreted as being equivalently performed by software, hardware, or a combination thereof.

While the foregoing is directed to various embodiments of the present invention, other and further embodiments of the invention may be devised without departing from the basic scope thereof. As such, the appropriate scope of the invention is to be determined according to the claims, which follow.

What is claimed is:

1. A method of creating a ranked join index for ordered data entries, comprising:
   determining, via a processor, a dominating set of said ordered data entries;
   mapping said dominating set of said ordered data entries according to rank attributes;
   determining a separating vector for each set of adjacent mapped data entries; and
   ordering said data entries according to a separating point associated with each of said separating vectors.

2. The method of claim 1, wherein said determining the dominating set of said ordered data entries, comprises:
   maintaining a priority queue of a predetermined size of said ordered data entries according to said rank attributes, wherein data entries having highest combined rank attribute values are maintained in said priority queue;
   wherein, if said priority queue has reached a maximum capacity, only data entries having combined rank attribute values greater than an attribute value of data with a minimum rank value present in the priority queue are added to the priority queue.

3. The method of claim 2, wherein said predetermined size corresponds to a minimum number of data entries necessary to generate, a ranked join index providing answers with a desired guaranteed performance on any top-k join query.

4. The method of claim 1, wherein said ordering said data entries, comprises:
   sweeping a vector across a plane of said mapped data entries, wherein each time said vector crosses a separating vector, a current composition of highest ranked data entries changes by swapping at least one of the data entries in the adjacent data entries set if it causes a change in the index; and
   wherein each time a data entry is swapped, the highest ranked data entries are materialized and a new index entry is initiated.

5. The method of claim 1, wherein said ordering is query independent such that substantially any user preference query is resolved using said index.

6. The method of claim 1, wherein each set of adjacent mapped data entries comprises more than two mapped data points if data entries are collinear.

7. The method of claim 1, wherein performance guarantees are provided for an amount of time said index requires for resolving user queries.

8. The method of claim 1, further comprising merging ordered data entries.

9. The method of claim 8, wherein said merging results in a storage space requirement for said index that is characterized according to an equation:
   \[ O(nK^2(K+m)/m) \]
   wherein \( n \) represents a total number of data entries to be indexed, \( K \) represents an upper bound on a number of high ranking data entries that is requested by a user, and \( m \) represents a total number of data entries to be merged.

10. The method of claim 8, wherein said merging results in a query time for said index that is characterized according to an equation:
    \[ O((nK^2/m)+(K+m)\log(K+m)) \]
    wherein \( n \) represents a total number of data entries to be indexed, \( K \) represents an upper bound on a number of high ranking data entries that is requested by a user, and \( m \) represents a total number of data entries to be merged.

11. The method of claim 8, wherein said merging said ordered data entries provides a space and time tradeoff.

12. The method of claim 11, wherein said space and time tradeoff comprises reducing a query time of said index by increasing a storage space of said index.

13. The method of claim 11, wherein said space and time tradeoff comprises reducing a storage space required by said index by increasing a query time of said index.

14. A method of providing solutions to top-k join queries of ranked data entries for user specified preferences, comprising:
   determining, via a processor, a dominating set of said ranked data entries;
   mapping said dominating set of said ranked data entries according to rank attributes;
   creating a ranked join index for said ranked data entries by determining a separating vector for each set of adjacent mapped data entries; and
   ordering said data entries according to a separating point associated with each of said separating vectors; and
   providing a solution for a user preference query using said ranked join index.

15. A computer-readable medium storing a set of instructions, which when executed by a processor, perform a method of creating a ranked join index for ordered data entries, comprising:
   determining a dominating set of said ordered data entries;
   mapping said dominating set of said ordered data entries according to rank attributes;
   determining a separating vector for each set of adjacent mapped data entries; and
   ordering said data entries according to a separating point associated with each of said separating vectors.

16. A computer-readable medium storing a set of instructions, which when executed by a processor, perform a method of providing solutions to top-k join queries of ranked data entries for user specified preferences, comprising:
   determining a dominating set of said ranked data entries;
   mapping said dominating set of said ranked data entries according to rank attributes;
   creating a ranked join index for said ranked data entries by determining a separating vector for each set of adjacent mapped data entries;
ordering said data entries according to a separating point associated with each of said separating vectors; and providing a solution for a user preference query using said ranked join index.

17. An apparatus, comprising a memory storing information and program instructions and a processor executing said instructions, said apparatus adapted to perform a method of creating a ranked join index for ordered data entries comprising:

determining a dominating set of said ordered data entries;

mapping said dominating set of said ordered data entries according to rank attributes;

determining a separating vector for each set of adjacent mapped data entries; and

ordering said data entries according to a separating point associated with each of said separating vectors.

18. The apparatus of claim 17, wherein said determining the dominating set of said ordered data entries, comprises:

maintaining a priority queue of a predetermined size of said ordered data entries according to said rank attributes, wherein data entries having highest combined rank attribute values are maintained in said priority queue;

wherein, if said priority queue has reached a maximum capacity, only data entries having combined rank attribute values greater than an attribute value of data entries with a minimum rank value present in the priority queue are added to the priority queue.

19. The apparatus of claim 18, wherein the predetermined size of said priority queue corresponds to a minimum number of data entries necessary to generate, a ranked join index providing answers with a desired guaranteed performance on any top-k join query.