JTop Algorithms for Top-k Join Queries

Reza Akbarinia[1], Ihab F. Ilyas[1], M. Tamer Özsu[1], Patrick Valduriez [2]

[1] David R. Cheriton School of Computer Science,
University of Waterloo
[2] INRIA and LINA, University of Nantes, France

Technical Report CS-2008-03

February 2008

Top-k join queries have become very important in many important areas of computing. One of the most efficient algorithms for top-k join queries is the Rank-Join algorithm [17][18]. However, there are many cases where Rank-Join does much unnecessary access to the input data sources. In this report, we first show that there are many cases where Rank-Join's stopping mechanism is not efficient, and it does much unnecessary accesses to the input data sources. Then, we propose JTop, a family of much more efficient algorithms for top-k queries. We prove that our algorithms always perform less work than Rank-Join, and thus are more efficient. We also show that the performance of our algorithms can be *O(n)* times better than that of Rank-Join where *n* is the number of data items in the database. We evaluated the performance of our algorithms through experimentation over databases with different distributions. The results show that over the tested databases our algorithms significantly outperform Rank-Join.

1. INTRODUCTION

Top-k queries have attracted much interest in many different areas of computing such as network and system monitoring [5][21], information retrieval [4][20][22], sensor networks [29][31], multimedia databases [1][10][16][27], spatial data analysis [33][19], probabilistic databases [28], data stream management systems [23][25], etc. The main reason is that they avoid overwhelming the user with large numbers of uninteresting answers.

The two main forms of top-k queries are top-k selection and top-k join. The seminal work by Fagin [13] on top-k selection queries (or top-k queries for short) proposes a general model for answering top-k queries as follows. Suppose we have *m* lists of *n* data items such that each data item has a local score in each list. The lists are sorted according to the local scores of their data items. Each data item has an overall score, which is computed based on its local scores in all lists using a given scoring function. The problem is to find the *k* data items with the highest overall scores. Most of the top-k algorithms work as follows. They continually read the data items in the lists starting from the head, and stop when a specific condition on the seen data items holds. To decide when to stop accessing the lists, each top-k algorithm has a *stopping mechanism*. Fagin [14] proposed a simple, yet efficient algorithm called the Threshold Algorithm (TA) [14][16][27]. Its stopping mechanism (based on a threshold) has been the basis for several TA-style algorithms in distributed environments, *e.g.* [3] [5][12][24]. Best Position Algorithms (BPA and BPA2) [2] which were recently proposed significantly outperform TA.

Top-k join queries allow users to join multiple inputs and report the top-k join results. The answer to a top-k join query is a set of join results ordered according to the scoring function. There has been work on top-k join queries with different applications and with different assumptions, *e.g.* [17][26][30]. To our knowledge, the most efficient algorithm for top-k join queries is the Rank-Join algorithm [17][18]. Rank-join continually reads the ranked inputs and generates valid join data among the seen data until its stopping condition decides to stop. The stopping mechanism of Rank-Join allows it to work on join queries with ad-hoc join conditions. However, it does not take advantage of either specific information on the join attribute values or the characteristics of the query. As we will show later, there are many cases where Rank-Join's stopping mechanism becomes lazy, *i.e.* it stops too late, which causes many unnecessary accesses in the inputs.

In this report, we propose a family of new algorithms for processing top-k join queries. They have efficient stopping mechanisms that make them much more efficient than Rank-Join, so their performance can be *O(n)* times better than that of Rank-Join. Our main contributions are summarized as follows:

- First, we propose SR_JTop, a new top-k join algorithm which takes advantage of both sorted and random accesses as well as information on the join condition. We prove that SR_JTop always stops at a position which is lower than or equal to that of Rank-Join, thereby doing less work. We also show that there is a class of databases over which the performance of SR_JTop is *O(n)* times better than that of Rank-Join where *n* is the number of data items of the database.
- Second, we propose a new algorithm, called BP_JTop, which is designed for systems with "position-based indexing", *i.e.* after accessing a data item in an index, we can know its position in the index. BP_JTop takes into account the position of seen data items, and has a stopping mechanism which is more efficient than that of SR_JTop. We show that BP_JTop always stops at a position which is lower than or equal to that of SR_JTop, and there are databases over which BP_JTop stops at a position which is *O(m)* times lower than that of SR_JTop where *m* is the number of scoring attributes, *i.e.* those which are used in the scoring function.
- Third, we propose two new algorithms, LR_JTop and NR_JTop, for systems where random accesses are expensive or not supported, respectively. LR_JTop does a very limited number of random accesses to the data sources, *i.e.* only for a set of final join data items. NR_JTop does only sorted access to the data sources. We show that LR_JTop and NR_JTop always stop at positions which are lower than or equal to that of Rank-Join. We also show that their performance can be *O(n)* times better than Rank-Join.
- Our extensive experimental study under different data distributions shows that our algorithms yield high performance gains against the Rank-Join algorithm.

The rest of this report is organized as follows. In Section 2, we give the problem definition. Section 3 introduces Rank-Join, the best algorithm proposed so far. In Section 4, we describe the SR_JTop algorithm. In Section 5, we describe the BP_JTop algorithm. In Section 6, we describe the LR_JTop and NR_JTop algorithms. Section 7 gives a performance evaluation of our algorithms. In Section 8, we discuss related work. Section 9 concludes.

1. PROBLEM DEFINITION

In this report, we address the problem of top-k join query processing. For simplicity, we assume that the query involves only two sources, *e.g.* two relations. However, our algorithms can be easily extended to process multi-way join queries. We also assume equi-joins. Like almost all previous top-k join algorithms, *e.g.* [17][26], we assume that the scoring function is monotonic.

A possible SQL-like notation for expressing the top-*k* join queries that we address is as follows:

SELECT *Select Expression*
FROM R_1, R_2
WHERE $R_1.a_i = R_2.b_j$
ORDER BY $f(R_1.a_1, R_1.a_2, \ldots, R_1.a_{m1}, R_2.b_1, R_2.b_2, \ldots, R_2.b_{m2})$
STOP AFTER k

In this query, f is a monotonic scoring function over m_1 attributes from R_1 and m_2 attributes from R_2. The attributes, which are used in the scoring function, are called *scoring attributes*. In the above query, the attributes $a_1, \ldots, a_{m1}$ and $b_1, \ldots, b_{m2}$ are the scoring attributes.

Let us now model each joining source and its indices with a general model as follows. We model each relation R as a data source $DS(D, L, n, m)$ such that D is a set of n data items (tuples of R), and $L=\{ l_1, l_2, \ldots, l_m \}$ be a set of m lists, where each list corresponds to an attribute in R. Each list l_i contains n pairs of the form (d, s), where $d \in D$ and s is a non-negative real number that denotes the *local score* of d in l_i. Any data item $d \in D$ appears once and only once in each list. Each list l_i is sorted in descending order of its local scores, hence called "sorted list". Let $sc(d, l_i)$ denote the local score of a data item d in a sorted list l_i. A set of two or more data sources is called a *database*.

Now, we define the join operation as follows. Consider two data sources $DS_1(D_1, L_1, n_1, m_1)$ and $DS_2(D_2, L_2, n_2, m_2)$. Given two lists $l \in L_1$ and $l' \in L_2$, called join lists[1], then the join of DS_1 and DS_2, denoted as $Join(DS_1, DS_2)$, is defined as the set of pairs (d, d') such that $d \in D_1$ and $d' \in D_2$ and $sc(d, l) = sc(d', l')$. Each pair (d, d') is called a join data item. For each join data item (d, d'), we compute an *overall score*, denoted by $ov(d, d')$ as follows. Given a scoring function f, $ov(d, d')$ is computed as $f(s_1, s_2, .., s_{m1}, s'_1, s'_2, \ldots, s'_{m2})$ such that $s_u = sc(d, l_u)$ for $1 \leq u \leq m_1$ and $s'_v = sc(d, l_v)$ for $1 \leq v \leq m_2$. In other words, for computing the overall score of (d, d'), we apply the scoring function on the local scores of d and d'.

As defined in [15], we consider two modes of access to the sorted lists. The first mode is *sorted (or sequential) access* by which we access the next data item in the sorted list. Sorted access begins by accessing the first data item of the list. The second mode of access is *random access* by which we lookup a given data item in the list. In this report, we assume that sorted access is available on each sorted list, *i.e.* including join lists. We also need random accesses for most of our algorithms, but not all of them.

Let us now state the problem. Given two data sources $DS_1(D_1, L_1, n_1, m_1)$ and $DS_2(D_2, L_2, n_2, m_2)$, two join lists $l_i \in L_1$ and $l' \in L_2$, and a scoring function f. Let J be the set of couples (d, d') involved in $Join(DS_1, DS_2)$. Our goal is to find a set $J' \subseteq J$ such that $|J'| = k$, and $\forall (d_1, d'_1) \in J'$ and $\forall (d_2, d'_2) \in (J - J')$ the overall score of (d_1, d'_1) is at least the overall score of (d_2, d'_2).

[1] In this model, we assume that the join attribute is a scoring attribute, *i.e.* it is used in the scoring function. However, if it is not a scoring attribute, we can simply assume it is, but with no impact on the scoring function, *e.g.* it has a zero coefficient in the scoring function.

2. RANK-JOIN

Rank-Join [17] is the best algorithm proposed so far for processing top-k join queries in relational databases. In this section, we introduce this algorithm which is useful for comparison with our algorithms.

Rank-Join's main assumptions are: the scoring function is monotonic; it works on relational databases and assumes that the tuples of each input relation are ranked according to the scoring function. If each input relation has only one scoring attribute (*i.e.* one of its attributes is used in the scoring function), then a sorted list (*e.g.*, an index on that attribute) gives a ranked list of tuples. But, if there are two or more scoring attributes (say $u > 1$) for an input relation R, then none of the sorted lists ranks R's tuples according to the scoring function, because more than one attribute of R influences the scoring function. In this case, Rank-Join fragments[1] R vertically into u relations, *i.e.* one relation per scoring attribute, and considers a join operation between any two fragments. Then, it treats each of these u fragments as an input relation which is ranked according to the scoring function.

Let us now briefly describe the Rank-Join algorithm. It continually does sorted access to the sorted lists in parallel, and for each new seen tuple in a list, produces valid join combinations with all seen tuples seen so far in the other lists. It stops if there are at least k join tuples whose overall scores are higher than or equal to a *threshold* which is computed as follows. Let m be the number of input lists, *i.e.* $m=m_1+m_2$. For each list l_i ($1 \le i \le m$), let h_i be the top (*i.e.* first) local score in l_i, and c_i be the last local score which is seen in l_i under sorted access. Let f be the scoring function, then the threshold of Rank-Join is the maximum of m values as follows:

$$\begin{aligned} T_{RJ} = Max \{ & f(c_1, h_2, \ldots, h_m), \\ & f(h_1, c_2, h_3, \ldots, h_m), \\ & f(h_1, h_2, c_3, h_4, \ldots, h_m), \\ & \ldots, \\ & f(h_1, h_2, h_3, \ldots, h_{m-1}, c_m) \} \end{aligned}$$

In other words, T_{RJ} is the maximum of m values such that each value is computed by applying the scoring function on the last seen score from one list and the first seen scores from the other lists.

With this threshold, Rank-Join works correctly (see the proof in [18]). However, there are many cases where the threshold decreases very slowly. Thus, the algorithm needs to go deep in the lists before the threshold becomes less than or equal to the overall score of k join tuples. In these cases, we say that the threshold is lazy, *i.e.* it moves very slowly. In the following, we define the problem of lazy threshold, and show that, if only one of the scoring attributes has a low impact on the scoring function then Rank-Join suffers from the lazy threshold problem. Let $s_i^{(p)}$ be the local score which is at position p in list l_i, *i.e.* the pth score seen in l_i. Without loss of generality, assume that, at each step, the last seen data items in all lists are at the same position. Let τ_p be the value of threshold at position p. Thus, τ_1 is the value of threshold at the first position. Also let $\delta \ge 0$ be a very small default real number. Then lazy threshold is defined as follows.

Definition 1: Lazy threshold. *A threshold τ is lazy if $\tau_1 - \tau_p \le (p-1)*\delta$ for each $p \ge 2$.*

In other words, at each step the decrease in the value of a lazy threshold is at most δ. If the threshold of an algorithm is lazy, then the algorithm may stop very late or after accessing all data items of all lists. We show that in the cases where at least one of the scoring attributes has a low progressive impact on the scoring function, Rank-Join' threshold mechanism is lazy. Let us formally define the low progressive impact of a scoring attribute.

Definition 2: Low progressive impact. *An attribute i has a low progressive impact on the scoring function f if for each position p we have $f(x_1, x_2, \ldots, x_{i-1}, s_i^{(p-1)}, x_{+-1}, \ldots, x_m) - f(x_1, x_2, \ldots, x_{i-1}, s_i^{(p)}, x_{+-1}, \ldots, x_m) \le \varepsilon$ for each $x_j \ge 0$, where $0 \le \varepsilon$ is a very small default real number.*

[1] Rank-Join could also pre-compute the total score per tuple in each relation, and work on it. However, this is very costly because for each scoring function, it has to do this pre-computation for all tuples.

In other words, a scoring attribute has a low progressive impact on the scoring attribute if at each step, its contribution to the reduction of the scoring function is at most ε. One case where an attribute has a low progressive impact is when all values of the attribute are equal or very close to each other. Another case is where the coefficient of the attribute in the scoring function is very small, *e.g.* close to zero.

The following lemma shows that the threshold of the Rank-Join algorithm is lazy if there is at least one scoring attribute whose progressive impact is low with $\varepsilon \leq \delta$.

Lemma 1. *If in the inputs of Rank-Join, there is at least one scoring attribute whose progressive impact on the scoring attribute is low with $\varepsilon \leq \delta$, then Rank-Join's threshold is lazy.*

Proof. Without loss of generality, assume the low progressive impact scoring attribute is the first attribute of the scoring function. Let l_1 be the sorted list corresponding to the first attribute of the scoring function, and $s_1^{(p)}$ be the local score which is at position p in l_1. Let τ_1 and τ_p be the value of T_{RJ} (*i.e.* Rank-Join's threshold) at positions 1 and p respectively. Using the definition of T_{RJ}, we have $\tau_p \geq f(s_1^{(p)}, h_2, \ldots, h_m)$. Using the monotonicity of the scoring function and the definition of T_{RJ}, we have $\tau_1 \leq f(h_1, h_2, \ldots, h_m)$. Thus, we have:

$$\tau_1 - \tau_p \leq f(h_1, h_2, \ldots, h_m) - f(s_1^{(p)}, h_2, \ldots, h_m) \qquad (1)$$

Since the first scoring attribute has a low progressive impact on the scoring function, we have the following inequalities:

$$f(s_1^{(1)}, h_2, \ldots, h_m) - f(s_1^{(2)}, h_2, \ldots, h_m) \leq \varepsilon$$

$$f(s_1^{(2)}, h_2, \ldots, h_m) - f(s_1^{(3)}, h_2, \ldots, h_m) \leq \varepsilon$$

$$\ldots$$

$$f(s_1^{(p-1)}, h_2, \ldots, h_m) - f(s_1^{(p)}, h_2, \ldots, h_m) \leq \varepsilon$$

Thus, we have $f(s_1^{(1)}, h_2, \ldots, h_m) - f(s_1^{(p)}, h_2, \ldots, h_m) \leq (p-1)*\varepsilon$. Since $s_1^{(1)} = h_1$, we have $f(h_1, h_2, \ldots, h_m) - f(s_1^{(p)}, h_2, \ldots, h_m) \leq (p-1)*\varepsilon$. By comparing this equation with equation 1, we have $\tau_1 - \tau_p \leq (p-1)*\varepsilon$. Thus, for $\varepsilon \leq \delta$, we have $\tau_1 - \tau_p \leq (p-1)*\delta$. □

As we show in the next sections, the performance of our JTop algorithms can be $O(n)$ times better than Rank-Join where n is the number of data items in the database. Unlike Rank-Join, the cases, where the threshold of JTop algorithms is lazy, are very restricted. For example, the threshold of SR_JTop and BP_JTop may be lazy only if all the scoring attributes of one of the data sources have a low progressive impact on the scoring function.

The main differences between the requirements of Rank-Join and those of JTop algorithms are as follows. Rank-join assumes a general "black box" join condition, and requires no sorted access on the join attribute. JTop algorithms work on equi-join queries and need sorted access to be available on the join attribute. In addition, some of the JTop algorithms take advantage of random accesses, thus they need random access to be available on the scoring attributes. But, Rank-Join does not require the availability of random access.

3. SR_JTop

In this section, we propose SR_JTop, a new algorithm for efficient top- join query processing. It does both random and sorted access to the lists. It is designed for systems where random data access has very low cost when it is done just after a sorted access to the data. As an example of these systems, we can mention database systems in which from each index entry there is a pointer to the whole tuple. Thus, when the value of an attribute of a tuple is seen in the index built on that attribute, *i.e.* via sorted access, then accessing the values of the other attributes of the tuple is done easily, *i.e.* just by one additional I/O.

In the rest of this section, we first propose the SR_JTop algorithm. Then, we prove its correctness, analyze its threshold and compare its performance against Rank-Join.

3.1 Algorithm

Let $DS_1(D_1, L_1, n_1, m_1)$ and $DS_2(D_2, L_2, n_2, m_2)$ be two given input data sources, and f be a given scoring function. Before describing our algorithm, let us define the *overall score of a single data item*. Let $d \in D_1$ be a data item, and $s_1, s_2, \ldots, s_{m1}$ be its local scores in the sorted lists of DS_1, then the overall score of d is computed as $ov(d) = f(s_1, s_2, \ldots, s_{m1}, 0, 0, \ldots, 0)$. In other words, the overall score of a single data item is computed by applying the scoring function on its local scores while putting zero (we assume that the scores are positive numbers) for the scores of the other side. In a similar way, for a data item $d' \in D_2$ with local scores $s'_1, s'_2, \ldots, s'_{m2}$, the overall score is computed as $ov(d') = f(0, 0, .., 0, s'_1, s'_2, \ldots, s'_{m2})$.

The SR_JTop algorithm works as follows.

1. Do continually sorted access in parallel to each list l in DS_1 or DS_2. As a data item d is seen under sorted access in a list l, do random access to read all local scores of d. Maintain the local scores of the seen data items.
2. Produce new valid join combinations of d with all data items seen in the opposite data source so far, and compute the overall score of the new join data items (if any). Maintain in a set Y the k produced join data items whose overall scores are the highest among all join data items produced so far.
3. Choose bdj_1 (called *best data for join in* DS_1) and bdj_2 (called *best data for join in* DS_2) as follows:

   bdj_i (for i=1,2): Let S_i be the set of data items seen in DS_i such that the join attribute value of each $d \in S_i$ is lower than or equal to the last local score which is seen under sorted access in the join list of the opposite data source. Then, $bdj_i \in S_i$ is the data item whose overall score is the highest among all data involved in S_i, *i.e.* $ov(bdj_i) \geq ov(d)$ for each $d \in S_i$. If $S_i = \{\}$ then let bdj_i be a virtual data whose local scores are equal to the last local scores seen in the lists of DS_i.
4. Stop when there are at least k produced joint data items whose overall scores are at least the SR_JTop threshold computed as follows. Let $(c_1, c_2, \ldots, c_{m1})$ and $(c'_1, c'_2, \ldots, c'_{m2})$ be the last local scores seen under sorted access in the lists of DS_1 and DS_2 respectively. Let $(e_1, e_2, \ldots, e_{m1})$ and $(e'_1, e'_2, \ldots, e'_{m2})$ be the local scores of bdj_1 and bdj_2 respectively. Then the threshold of SR_JTop, denoted by T_{SR_JT}, is the maximum of three values as follows:

$$T_{SR\_JT} = Max \{f(c_1, c_2, \ldots, c_{m1}, c'_1, c'_2, \ldots, c'_{m2}),$$
$$f(c_1, c_2, \ldots, c_{m1}, e'_1, e'_2, \ldots, e'_{m2}),$$
$$f(e_1, e_2, \ldots, e_{m1}, c'_1, c'_2, \ldots, c'_{m2})\}$$

5. Return Y to the user.

The threshold of SR_JTop is designed based on the fact that any unproduced join data *(d, d')*, *i.e.* a join data which has not been produced by the algorithm, has at least one unseen element, *i.e.* d or d' or both. If only one of its elements is unseen, *e.g.* d', then the highest overall for the unproduced join data *(d, d')* is when d is a special seen data which is called best data for join in DS_1. Otherwise, *i.e.* if both d are d' are unseen, then their local score in any list is lower than or equal to the last local score seen in the list under sorted access.

Let us illustrate SR_JTop with the following example.

Example 1. Consider the two data sources shown in Figure 1. The join lists are l_2 from DS_1 and l'_1 from DS_2. Assume a top-3 query Q, *i.e.* $k=3$, and suppose the scoring function is $f(x_1, x_2, x'_1, x'_2) = x_1 + x_2 + x'_1 + x'_2$, *i.e.* it computes the sum of the local scores of the data item in all lists. Let us apply SR_JTop on this example. At position 1, the set of seen data items in DS_1 is $\{d_1, d_2\}$. The join attribute values of d_1 and d_2, *i.e.* 99 and 97 respectively, are lower than or equal to the local score which is at position 1 in the join list of DS_2, *i.e.* 99, thus we have $S_1 = \{d_1, d_2\}$. Since the overall score of d_2 is higher than that of d_1, we have $bdj_1 = d_2$. In DS_2, we have $S_2 = \{d'_1\}$, thus $bdj_2 = d'_1$. At this position, there is only one produced join data item, *i.e.* (d_1, d'_1), and its overall score is $ov(d_1, d'_1) = 399$. At position 2, there is one new seen data item in DS_1, *i.e.* d_3, and two new data items in DS_2, *i.e.* d'_2 and d'_3. At this position, the seen data items in DS_1 are $\{d_1, d_2, d_3\}$, but only the join attribute value of d_3, *i.e.* 96, is lower than or equal to the local score which is at 2nd position in the join list of DS_2, *i.e.* 96. Thus we have $S_1 = \{d_3\}$, and $bdj_1 = d_3$. In DS_2, we have $S_2 = \{d'_2, d'_3\}$, and since the overall score of d'_2 and d'_3 is the same, *i.e.* equal to 196, we can choose one of them

Position	DS_1		DS_2	
	l_1	l_2	l'_1	l'_2
1	$d_2 : 101$	$d_1 : 99$	$d'_1 : 99$	$d'_1 : 100$
2	$d_3 : 98$	$d_2 : 97$	$d'_3 : 96$	$d'_2 : 100$
3	$d_1 : 97$	$d_3 : 96$	$d'_2 : 96$	$d'_3 : 100$
4	$d_4 : x_4$	$d_6 : x_6$	$d'_5 : x'_5$	$d'_4 : 100$
5	…	…	…	…
…	$d_i : x_i$	$d_p : x_p$	$d'_j : x'_j$	$d'_q : 100$
…	…	…		
n	…	…	…	$d'_n : 100$

Figure 1. Example database. There are two data sources each one with two sorted lists. The join lists are l_2 and l'_1.

as bdj_2. For example we choose $bdj_2 = d'_2$. At this position, there are two new join data items (d_3, d'_2) and (d_3, d'_3) with overall scores $ov(d_3, d'_2) = ov(d_3, d'_3) = 390$. Let us now compute the threshold of SR_JTop for 2nd position. The last local scores which are seen under sorted access in DS_1 and DS_2 are (c_1, c_2) = (98, 97) and (c'_1, c'_2) = (96, 100) respectively. Since $bdj_1 = d_3$ and $bdj_2 = d'_2$, we have (e_1, e_2) = (98, 96) and (e'_1, e'_2) = (96, 100). Therefore, the threshold of SR_JTop at 2nd position is computed as T_{SR_JT} = max {f(98, 97,96, 100)), f(98, 97, 96, 100), f(98, 96, 96, 100)} = 391. Since at 2nd position there are not k produced join data items with overall scores higher than or equal to T_{SR_JT}, the algorithm does not stop at this position. At position 3, there is no new seen data item in DS_1 and not in DS_2. There is no modification in bdj_1 and bdj_2, *i.e.* $bdj_1 = d_3$ and $bdj_2 = d'_2$. Thus, we have (e_1, e_2) = (98, 96) and (e'_1, e'_2) = (96, 100). For the last seen local scores in DS_1 and DS_2 we have (c_1, c_2) = (97, 96) and (c'_1, c'_2) = (96, 100). Therefore, the threshold is computed as T_{SR_JT} = max {f(97, 96, 96, 100)), f(97, 96, 96, 100), f(98, 96, 96, 100)} = 390. Since we have 3 join data items whose overall scores are at least T_{SR_JT}, i.e. (d_1, d'_1), (d_3, d'_2) and (d_3, d'_3), the algorithm stops at position 3.

If we apply Rank-Join on this example, at position 1, the threshold of rank-join is $T_{RJ} = f$(101, 99, 99, 100) = 399. For the other positions, Rank-Join has the same threshold value, because in each position, one of the m values, which are used in computing the threshold, is obtained by applying the scoring function on the last local score seen in l'_2 and the first local scores in the other lists. Since the local score at any position of l'_2 is equal to 100, the Rank-Join's threshold at any position is equal to $T_{RJ} = = f$(101, 99, 99, 100) = 399. In this database, there is only one join data item whose overall score is higher or equal to 399, *i.e.* (d'_1, d'_1). The overall score of any other join data item is less than 399, *i.e.* this can be seen by regarding the fact that d_1 and d'_1 (and also d_2) can not take part in any other join data item. Therefore, there is not $k=3$ join data items with an overall score higher than or equal to the threshold of Rank-Join in any position. Hence, it does not stop before the nth position, *i.e.* it reads all the n local scores in all lists[1].

3.2 Correctness and Analysis

The following theorem provides the correctness of our SR_JTop algorithm.

[1] Notice that over the database of Example 1, if we use a scoring function in which join attribute values have no impact, *e.g.* $f(x_1, x_2, x'_1, x'_2) = x_1 + x'_2$, then we will have a better result for SR_JTop; it stops at 2nd position. However, this makes no difference for Rank-Join, *i.e.* stops at nth position.

Theorem 1. *If the scoring function f is monotonic, then SR_JTop finds correctly the top-k answers.*

Proof. Let T_{SR_JT} be the value of SR_JTop's threshold when it stops. To prove the theorem, it is sufficient to show that every unproduced join data, *i.e.* a valid join data which is not produced by SR_JTop before its end, has an overall score which is less than or equal to T_{SR_JT}. Let *(d, d')* be an unproduced join data. Since *(d, d')* has not been produced by SR_JTop, at least one of its elements, *i.e. d* or *d'*, has not been seen by SR_JTop. Thus, there are three possible cases for *d* and *d'*: 1) None of them are seen by SR_JTop; 2) *d* is seen but not *d'*; 3) *d'* is seen but not *d*. We show that in all these three cases the overall score of *(d, d')* is at most the threshold of SR_JTop, *i.e.* T_{SR_JT}.

Let us start with the first case. Let $(s_1, s_2, \ldots, s_{m1})$ and $(s'_1, s'_2, \ldots, s'_{m2})$ be the local scores of *d* and *d'* respectively. Since *d* and *d'* are not seen by SR_JTop, their local scores in any list are less than the last local scores in the lists. Thus, we have $s_i \leq c_i$ and $s'_j \leq c'_j$ for $1 \leq i \leq m_1$ and $1 \leq j \leq m_2$. Therefore, since the scoring function is monotonic, we have $f(s_1, s_2, \ldots, s_{m1}, s'_1, s'_2, \ldots, s'_{m2}) \leq f(c_1, c_2, \ldots, c_{m1}, c'_1, c'_2, \ldots, c'_{m2})$. Since $T_{SR\_JT} \geq f(c_1, c_2, \ldots, c_{m1}, c'_1, c'_2, \ldots, c'_{m2})$, we have $T_{SR\_JT} \geq f(s_1, s_2, \ldots, s_{m1}, s'_1, s'_2, \ldots, s'_{m2})$, thus the threshold of SR_JTop is greater than or equal to the overall score of *(d, d')*.

In the second case, *d'* is not seen, thus we have $s'_j \leq c'_j$ for $1 \leq j \leq m_2$. The data item *d* is seen by SR_JTop. According to the definition of bdj_1, *i.e.* best data for join in DS_1, the overall score of bdj_1 must be higher than or equal to that of *d*, *i.e.* $ov(d) \leq ov(bdj_1)$, thus we have $f(s_1, s_2, \ldots, s_{m1}, 0, 0, \ldots, 0) \leq f(e_1, e_2, \ldots, e_{m1}, 0, 0, \ldots, 0)$. Since $s'_1 \leq c'_1$ and the scoring function is monotonic, we have $f(s_1, s_2, \ldots, s_{m1}, s'_1, 0, \ldots, 0) \leq f(e_1, e_2, \ldots, e_{m1}, c'_1, 0, \ldots, 0)$, *i.e.* we replace one of the zeros with corresponding values in the inequality $s'_1 \leq c'_1$. By continuing this replacement, we have $f(s_1, s_2, \ldots, s_{m1}, s'_1, s'_2, \ldots, s'_{m2}) \leq f(e_1, e_2, \ldots, e_{m1}, c'_1, c'_2, \ldots, c'_{m2})$. Since $T_{SR\_JT} \geq f(e_1, e_2, \ldots, e_{m1}, c'_1, c'_2, \ldots, c'_{m2})$, we have $T_{SR\_JT} \geq f(s_1, s_2, \ldots, s_{m1}, s'_1, s'_2, \ldots, s'_{m2})$. Thus in the second case like the first one, the threshold of SR_JTop is greater than or equal to the overall score of *(d, d')*. In a similar way, we can do the proof for the third case. Therefore, in all the three cases the overall score of the unproduced join tuple is at most T_{SR_JT}. □

By the following theorem, we compare the stop positions of SR_JTop and Rank-Join.

Theorem 2. *The position, at which SR_JTop stops, is always lower than or equal to that of Rank-Join.*

Proof. Assume Rank-Join stops at position *p*. We show that SR_JTop stops at a position lower than or equal to *p*. For this, it is sufficient to show that the value of SR_JTop's threshold at *p* is less than or equal to that of Rank-Join. Let T_{RJ} and T_{SR_JT} be the threshold value of Rank-Join and SR_JTop respectively. Recall that T_{RJ} is computed as the maximum of $m_1 + m_2$ values such that each value is computed by applying the scoring function on the last seen score from one list and the first seen scores from the other lists. We show that each of the three values used for computing T_{SR_JT} are less than or equal to one of the values used in computing T_{RJ}, thus they are less than or equal to T_{RJ}, and thereby $T_{SR\_JT} \leq T_{RJ}$.

This is done by the following inequalities:

$$f(c_1, c_2, \ldots, c_{m1}, c'_1, c'_2, \ldots, c'_{m2}) \leq f(c_1, h_2, h_2 \ldots, h_{m1+m2})$$

$$f(c_1, c_2, \ldots, c_{m1}, e'_1, e'_2, \ldots, e'_{m2}) \leq f(c_1, h_2, h_2 \ldots, h_{m1+m2})$$

$$f(e_1, e_2, \ldots, e_{m1}, c'_1, c'_2, \ldots, c'_{m2})\} \leq f(h_1, h_2, h_2 \ldots, h_{m1+m2-1}, c'_{m1+m2})$$

These inequalities are implied by using the monotonicity of the scoring function as well as the fact that any local score in the list, *e.g.* c_i or e_i, is lower than or equal to the first local score of the list, *i.e.* h_i. □

Theorem 2 shows that the position at which SR_JTop stops is always lower than or equal to that of Rank-Join. We also make the following observation for the difference between the performance of SR_JTop and Rank-Join.

Observation 1. *There is a class of databases over which the number of accesses to the data sources done by SR_JTop is O(n) times lower than that of Rank-Join where n is the number of data items.*

Proof. An example of these databases is that of Figure 1. If we apply SR_JTop on this database, it stops at 3rd position. However, Rank-Join reads all data items of the database. There are many examples over which we have this high difference between SR_JTop and Rank-Join, in particularly when the following conditions hold:

1) The top-k join data items are produced after accessing the early positions of the lists.

2) Some of the scoring attributes, but not all of them, have a low progressive impact on the scoring function, *e.g.* all values of one of the attributes are equal or very close.

Over all the databases which the above conditions hold, it is very probable that the stop position of SR_JTop be $O(n)$ times lower than that of Rank-Join. □

Let us now discuss the cases where the threshold of SR_JTop, *i.e.* T_{SR_JT}, may be lazy. According to the definition of T_{SR_JT}, it is the maximum of three values, and in each of these values, we use $(c_1, c_2, \ldots, c_{m1})$, or $(c'_1, c'_2, \ldots, c'_{m2})$, *i.e.* the last seen local scores in DS_1 and DS_2. Assume at least one attribute from each data source does not have a low progressive impact on the scoring function, *e.g.* a_u from DS_1 and a'_v from DS_2 for some $1 \leq u \leq m_1$ and $1 \leq v \leq m_1$. Let l_u and l'_v and be the corresponding sorted lists of attributes a_u and a'_v respectively. Also, let c_u be c'_v be the last local scores seen in l_u and l'_v respectively. In each of the three values of T_{SR_JT} there exist c_u or c'_v, and the attributes a_u and a'_v does not have a low progressive impact on the scoring function, thus none of the three values of T_{SR_JT} decreases slowly, therefore their maximum, *i.e.* T_{SR_JT}, does not decrease slowly. Thus, if at least one of the scoring attributes of each data source does not have a low progressive impact on the scoring function, then the threshold of SR_JTop is not lazy. In other words, only in the cases where all scoring attributes of one of the data sources have low progressive impacts on the scoring function, T_{SR_JT} may be lazy. This is stated in the following theorem.

Theorem 3. *Only in the cases where all scoring attributes of one of the data sources have low progressive impacts on the scoring function, the threshold of SR_JTop may be lazy.*

Proof. Implied by the above discussion. □

4. LEVERAGING SEEN POSITONS

Position-based indexing is a mechanism to report the position (rank) of a data in a sorted list when it is seen via sorted or random access [2]. The position of a data in a list is formally defined as follows. Let j be the number of data items which are before a data item d in a list l_i, then the *position* of d in l_i is equal to $(j + 1)$. For example, if the sorted list is implemented using an array structure, then the position of each data item is the index of the element containing the data item (and its local score). For the other data structures such as linked list, B+-tree, etc. we can add a field to each element of the data structure to denote its position in the list. We can also have a separate data structure that maintains for each data, its positions in all lists. Thus, with only one access to the data, we obtain its position in any list.

In this section, we propose BP_JTop, an efficient top-k join algorithm for systems with position-based indexing. We show that exploiting position information leads to design faster stopping mechanisms in answering top-k join queries.

4.1 BP_JTop

BP_JTop is similar to JTop with respect to performing both sorted and random access to the lists. However, BP_JTop takes into account the positions of seen data items and develops a threshold which is much tighter than that of JTop. In contrast to JTop, which as part of its threshold uses the last local scores seen under sorted access, BP_JTop uses the local scores that are at *best positions*. These positions are usually much deeper than the last position seen under sorted access.

BP_JTop works as follows:

1. Do continually sorted access in parallel to each list l in DS_1 or DS_2. As a data item d is seen under sorted access in a list l, do random access to read all local scores of d. Maintain the local scores of the seen data item. Maintain also the positions of each data seen under sorted or random access.
2. Produce new valid join combinations of d with all data items seen in the opposite data source so far. Maintain the k join data items whose overall scores are the highest among all join data items produced so far. Maintain in a set Y the k produced join data items whose overall scores are the highest among all join data items produced so far.

3. Let P_l be the set of positions which are seen under sorted or random access in l. Let bp_l, called *best position* in l, be the highest position in P_l such that any position between 1 and bp_l is also involved in P_l.

4. Choose data items bdj_1 (called *best data for join in* DS_1) and bdj_2 (called *best data for join in* DS_2) as follows:

   bdj_i (for i=1, 2)*:* Let B_i be the set of seen data items in the data source DS_i such that each $d \in B_i$ has a join attribute value which is lower than or equal to the local score which is at the best position in the join list of the other data set. Then $bdj_i \in B_i$ is the data whose overall score is the highest among the data involved in B_i. If $B_i = \{\}$ then let bdj_i be a virtual data whose local scores are equal to the last local scores at best positions in the lists of DS_i.

5. Stop when there are at least k produced join data items whose overall scores are at least the BP_JTop threshold computed as follows. Let $(b_1, b_2, \ldots, b_{m1})$ and $(b'_1, b'_2, \ldots, b'_{m2})$ be the local scores which are at best positions in the lists of DS_1 and DS_2 respectively. Let $(e_1, e_2, \ldots, e_{m1})$ and $(e'_1, e'_2, \ldots, e'_{m2})$ be the local scores of bdj_1 and bdj_2 respectively. Then, the threshold of SR_JTop, denoted by T_{SR_JT}, is the maximum of three values as follows:

$$T_{BP\text{-}JT} = Max \{f(b_1, b_2, \ldots, b_{m1}, b'_1, b'_2, \ldots, b'_{m2}),$$
$$f(b_1, b_2, \ldots, b_{m1}, e'_1, e'_2, \ldots, e'_{m2}),$$
$$f(e_1, e_2, \ldots, e_{m1}, b'_1, b'_2, \ldots, b'_{m2})\}$$

6. Return Y to the user.

4.2 Correctness and Analysis

The following theorem provides BP_JTop's correctness.

Theorem 4. *If the scoring function f is monotonic, then BP_JTop finds correctly the top-k join answers.*

Proof. Let l be a sorted list in DS_1, and bp_l be the best position in l when BP_JTop stops. Let b_l be the local score which is at bp_l in l. The definition of best position implies that all positions before bp_l in the list l are seen. Thus, any unseen data d involved in DS_1 has a position higher than bp_l in l. Thus, the local score of d in l, *i.e.* $sc(d, l)$, is lower than or equal to the local score which is at bp_l, *i.e.* $b_l \geq sc(d, l)$. Using this inequality and in a way similar to the proof of Theorem 1, *i.e.* correctness of SR_JTop, we can easily show that the overall score of each unproduced join data is lower than or equal to the threshold of BP_JTop. □

The following theorem compares the performance of SR_JTop and BP_JTop.

Theorem 5. *The number of sorted (random) accesses done by BP_JTop is always lower than or equal to that of SR_JTop.*

Proof. Since both BP_JTop and SR_JTop do sorted access and after each sorted access they do random access to the lists, to prove the theorem it is sufficient to show that BP_JTop does less number of sorted accesses. Thus, we must show that BP_JTop stops at a position which is lower than or equal to that of SR_JTop. For this, it is sufficient to show that when SR_JTop stops, its threshold is higher than or equal to that of BP_JTop. Let l be a list, and p be the position at which SR_JTop stops, *i.e.* under sorted access. Let c be the local score which is at p. Let bp_l be the best position in the list l when SR_JTop stops, and b_l be the local score at bp_l. Since all positions before p are seen, we have $bp_l \geq p$. Thus since the lists are sorted, we have $b_l \geq c$. Using this inequality and the monotonicity of the scoring function, it can be easily shown that each of the three values, which are used in the threshold of BP_JTop, is lower than or equal to one of the three values used in the threshold of SR_JTop. Thus, the threshold of BP_JTop is lower than or equal to that of SR_JTop. □

The following observation shows that the performance of BP_JTop can be $O(m)$ times better than that of SR_JTop where m is the total number scoring attributes.

Observation 2. *There is a class of databases over which the number of sorted (or random) accesses done by BP_JTop is O(m) times lower than that of SR_JTop where m is the total number scoring attributes.*

Proof. It is sufficient to show that there are databases over which the number of sorted accesses done by BP_JTop is O(*m*) times lower than that of SR_JTop. In other words, under sorted access, BP_JTop stops at a position which is O(*m*) times lower than the position at which SR_JTop stops. Let SR_JTop stops at position *j*. For simplicity assume that $j=(m-1)*u$ where *u* is an integer. Let bdj_1 and bdj_2 be the best data for join at position *u*. Consider all cases where the two following conditions hold:

1) Each of the top-k join answers have a local score at a position which is less than or equal to *u*.

2) If a data item is at a position in interval [1 .. u)] in any list, then *m-2* of its corresponding local scores in other lists are at positions which are in interval [((u + 1) .. j], and one of its corresponding local scores is in a position higher than *j*.

3) The join local scores of bdj_1 and bdj_2 at position *u* are lower than the local score which is at position *j* in the join list of the opposite data source. This guarantees that the best data for join, *i.e.* bdj_1 and bdj_2, at positions *u* and *j* be the same.

In all cases where the two above conditions hold, we can argue as follows. After doing its sorted access and random access at position *u*, BP_JTop has seen all positions in interval [1.. u], *i.e.* under sorted access, and for each seen data item it has seen *m-2* positions in interval [u + 1) .. j], *i.e.* under random access. Let n_s be the total number of seen positions in interval [1..j], then we have:

$$n_s = (\text{number of seen positions in } [1..u]) + (\text{number of seen positions in } [(u + 1) .. j])$$

After replacing the number of seen positions, we have:

$$n_s = u*m + u*m*(m-2) = ((j/(m-1)*m) + (((j/(m-1) *m) * (m-2))$$

After simplifying the right side of the equation, we have $n_s=m*j$. Thus, when BP_JTop is at position *u*, it has seen all positions in interval [1 .. j] in all lists. Therefore, the best position in each list is at least *j*. Also, Condition 3 assures that bdj_1 and bdj_2 at positions *u* and *j* are the same. Therefore, the threshold of BP_JTop at position *u* is equal to the threshold of SR_JTop at *u*. Thus, BP_JTop stops at *u* which is *(m-1)* times lower than *j*. In other words, BP_JTop stops at a position which is O(*m*) times lower than the position at which SR_JTop stops. □

5. LIMITING RANDOM ACCESSES

In the previous sections, we assumed that random access to the data is available and its cost is very low. In this section, we deal with systems where random accesses are very expensive or impossible. As an example of such systems, we can mention information retrieval systems in which sorted lists correspond to lists of ranked documents. We propose two top-k join algorithms, LR_JTop and NR_JTop for such systems. LR_JTop does a very limited number of random accesses to the data sources, *i.e.* only for a set of final join data items, and NR_JTop does no random access.

We show that both LR_JTop and NR_JTop stop at positions which are lower than or equal to that of Rank-Join, and their performance can be *O(n)* times better than that of Rank-Join where *n* is the number of data items of the database.

5.1 LR_JTop

The main difference between SR_JTop and LR_JTop is their stopping condition. For describing the stopping condition of LR-SR_JTop, we need to give some definitions about optimistic and pessimistic overall score of a single or join data as follows. Since LR_JTop does no random access, some of the data items may be partially seen during its execution, *i.e.* only part of their local scores are seen. Let *d* be a data which is seen in at least one of the lists of DS_1 or DS_2. Let *l* be a list in DS_1 (or DS_2), and *c* be the last local score seen in *l*. Let *optimistic local score of data item d in the list l*, denoted as *opt(d, l)*, be a function computed as follows. If the local score of *d* is seen in *l* then *opt(d, l)=sc(d, l)*, else *opt(d, l)=c*. In other words, if the data is not seen in *l*, then we choose the last seen local score in *l* as the optimistic local score of the data in *l*, else we choose its real value. Let *f* be the scoring function. Assume *d* is involved in DS_1. Then, the *optimistic overall score of a single data d*, denoted by *opt-ov(d)*, is defined as *opt-*

$ov(d)= f(x_1, x_2, \ldots, x_{m1}, 0, 0, \ldots, 0)$ where $x_i= opt(d, l)$ for $1\leq i\leq m_1$. In other words, we apply the scoring function on the optimistic local scores of *d*, while setting zero for the other inputs of the scoring function. Similarly, for a data *d'* that belongs to DS_2, the optimistic overall score is defined as $opt\text{-}ov(d')= f(0, 0, \ldots,0, x'_1, x'_2, \ldots, x'_{m2})$ where $x'_j= opt(d', l_j)$ for $1\leq j\leq m_2$. Now, we define the *optimistic overall score of a join data (d, d')* to be $opt\text{-}ov(d, d')= f(x_1, x_2, \ldots, x_{m1}, x'_1, x'_2, \ldots, x'_{m2})$ where $x_i= opt(d, l)$ and $x'_j= opt(d', l_j)$ for $1\leq i\leq m_1$ and $1\leq j\leq m_2$.

Let us now define the pessimistic overall score of a join data. Let *pessimistic local score of data item d in the list l*, denoted as *psm(d, l)*, be a function computed as follows. If the local score of *d* is seen in *l* then $psm(d, l)=sc(d, l)$, else $psm(d, l)=0$. Now, we define the *pessimistic overall score of a join data (d, d')* to be $psm\text{-}ov(d, d')= f(x_1, x_2, \ldots, x_{m1}, x'_1, x'_2, \ldots, x'_{m2})$ where $x_i= psm(d, l)$ and $x'_j= psm(d', l_j)$ for $1\leq i\leq m_1$ and $1\leq j\leq m_2$.

The stopping condition of SR_JTop and LR_JTop differs in two aspects. First, SR_JTop compares the overall score of the produced join data items, *i.e.* the *k* highest ones, with its threshold, while LR_JTop compares their pessimistic overall scores with its threshold. This is because with LR_JTop, the values of the join data in some lists may be unseen. The second difference is in their threshold. The threshold of SR_JTop is designed based on the fact that with SR_JTop for any unproduced join data item *(d, d')* at least one of its elements, *i.e. d* or *d'*, is not seen by SR_JTop. However, with LR_JTop, this is not true due to no random accesses. Thus, there may be unproduced join data items such that both of their elements are partially seen, *i.e.* both of them can be seen in some lists. In the design of LR_JTop we use another property as follows: for any unproduced join data item *(d, d')* at least one of its elements has an unseen join attribute value. Based on this property, in the threshold of LR_JTop, we use two special seen data items from each data source: $Ubdj_1$ and $Sbdj_2$ from DS_1, and also $Ubdj_2$ and $Sbdj_2$ from DS_2.

5.1.1 Algorithm

LR_JTop works as follows:

1. Do continually sorted access in parallel to each list in DS_1 and DS_2, and maintain the seen local scores.
2. As a data item *d* is seen in the join lists of DS_1 or DS_2, compare it with the seen data items in the opposite data set, and produce new valid join combinations of *d* with them.
3. Choose the seen data items $Ubdj_1$ and $Ubdj_2$ respectively from $DS_1$and DS_2 as follows:

   $Ubdj_i$ *(*for i=1,2*):* Let U_i be the set of seen data in DS_i whose local score in the join list of DS_i *is not seen*. Then, $Ubdj_i$ is the data in U_i whose optimistic overall score is the highest. If $U_i=\{\}$ then let $Ubdj_i$ be a virtual data whose local scores are equal to the last local scores seen in the lists of DS_i.
4. Choose the seen data items $Sbdj_1$ and $Sbdj_2$ respectively from DS_1 and DS_2 as follows:

   $Sbdj_i$ *(*for i=1,2*):* Let S_i be the set of seen data in DS_i whose local score in the join list of DS_i *is seen*, and the local score is lower than or equal to the last local score which is seen in the join list of the opposite data source. Then, $Sbdj_i$ is the data in S_i whose optimistic overall score is the highest. If $S_i=\{\}$ then let $Sbdj_i$ be a virtual data whose local scores are equal to the last local scores seen in the lists of DS_i.
5. Stop when there are at least *k* join data items whose pessimistic overall scores are at least the LR_JTop's threshold computed as follows. Let *f* be the scoring function, then the LR_JTop's threshold is the maximum of three values as follows:

$$T_{LR\_JT} = Max\ (opt\_ov(Ubdj_1, Ubdj_2),\ opt\_ov(Sbdj_1, Ubdj_2),\ opt\_ov(Ubdj_1, Sbdj_2)\}$$

6. Let Y be the set of produced join data items. For each data $(d, d') \in Y$, do random access to find its unseen local scores (if any)[1]. Return the k join data items whose overall scores are the highest among the join data items involved in Y.

5.1.2 Correctness and Analysis

The following theorem proves LR_JTop's correctness.

Theorem 6. *If the scoring function f is monotonic, then LR_JTop finds correctly the top-k join answers.*

Proof. All join data items found by LR_JTop have a pessimistic overall score higher than or equal to LR_JTop's threshold, *i.e.* T_{LR_JT}. Thus, their real overall score is higher than T_{LR_JT}. Therefore, to prove the theorem, it is sufficient to show that any unproduced join data has an overall score lower than or equal to T_{LR_JT}. Let (d, d') be an unproduced join data item. We must show that $ov(d, d') \leq T_{LR\_JT}$. As explained before, d and d' are necessarily in one of the following situations: 1) The join attribute value of none of d and d' is seen by LR_JTop; 2) The join attribute value of d is seen but not that of d'; 3) The join attribute value of d' is seen but not that of d. Using the definition of $Ubdj_1$, $Ubdj_2$, $Sbdj_1$ and $Sbdj_2$, we can show that in the first situation $opt\_ov(d, d') \leq opt\_ov(Ubdj_1, Ubdj_2)$, in the second situation $opt\_ov(d, d') \leq opt\_ov(Sbdj_1, Ubdj_2)$, and in the third situation $opt\_ov(d, d') \leq opt\_ov(Ubdj_1, Sbdj_2)$. Thus, $opt\_ov(d, d') \leq T_{LR\_JT}$. Since the optimistic overall score of each data is higher than or equal to its real overall score, we have $opt\_ov(d, d') \geq ov(d, d')$. Therefore, we have $ov(d, d') \leq T_{LR\_JT}$. □

The following theorem compares LR_JTop and Rank-Join in terms of the position at which they stop.

Theorem 7. *The position, at which LR_JTop stops, is always lower than or equal to that of Rank-Join.*

Proof. Let p be the position at which Rank-Join stops. To prove the theorem it is sufficient to show that LR_JTop stops at most at p. Let J_k be the set of answers. The stopping condition of Rank-Join implies that all local scores of the join data involved in J_k are seen at most at position p. Thus, at most at p, LR_JTop also produces these data items, and their pessimistic local score is equal to their real overall score, *i.e.* because all their local scores are seen. Therefore, to prove the theorem, it is sufficient to show that at p, the threshold of LR_JTop is less than or equal to that of Rank-Join, *i.e.* $T_{LR\_JT} \leq T_{RJ}$. This can be done by showing that each of the three values used in computing the threshold of LR_JTop is less than or equal to at least one of the values which are used in the threshold of Rank-Join. Without loss of generality assume that the join lists in DS_1 and DS_2 are l_1 and l'_1 respectively. Let c_1 and c'_1 be the local scores which are at p in l_1 and l'_1 respectively. Since the join attribute value of $Ubdj_1$ and $Ubdj_2$ are not seen, then at p, we have $opt(Ubdj_1, l_1) = c_1$ and $opt(Ubdj_2, l'_1) = c'_1$, *i.e.* their optimistic local score in the join lists is equal to the last seen local score in the lists. Thus, for the first value of T_{LR_JT} we have: $opt\_ov(Ubdj_1, Ubdj_2) = f(c_1, x_2, \ldots, x_{m1}, c'_1, x'_2, \ldots, x'_{m2})$ where $x_i = opt(d, l_i)$ and $x'_j = opt(d', l'_j)$ for $1 \leq i \leq m_1$ and $1 \leq j \leq m_2$. Let h_i be the first local score of list l_i, for $1 \leq i \leq m_1+m_2$. Then, since the scoring function is monotonic, we have $f(c_1, x_2, \ldots, x_{m1}, c'_1, x'_2, \ldots, x'_{m2}) \leq f(c_1, h_2, \ldots, h_{m1+m2})$. The right side of this equation is one of the values used in the threshold of Rank-Join, thus we have $opt\_ov(Ubdj_1, Ubdj_2) \leq T_{RJ}$. In a similar way, we can show that $opt\_ov(Sbdj_1, Ubdj_2) \leq T_{RJ}$ and $opt\_ov(Ubdj_1, Sbdj_2) \leq T_{RJ}$. Thus, we have $T_{LR\_JT} \leq T_{RJ}$. □

Theorem 7 shows that the position at which LR_JTop stops is always lower than or equal to that of Rank-Join. We also make the following observation for the difference between the performance of LR_JTop and Rank-Join.

Observation 3. *There are databases over which the number of accesses to the data sources done by LR_JTop is O(n) times lower than that of Rank-Join where n is the number of data items.*

Proof. An example of these databases is Example 1. If we apply LR_JTop on this database then it stops at position 3. However, Rank-Join reads all data items of the database. □

[1] An optimization to LR_JTop is to remove from Y all join data whose optimistic overall score is less than the pessimistic overall score of at least k join data involved in Y. This can be done before each random access to the lists.

Let us now discuss the cases where the threshold of LR_JTop may be lazy. Let l_1 and l'_1 be the join lists in DS_1 and DS_2 respectively, and c_1 and c'_1 be the last local scores which are seen in l_1 and l'_1 respectively. As shown in the proof of Theorem 7, we have $opt(Ubdj_1, l_1)= c_1$ and $opt(Ubdj_2, l'_1)= c'_1$. According to its definition, LR_JTop's threshold, *i.e.* T_{LR_JT}, is the maximum of three values, and in each of these values, we use $Ubdj_1$ or $Ubdj_2$. Thus, in each of the three values, we use c_1 or c'_1. Therefore, as in the proof of Theorem 3, we can show that if the join attributes do not have a low progressive impact on the scoring function, then the threshold of LR_JTop is not lazy. However, even if the join attributes have a low progressive impact, the threshold of LR_JTop may not be lazy, because the local scores of $Sbdj_1$ and $Sbdj_2$ may avoid the lazyness of the threshold.

5.2 NR_JTop

In this section, we propose NR_JTop, a top-k join algorithm that does no random access to the lists, *i.e.* it does only sorted access.

5.2.1 Algorithm

The first steps of NR_JTop are the same as LR_JTop, *i.e.* until stopping sorted accesses and having the Y set. But in contrast to LR_JTop, NR_JTop does not do random access to find the unseen local scores of the data involved in Y. It does sorted access to the lists, and after each sorted access, removes useless join data items from Y, *i.e.* those that cannot be an answer. It continues until there remains only k join data in Y. For removing useless data from Y, NR_JTop uses the following rule: If the optimistic overall score of a join data is lower than the pessimistic overall score of at least k join data, then it cannot be a top-k join data item.

The NR_JTop algorithm works as follows:

Steps 1 to 5: same as in LR_JTop, *i.e.* until it stops doing sorted access to the lists.

6. Let Y be the set of produced join data. Let L_1 and L_2 be the set of sorted lists of DS_1 and DS_2 respectively. Let $L \subseteq L_1 \cup L_2$ be the set of sorted lists such that for each list $l \in L$ there is at least one join data item in Y which has an unseen local score in l.

7. For each $l \in L$ and in parallel, continue doing sorted access to l. After each sorted access, remove useless join data items from Y as follows. Let $Y_k \subseteq Y$ be the set of k join data items whose pessimistic overall score is the highest among the join data involved in Y. Remove from Y each join data item whose optimistic overall score is lower than or equal to the pessimistic overall score of all join data items involved in Y_k.

   If $/Y/ \leq k$, then stop.

8. Return Y_k to the user.

In step 6 of NR_JTop, after each sorted access and even if the algorithm does not see an unseen local score of any join data item, it checks Y for the useless join data items. The reason is that each sorted access may reduce the optimistic overall scores of some data items, *i.e.* because in computing the optimistic overall score, for unseen local scores, we consider the last local score seen in the list. Thus, each sorted access may reduce the optimistic overall score of some join data items; even if it does not see an unseen local score of any join data item.

5.2.2 Correctness and Analysis

The following lemma proves NR_JTop's correctness.

Theorem 8. *If the scoring function f is monotonic, then NR_JTop finds correctly the top-k join answers.*

Proof. The threshold of NR_JTop is the same as LR_JTop, thus Theorem 6, *i.e.* the correctness of NR_JTop, implies that at step 5 of the algorithm, the set Y involves all top-k answers. Thus, to prove the theorem, it is sufficient to show that LR_JTop finds correctly the top-k answers from Y. This can be done easily by using the fact that for any two join data items (d_1, d'_1) and (d_2, d'_2), if $opt\_ov(d_2, d'_2) \leq psm\_ov(d_1, d'_1)$ then we have $ov(d_2, d'_2) \leq ov(d_1, d'_1)$. Thus, the k join data items that finally remain in Y are the top-k join answers. □

The following theorem compares the performance of NR_JTop and Rank-Join.

Theorem 9. *The number of accesses to the data sources done by NR_JTop is always less than or equal to that of Rank-Join.*

Proof. Let p be the position at which Rank-Join stops. Let J_k be the set of answers. The stopping condition of Rank-Join implies that all local scores of the join data involved in J_k are seen at most at position p, and their overall score is higher than the maximum possible overall score of any join data item which is not involved in J_k. Theorem 7 implies that with LR_JTop, all top-k answers are involved in the set Y before position p. This is also true for NR_JTop, since LR_JTop and NR_JTop have the same threshold. Thus for proving the theorem, it is sufficient to show that at most at p, the pessimistic overall score of top-k answers becomes higher than or equal to the optimistic overall score of other join data involved in Y. This can be done easily by using the fact that at p, the pessimistic overall score of top-k answers are equal to their overall score, *i.e.* because their local scores are seen, and their overall score is higher than or equal to the maximum possible overall score of any other join data items, *i.e.* their optimistic overall score. □

The following observation shows that performance of NR_JTop can be $O(n)$ times better than that of Rank-Join.

Observation 4. *There are databases over which the number of accesses to the data sources done by NR_JTop is O(n) times lower than that of Rank-Join where n is the number of data items.*

Proof. An example of these databases is that of Example 1. If we apply NR_JTop on this database then it stops at position 3. However, Rank-Join reads all data items of the database. □

6. PERFORMANCE EVALUATION

In this section, we compare our algorithms with Rank-Join through experimentation. The rest of this section is organized as follows. We first describe our experimental setup. Then, we compare the performance of our algorithms with Rank-Join by varying experimental parameters such as the number of scoring attributes, the number of top data items requested, *i.e. k*, the type of the scoring function, the number of data items in data sources, etc.

6.1 Experimental Setup

We implemented the JTop algorithms and Rank-Join in Java. To evaluate our algorithms, we tested them over both independent and correlated databases, thus covering all practical cases. The independent databases are uniform and Gaussian databases generated using the two main probability distributions (*i.e.* uniform and Gaussian). With Uniform and Gaussian databases, the positions of a data item in any two lists are independent of each other. To generate the uniform database, the scores of the data items in each list are generated using a uniform random generator, and then the list is sorted. To generate the Guassian database, the scores of the data items in each list are Gaussian random numbers with a mean of 0 and a standard deviation of 1.

In addition to these independent databases, we use correlated databases, *i.e.* databases where the positions of a data item in the lists are correlated. This is to take into account applications where there are correlations among the positions of a data item in different lists. Inspired from [24], we use a correlation parameter α ($0 \leq \alpha \leq 1$), and generate the data items of each data source of the correlated database as follows. For the first list of the data source, we randomly select the position of data items. Let p_1 be the position of a data item in the first list, then for each list L_i ($2 \leq i \leq m$) we generate a random number r in interval $[1 .. n*\alpha]$ where n is the number of data items, and we put the data item at a position p whose distance from p_1 is r. If p is not free, *i.e.* occupied previously by another data item, we put the data item at the free position closest to p. By controlling the value of α, we can create databases with stronger or weaker correlations. In our tests, the default value for α is 0.01. After setting the positions of all data items in all lists, we generate the scores of the data items in each list in such a way that they follow the Zipf law [34] with the Zipf parameter $\theta = 0.7$.

In our tests, the scoring function has the following form: $f_w(x_1, x_2, .., x_{m1}, x'_1, \ldots, x'_{m2}) = \omega*(x_1) + x_2, .., x_{m1} + x'_1, \ldots, x'_{m2}$. In this scoring function, we use the parameter ω in order to control the progressive impact of one of the attributes, *i.e.* x_1, on the scoring function. In our tests, we use four versions of this scoring function: f_1, $f_{0.5}$, $f_{0.1}$, and f_0, *i.e.* with ω= 1, 0.5, 0.1, and 0. The scoring function f_1 computes the sum of the local scores for each data item. This is our default scoring function. The attribute x_1 has no impact on the scoring function f_0, and its impact on $f_{0.5}$, $f_{0.1}$ is lower than that of the other attributes.

Our default settings for different experimental parameters are shown in Table 1. The default value for join selectivity is 0.01. To provide this selectivity, we randomly select some scores from the join list of one data source of the database, and repeat them in the join list of the other data source. In our tests, the default number of data items in each data source is 20,000. Typically, users are interested in a small number of top answers, thus unless otherwise specified we set *k=20*. The default number for the scoring attributes of each data source is 2, *i.e.* $m_1=m_2=2$.

Table 1. Default setting of experimental parameters

Parameter	Default values
Number of data items in each data source	20,000
k	20
Join selectivity	0.01
Scoring function	SUM
Number of scoring attributes per data source	$m_1=m_2= 2$

To evaluate the performance of the algorithms, we measure the two following metrics:

1) Stop position. This is the position at which an algorithm stops. This metric can also be used for comparing the number of sorted accesses done by algorithms, because the number of sorted accesses is proportional to stop position.

2) Number of accesses. This is the total number of sorted and random accesses done by an algorithm. For Rank-Join and NR_JTop algorithms, this metric measures the number of sorted accesses. For the other JTop algorithms, in addition to sorted accesses, this metric also counts the random accesses that they do after seeing each data item via sorted access for the first time.

6.2 Performance Results

6.2.1 Effect of the number of scoring attributes

In this section, we compare the performance of our JTop algorithms with Rank-Join by varying the number of scoring attributes per data source. Let us denote by *m* the number of scoring attributes per data source, *i.e.* $m= m_1=m_2$.

Over the uniform database, with the number of scoring attributes per data source equal to 2, 3 and 4, and the other parameters set as in Table 1, Figures 2 and 3 show the results measuring number of accesses and stop position respectively. The number of accesses done by JTop algorithms is much lower than that of Rank-Join; *e.g.* they outperform Rank-Join by a factor of at least 2.5 when $m\geq3$. BP_JTop is the strongest performer: it outperforms Rank-Join by a factor of approximately *5* for $m\geq3$. Even the NR_JTop algorithm, which does no random access, outperforms Rank-Join by a factor of about 3. On the second metric, *i.e.* stop position, generally JTop algorithms outperform Rank-Join with factors higher than that on number of accesses; *e.g.* SR_JTop outperforms Rank-Join by a factor of at least 6 when m=3. However, on both metrics, NR_JTop outperforms Rank_Join with the same factors, because NR_JTop does no random access.

6.2.2 Effect of k

We now study the effect of the number of requested answers, *i.e. k*, on performance. Figure 4 shows how the stop position increases over the uniform database, with increasing *k* up to 50, and the other parameters set as in Table 1. The stop position of all five algorithms increases with *k* because more data items are needed to be returned in order to obtain the top-k join data items. However, the increase is very small. The reason is that over the uniform database, when an algorithm (*i.e.* any of the five algorithms) stops its execution for a top-k query, with a high probability, it has seen also the *(k + 1)*th join data item. Thus, with a high probability, it stops at the same position for a top-(k+1) query.

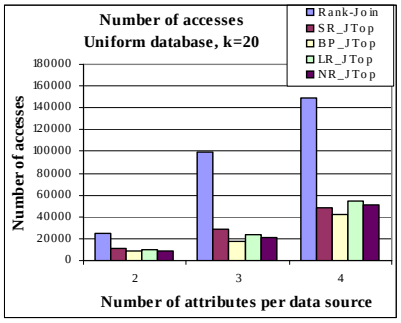

Figure 2. Number of accesses vs. number of scoring attributes per data source

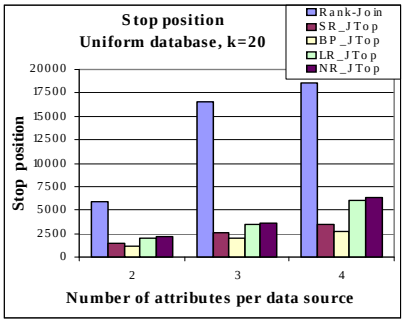

Figure 3. Stop position vs. number of scoring attributes per data source

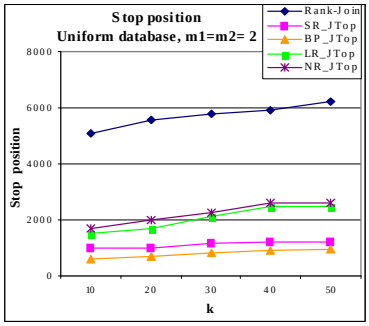

Figure 4. Stop position vs. *k*

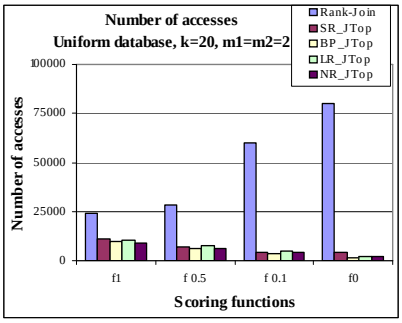

Figure 5. Number of accesses vs. different scoring functions

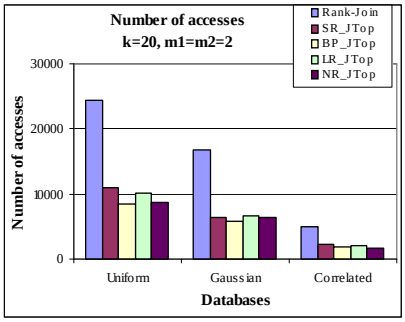

Figure 6. Number of accesses over different databases

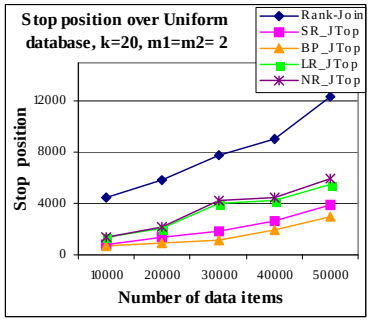

Figure 7. Stop position vs. number of data items

6.2.3 Effect of scoring function

We now investigate the impact of our scoring functions on the performance of the JTop algorithms and Rank-Join. Over the uniform database, Figure 5 shows the number of accesses done by the five algorithms with the scoring function equal to f_1, $f_{0.5}$, $f_{0.1}$, and f_0 (*i.e.* ω= 1, 0.5, 0.1, and 0) as described in Section 6.1, and the other parameters set as in Table 1. With decreasing the parameter ω, the number of accesses done by Rank-Join increases significantly. The reason is that decreasing ω decreases the progressive impact of one of the attributes on the scoring function, thus the threshold of Rank-Join decreases more slowly, so it stops later. The worst performance of Rank-Join is with the scoring function f_0, because the progressive impact of the scoring attribute x_1 is zero. Thus the threshold of Rank-Join does not decrease, and it has to read all data items of the data sources. Unlike Rank-Join, the number of accesses done by JTop algorithms decreases with decreasing the parameter ω.

6.2.4 Results over different databases

Figure 6 shows the number of accesses done by the algorithms over uniform, Gaussian and correlated databases, with the other parameters set as in Table 1. Over all three databases, JTop algorithms outperform significantly Rank-Join. Over the correlated database, the performance of the five algorithms is much better than that over Gaussian and uniform databases. The reason is that in a highly correlated database, the top-k data items are distributed over low positions of the lists, so usually the algorithms do not need to go much down in the lists, and they stop soon. However, due to their efficient stopping mechanism, JTop algorithms stop much sooner than Rank-Join.

6.2.5 Effect of the number of data items

We now vary the number of data items in each data source, and investigate its effect on performance. Figure 7 shows how stop position increases over the uniform database with increasing the number of data items up to 50,000, and with the other parameters set as in Table 1. Increasing the number of data items has a considerable impact on the performance of all five algorithms. The reason is that when we enlarge the lists and generate random data for them, the top-k data items are distributed over higher positions in the lists.

7. RELATED WORK

Efficient processing of top-k queries is an important and hard problem that is still receiving much attention. One of the most efficient algorithms for top-k selection queries is the TA algorithm which was proposed by several groups [14][16][27]. Several TA-style algorithms, *i.e.* extensions of TA, have been proposed for processing top-k queries in different environments, *e.g.* [3][5] [12][24][32]. Recently, we proposed efficient algorithms, called BPA [2], which by taking into account the positions of seen data items develop a stopping mechanism which is much more efficient than TA. The idea of best positions which we used as part of the BP_JTop algorithm is inspired from BPA. However, top-k selection algorithms, including BPA, assume that there is no join operation in the query. Otherwise, their stopping mechanisms do not work correctly.

In previous works on top-k join queries, the most efficient top-k join algorithm is Rank-Join [17][18] which we already discussed much. In [26], the authors introduce the *J** algorithm which deals with efficient processing of top-k join queries over ranked inputs. *J** maps the top-k join problem to a search problem in the Cartesian space of the ranked inputs. It uses a version of the *A** search algorithm to guide navigation in this space to produce the results. The experimental studies reported in [18] show that Rank-Join significantly outperform *J**. In [30], ranked join indices are proposed for the efficient evaluation of top-k join queries. The indices need to be pre-produced, and make the number of requested results, *i.e. k*, limited to a predefined number, *e.g. K*, thus the user can not choose *k* to a higher number than *K*. In [1] and [8], the relational algebra is extended to support rank queries, *i.e.* top-k join queries, as a first-class construct. In [8], the authors also present a pipelined and incremental execution model of rank query plans. In [9], top-k join query processing is extended to aggregate queries. The top-*k* join queries are also discussed briefly in [7] as a possible extension to their algorithm which evaluates top-*k* selection queries.

8. CONCLUSION

In this report, we addressed the problem of processing of top-k join queries, and proposed JTop, a family of efficient algorithms for top-k queries. The main idea is to take advantage of the specific information on join attribute values as well as the characteristics of the underlying system. We analytically compared our algorithms with Rank-Join, which is considered as the most efficient algorithm for top-k queries, and proved that our algorithms always stop sooner than Rank-Join, and thus are more efficient. We also showed that there are databases over which the performance of our algorithms is *O(n)* times better than that of Rank-Join where *n* is the number of data items in the database.

We conducted an extensive experimental study to evaluate the performance of our algorithms under different data distributions. The performance evaluation shows that over the tested databases our algorithms significantly outperform the Rank-Join algorithm.

REFERENCES

[1] S. Adali, C. Bufi, and M. L. Sapino. Ranked relations: Query languages and query processing methods for multimedia. *Multimedia Tools and Applications*, 24(3), 197-214, 2004.

[2] R. Akbarinia, E. Pacitti and P. Valduriez. Best position algorithms for top-k queries. *VLDB Conf.*, 495-506, 2007.

[3] H. Bast, D. Majumdar, R. Schenkel, M. Theobald and G. Weikum. IO-Top-k: index-access optimized top-k query processing. *VLDB Conf.*, 475-486, 2006.

[4] N. Bruno, L. Gravano and A. Marian. Evaluating top-k queries over web-accessible databases. *ICDE Conf.*, 369-382, 2002.

[5] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. *PODC Conf.*, 206-215, 2004.

[6] G. Das, D. Gunopulos, N. Koudas and N. Sarkas. Ad-hoc Top-k Query Answering for Data Streams. *VLDB Conf.*, 183-194, 2007.

[7] K.C.-C. Chang and S.-W. Hwang. Minimal probing: supporting expensive predicates for top-k queries. *SIGMOD Conf.*, 2002.

[8] C. Li, K. C.-C. Chang, I.F. Ilyas, S. Song. RankSQL: Query Algebra and Optimization for Relational Top-k Queries. *SIGMOD Conf.*, 131-142, 2005.

[9] C. Li, K. C.-C. Chang, I.F. Ilyas. Supporting ad-hoc ranking aggregates. *SIGMOD Conf*, 61-72, 2006.

[10] S. Chaudhuri, L. Gravano and A. Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE Trans. on Knowledge and Data Engineering*, 16(8), 992- 1009, 2004.

[11] P. Ciaccia and M. Patella. Searching in metric spaces with user-defined and approximate distances. *ACM Transactions on Database Systems (TODS)*, 27(4), 398-437, 2002.

[12] G. Das, D. Gunopulos, N. Koudas and D. Tsirogiannis. Answering top-k queries using views. *VLDB Conf.*, 451-462, 2006.

[13] R. Fagin. Combining fuzzy information from multiple systems. *J. Comput. System Sci.*, 58(1), 83-99, 1999.

[14] R. Fagin, A. Lotem and M. Naor. Optimal aggregation algorithms for middleware. *PODS Conf.*, 102-113, 2001.

[15] R. Fagin, J. Lotem and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. System Sci.*, 66(4), 614-656, 2003.

[16] U. Güntzer, W. Kießling and W.-T. Balke. Towards efficient multi-feature queries in heterogeneous environments. *IEEE Int. Conf. on Information Technology, Coding and Computing (ITCC)*, 419-428, 2001.

[17] I.F. Ilyas, W.G. Aref, A.K. Elmagarmid. Supporting Top-k Join Queries in Relational Databases. *VLDB Conf.*, 754-765, 2003.

[18] I.F. Ilyas, W.G. Aref, A.K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB Journal*, 13(3), 207-221, 2004.

[19] G.R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems (TODS)*, 28(4), 517-580, 2003.

[20] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. *PODS Conf.*, 173-182, 2006.

[21] N. Koudas, B.C. Ooi, K.L. Tan and R. Zhang. Approximate NN queries on streams with guaranteed error/performance bounds. *VLDB Conf.*, 804-815, 2004.

[22] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. *VLDB Conf.*, 129-140, 2003.

[23] A. Metwally, D. Agrawal, A. El Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *J. ACM Transactions on Database Systems (TODS)*, 31(3), 1095-1133, 2006.

[24] S. Michel, P. Triantafillou and G. Weikum. KLEE: A framework for distributed top-k query algorithms. *VLDB Conf.*, 637-648, 2005.

[25] K. Mouratidis, S. Bakiras and D. Papadias. Continuous monitoring of top-k queries over sliding windows. *SIGMOD Conf.*, 635-646, 2006.

[26] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, J. S. Vitter. Supporting incremental join queries on ranked input. *VLDB Conf.*, 281–290, 2001.

[27] S. Nepal and M.V. Ramakrishna. Query processing issues in image (multimedia) databases. *ICDE Conf.*, 22-29, 1999.

[28] C. Re, N.N. Dalvi, D. Suciu. Efficient Top-k Query Evaluation on Probabilistic Data. *ICDE Conf.*, 886-895, 2007.

[29] A. Silberstein, R. Braynard, C.S. Ellis, K. Munagala and J. Yang. A sampling-based approach to optimizing top-k queries in sensor networks. *ICDE Conf.*, 2006.

[30] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, D. Srivastava. Ranked Join Indices. *ICDE Conf.*, 2003.

[31] M. Wu, J. Xu, X. Tang and W-C Lee. Monitoring top-k query in wireless sensor networks. *ICDE Conf.*, 2006.

[32] D. Xin, J. Han and K. C-C. Chang. Progressive and selective merge: computing top-k with ad-hoc ranking functions. *SIGMOD Conf.*, 103-114, 2007.

[33] M. L. Yiu, X. Dai, N. Mamoulis, M. Vaitis. Top-k Spatial Preference Queries. *ICDE Conf.*, 1076-1085, 2007.

[34] G.K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley Press, 1949.