Best Position Algorithms for Efficient Top-k Query Processing

Reza Akbarinia, Esther Pacitti, Patrick Valduriez

Best Position Algorithms for Efficient Top-k Query Processing

Reza Akbarinia[1], Esther Pacitti[2], Patrick Valduriez[1]

[1]*INRIA and LIRMM, Montpellier, France*

[2] *LIRMM and INRIA, Univ. Montpellier, France*

Email: {[1]Firstname.Lastname@inria.fr, [2]pacitti@lirmm.fr }

Abstract. The general problem of answering top-k queries can be modeled using lists of data items sorted by their local scores. The main algorithm proposed so far for answering top-k queries over sorted lists is the Threshold Algorithm (TA). However, TA may still incur a lot of useless accesses to the lists. In this paper, we propose two algorithms that are much more efficient than TA. First, we propose the best position algorithm (BPA). For any database instance (*i.e.* set of sorted lists), we prove that BPA stops as early as TA, and that its execution cost is never higher than TA. We show that there are databases over which BPA executes top-k queries *O(m)* times faster than that of TA, where *m* is the number of lists. We also show that the execution cost of our algorithm can be *(m-1)* times lower than that of TA. Second, we propose the BPA2 algorithm which is much more efficient than BPA. We show that the number of accesses to the lists done by BPA2 can be about *(m-1)* times lower than that of BPA. We evaluated the performance of our algorithms through extensive experimental tests. The results show that over our test databases, BPA and BPA2 achieve significant performance gains in comparison with TA.

Keywords: Query processing; top-k queries; threshold algorithm; instance optimality.

1 Introduction

Top-k queries have attracted much interest in many different areas such as network and system monitoring [8][23], information retrieval [18][22][32][36], sensor networks [34][39], multimedia databases [9][17][31], spatial data analysis [10], P2P systems [3][4], data stream management systems [13][21][27][30], probabilistic databases [33][35][40], temporal databases[26], etc. The main reason for such interest is that they avoid overwhelming the user with large numbers of uninteresting answers which are resource-consuming.

The problem of answering top-k queries can be modeled as follows [14][16]. Suppose we have *m* lists of *n* data items such that each data item has a local score in each list and the lists are sorted according to the local scores of their data items. Each data item has an overall score computed based on its local scores in all lists using a given scoring function. Then, the problem is to find the *k* data items whose overall scores are the highest. This problem model is simple and general. Let us illustrate with the following examples. Suppose we want to find the top-k tuples in a relational table according to some scoring function over its attributes. To answer this query, it is sufficient to have a sorted (indexed) list of the values of each attribute involved in the scoring function, and return the *k* tuples whose overall scores in the lists are the highest. As another example, suppose we want to find the top-k documents whose aggregate rank is the highest wrt. some given keywords. To answer this query, the solution is to have for each keyword a ranked list of documents, and return the *k* documents whose aggregate rank in all lists are the highest.

There has been much work on efficient top-k query processing over sorted lists. A naïve algorithm is to scan all lists from beginning to end, maintain the local scores of each data item, compute the overall scores, and return the *k* highest scored data items. However, this algorithm is executed in *O(m∗n)* and thus it is inefficient for very large lists.

In related work, the main efficient algorithm for answering top-k queries over sorted lists is the Threshold Algorithm (TA) [15][17][31]. TA is applicable for queries where the scoring function is monotonic. It is simple and elegant. Based on TA, many algorithms have been proposed for top-k query processing in centralized and distributed applications, *e.g.* [5][13][28]. The basic difference between TA and previously designed algorithms, *e.g.* Fagin's algorithm (FA) [14], is its stopping mechanism that enables TA to stop scanning the lists very soon. However, there are many database instances over which TA continuous scanning the lists although it has seen all top-k answers (see Example 2 in Section 3.2), while it is possible to stop much sooner.

In this paper, we propose two new algorithms for processing top-k queries over sorted lists. First, we propose the best position algorithm (BPA) which executes top-k queries much more efficiently than TA. The key idea of BPA is that its stopping mechanism takes into account special seen positions in the lists, the *best positions*. For any database instance (*i.e.* set of sorted lists), we prove that BPA stops as early as TA, and that its execution cost (called middleware cost in [16]) is never higher than TA. We prove that the position at which BPA stops can be *O(m)* times lower than that of TA, where *m* is the number of lists. We also prove that the execution cost of our algorithm can be *(m-1)* times lower than that of TA. Second, based on BPA, we propose the BPA2 algorithm which is much more efficient than BPA. We show that the number of accesses to the lists done by BPA2 can be up to about *(m-1)* times lower than that of BPA. To validate our contributions, we implemented our

algorithms. The performance evaluation shows that over our test databases, BPA and BPA2 outperform TA by factors of about $(m+6)/8$ and $(m+1)/2$ respectively, *e.g.* for $m=10$, the factor is about 2 and 5.5, respectively.

This paper is a major extension of [2], with the following new material. In addition to a more complete related work and bibliography study, this paper contains two new sections. First, in Section 6, we propose several techniques using different data structures for best position management which is crucial for correct and efficient execution of our algorithms. We analyze the time and space complexity of the proposed techniques and compare them. Second, in Section 7 we provide a complete discussion on the instance optimality of TA and BPA. Instance optimality has been advocated as a main feature of TA [16], i.e. TA is optimal over all database instances. In [2], by using the instance optimality of TA and the fact that BPA stops always before TA, we proved the instance optimality of BPA. But as we will illustrate in Section 7, the existence of deterministic algorithms such as BPA shows that if we are aware of positions of seen data, then one of the main arguments used for proving the instance optimality of TA (in [16]) is invalidated. Therefore, in this case the proof of TA's instance optimality is incorrect, and must be revisited.

The rest of this paper is organized as follows. In Section 2, we define the problem which we address in this paper. Section 3 presents some background on FA and TA. In Sections 4 and 5, we present the BPA and BPA2 algorithms, respectively, with a cost analysis. In Section 6, we propose several efficient techniques for managing best positions. Section 7 gives a discussion on the instance optimality of TA. Section 8 reports a performance evaluation of our algorithms. In Section 9, we discuss related work. Section 10 concludes.

2   Problem definition

Let D be a set of n data items, and $L_1, L_2, …, L_m$ be m lists such that each list L_i contains n pairs of the form $(d, s_i(d))$ where $d \in D$ and $s_i(d)$ 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 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 $(j + 1)$.

The set of m sorted lists is called a *database*. In a distributed system, sorted lists may be maintained at different nodes. A node that maintains a list is called a *list owner*. In centralized systems, the owner of all lists is only one node.

The *overall score* of each data item d is computed as $f(s_1(d), s_2(d), …, s_m(d))$ where f is a given scoring function. In other words, the overall score is the output of f where the input is the local scores of d in all lists. In this paper, we assume that the scoring function is monotonic. A function f is monotonic if $f(x_1, …, x_m) \leq f(x'_1, …, x'_m)$ whenever $x_i \leq x'_i$ for every i.

Many of the popular aggregation functions, *e.g.* Min, Max, Average, are monotonic. The k data items whose overall scores are the highest among all data items, are called the *top-k data items*.

As defined in [16], we consider two modes of access to a sorted list. 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. Let c_s be the cost of a sorted access, and c_r be the cost of a random access. Then, if an algorithm does a_s sorted accesses and a_r random accesses for finding the top-k data items, then its *execution cost* is computed as $a_s * c_s + a_r * c_r$. The execution cost (called middleware cost in [16]) is a main metric to evaluate the performance of a top-k query processing algorithm over sorted lists [16].

Let us now state the problem we address. Let $L_1, L_2, …, L_m$ be m sorted lists, and D be the set of data items involved in the lists. Given a top-k query which involves a number $k \leq n$ and a monotonic scoring function f, our goal is to find a set $D' \subseteq D$ such that $|D'| = k$, and $\forall d_1 \in D'$ and $\forall d_2 \in (D-D')$ the overall score of d_1 is at least the overall score of d_2, while minimizing the execution cost.

3   Background

The background for this paper is the TA algorithm which is itself based on Fagin's Algorithm (FA). FA and TA are designed for processing top-k queries over sorted lists. In this section, we briefly describe and illustrate FA and TA.

3.1   FA

The basic idea of FA is to scan the lists until having at least k data items which have been seen in all lists, then there is no need to continue scanning the rest of the lists [14]. FA works as follows:

1. Do sorted access in parallel to each of the m sorted lists, and maintain each seen data item in a set S. If there are at least k data items in S such that each of them has been seen in each of the m lists, then stop doing sorted access to the lists.

2. For each data item d involved in S, do random access as needed to each of the lists L_i to find the local score of d in L_i, compute the overall score of d, and maintain it in a set Y if its score is one of the k highest scores computed so far.

Position	List 1		List 2		List 3		$f = s_1 + s_2 + s_3$			
	Data item	Local score s_1	Data item	Local score s_2	Data item	Local score s_3	TA Threshold		Data item	Overall Score
1	d_1	30	d_2	28	d_3	30	88		d_1	65
2	d_4	28	d_6	27	d_5	29	84		d_2	63
3	d_9	27	d_7	25	d_8	28	80		d_3	70
4	d_3	26	d_5	24	d_4	25	75		d_4	66
5	d_7	25	d_9	23	d_2	24	72		d_5	70
6	d_8	23	d_1	21	d_6	19	63		d_6	60
7	d_5	17	d_8	20	d_{13}	15	52		d_7	61
8	d_6	14	d_3	14	d_1	14	42		d_8	71
9	d_2	11	d_4	13	d_9	12	36		d_9	62
10	d_{11}	10	d_{14}	12	d_7	11	33		…	…
11	d_{14}	9	d_{11}	10	d_{11}	8	27		…	…
…	…	…	…	…	…	…	…		…	…
	(a)						(b)		(c)	

Figure 1. Example database. a) 3 sorted lists. b) TA threshold at positions 1 to 10. c) The overall score of each data item.

3. Return Y.

The correctness proof of FA can be found in [14]. Let us illustrate FA with the following example.

Example 1. Consider the database (*i.e.* three sorted lists) shown in Figure 1.a. Assume a top-3 query Q, *i.e.* $k=3$, and suppose the scoring function computes the sum of the local scores of the data item in all lists. In this example, before position 7, there is no data item which can be seen in all lists, so FA cannot stop before this position. After doing the sorted access at position 7, FA sees d_5 and d_8 which are seen in all lists, but this is not sufficient for stopping sorted access. At position 8, the number of data items which are seen in all lists is 5, *i.e.* d_1, d_3, d_5, d_6 and d_8. Thus, at position 8, there are at least k data items which are seen by FA in all lists, thus FA stops doing sorted access to the lists. Then, for the data items which are seen only in some of the lists, *e.g.* d_2, FA does random access and finds their local scores in all lists, *e.g.* d_2 is not seen in L_1 so FA needs a random access to L_1 to find the local score of d_2 in this list. It computes the overall score of all seen data items, and returns to the user the k highest scored ones.

3.2 TA

The main difference between TA and FA is their stopping mechanism which decides when to stop doing sorted access to the lists. The stopping mechanism of TA uses a threshold which is computed using the last local scores seen under sorted access in the lists. Thanks to its stopping mechanism,

over any database, TA stops at a position (under sorted access) which is less than or equal to the position at which FA stops [16]. TA works as follows:

1. Do sorted access in parallel to each of the m sorted lists. As a data item d is seen under sorted access in some list, do random access to the other lists to find the local score of d in every list, and compute the overall score of d. Maintain in a set Y the k seen data items whose overall scores are the highest among all data items seen so far.

2. For each list L_i, let s_i be the last local score seen under sorted access in L_i. Define the threshold to be $\delta = f(s_1, s_2, \ldots, s_m)$. If Y involves k data items whose overall scores are higher than or equal to δ, then stop doing sorted access to the lists. Otherwise, go to 1.

3. Return Y.

The correctness proof of TA can be found in [16]. Let us illustrate TA with the following example.

Example 2. Consider the three sorted lists shown in Figure 1.a and the query Q of Example 1, *i.e.* $k=3$ and the scoring function computes the sum of the local scores. The thresholds of the positions and the overall score of data items are shown in Figure 1.b and 1.c, respectively. TA first looks at the data items which are at position 1 in all lists, *i.e.* d_1, d_2, and d_3. It looks up the local score of these data item in other lists using random access and computes their overall scores. But,

the overall score of none of them is as high as the threshold of position 1. Thus, at position 1, TA does not stop. At this position, we have $Y=\{d_1, d_2, d_3\}$, i.e. the k highest scored data items seen so far. At positions 2 and 3, Y involves $\{d_3, d_4, d_5\}$ and $\{d_3, d_5, d_8\}$ respectively. Before position 6, none of the data items involved in Y has an overall score higher than or equal to the threshold value. At position 6, the threshold value gets 63, which is less than the overall score of the three data items involved in Y, i.e. $Y=\{d_3, d_5, d_8\}$. Thus, there are k data items in Y whose overall scores are higher than or equal to the threshold value, so TA stops at position 6. The contents of Y at position 6 are exactly equal to its contents at position 3. In other words, at position 3, Y already contains all top-k answers. But TA cannot detect this and continues until position 6. In this example, TA does three useless sorted accesses in each list, thus a total of 9 useless sorted accesses and 9∗2 useless random accesses.

In the next section, we propose an algorithm that always stops as early as TA, so it is as fast as TA. Over some databases, our algorithm can stop at a position which is (m-1) times lower than the stopping position of TA.

4   Best position algorithms

In this section, we first propose our Best Position Algorithm (BPA), which is an efficient algorithm for the problem of answering top-k queries over sorted lists. Then, we analyze its execution cost and compare it with TA.

4.1   Algorithm

BPA works as follows:

1. Do sorted access in parallel to each of the m sorted lists. As a data item d is seen under sorted access in some list, do random access to the other lists to find the local score and the position of d in every list. Maintain the seen positions and their corresponding local scores. Compute the overall score of d. Maintain in a set Y the k seen data items whose overall scores are the highest among all data items seen so far.

2. For each list L_i, let P_i be the set of positions which are seen under sorted or random access in L_i. Let bp_i, called *best position*[1] in L_i, be the greatest position in P_i such that any position of L_i between 1 and bp_i is also in P_i. Let $s_i(bp_i)$ be the local score of the data item which is at position bp_i in list L_i.

3. Let *best positions overall score* be $\lambda = f(s_1(bp_1), s_2(bp_2), ..., s_m(bp_m))$. If Y involves k data items whose overall scores are higher than or equal to λ, then stop doing sorted access to the lists. Otherwise, go to 1.

4. Return Y.

---

[1] bp_i is called best because we are sure that all positions of L_i between 1 and bp_i have been seen under sorted or random access.

Example 3. To illustrate our algorithm, consider again the three sorted lists shown in Figure 1.a and the query Q in Example 1. At position 1, BPA sees the data items d_1, d_2, and d_3. For each seen data item, it does random access and obtains its local score and position in all lists. Therefore, at this step, the positions which are seen in list L_1 are the positions 1, 4, and 9 which are respectively the positions of d_1, d_3 and d_2. Thus, we have $P_1=\{1, 4, 9\}$ and the best position in L_1 is $bp_1 = 1$ (since the next position in P_1 is 4 meaning that positions 2 and 3 have not been seen). For L_2 and L_3 we have $P_2=\{1, 6, 8\}$ and $P_3=\{1, 5, 8\}$, so $bp_2 = 1$ and $bp_3 = 1$. Therefore, the best positions overall score is $\lambda = f(s_1(1), s_2(1), s_3(1)) = 30 + 28 + 30 = 88$. At position 1, the set of three highest scored data items is $Y=\{d_1, d_2, d_3\}$, and since the overall score of these data items is less than λ, BPA cannot stop. At position 2, BPA sees d_4, d_5, and d_6. Thus, we have $P_1=\{1, 2, 4, 7, 8, 9\}$, $P_2=\{1, 2, 4, 6, 8, 9\}$ and $P_3=\{1, 2, 4, 5, 6, 8\}$. Therefore, we have $bp_1=2$, $bp_2=2$ and $bp_3=2$, so $\lambda = f(s_1(2), s_2(2), s_3(2)) = 28 + 27 + 29 = 84$. The overall score of the data items involved in $Y=\{d_3, d_4, d_5\}$ is less than 84, so BPA does not stop. At position 3, BPA sees d_7, d_8, and d_9. Thus, we have $P_1 = P_2 = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and $P_3 = \{1, 2, 3, 4, 5, 6, 8, 9, 10\}$. Thus, we have $bp_1=9$, $bp_2=9$ and $bp_3=6$. The best positions overall score is $\lambda = f(s_1(9), s_2(9), s_3(6)) = 11 + 13 + 19 = 43$. At this position, we have $Y=\{d_3, d_5, d_8\}$. Since the score of all data items involved in Y is higher than λ, our algorithm stops. Thus, BPA stops at position 3, i.e. exactly at the first position where BPA has all top-k answers. Remember that over this database, TA and FA stop at positions 6 and 8 respectively.

The following theorem provides the correctness of our algorithm.

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

Proof. For each list L_i, let bp_i be the best position in L_i at the moment when BPA stops. Let Y be the set of the k data items found by BPA, and d be the lowest scored data item in Y. Let s be the overall score of d, then we show that each data item, which is not involved in Y, has an overall score less than or equal to s. We do the proof by contradiction. Assume there is a data item $d'\notin Y$ with an overall score s' such that $s' > s$. Since d' is not involved in Y and its overall score is higher than s, we can imply that d' has not been seen by BPA under sorted or random access. Thus, its position in any list L_i is greater than the best position in L_i, i.e. bp_i. Therefore, the local score of d' in any list L_i is less than the local score which is at bp_i, and since the scoring function is monotonic, the overall score of d' is less than or equal to the best positions overall score, i.e. $s' \leq \lambda$. Since the score of all data items involved in Y is higher than or equal to λ, we have $s \geq \lambda$. By comparing the two latter inequalities, we have $s \geq s'$, which yields to a contradiction. □

4.2 Cost analysis

In this section, we compare the execution cost of BPA and TA. Since TA and BPA are designed for monotonic scoring functions, we implicitly assume that the scoring function is monotonic.

The two following lemmas compare the number of sorted/random accesses done by BPA and TA.

Lemma 1. *The number of sorted accesses done by BPA is always less than or equal to that of TA. In other words, BPA stops always as early as TA.*

Proof. Let Y be the set of answers found by TA, and δ be the value of TA's threshold at the time it stops. We know that the overall score of any data item involved in Y is less than or equal to δ. For each list L_i, let p_i be the position of the last data item seen by TA under sorted access. Since any position less than or equal to p_i has been seen under sorted access, the best position in L_i, *i.e.* bp_i, is greater than or equal to p_i. Thus, the local score which is at p_i is higher than or equal to the local score at bp_i. Therefore, considering the monotonicity of the scoring function, the TA's threshold, *i.e.* δ, is higher than or equal to the best positions overall score which is used by BPA, *i.e.* λ,. Thus, the overall scores of the data items involved in Y get higher than or equal to λ when the position of BPA under sorted access is less than or equal to p_i. Therefore, BPA stops with a number of sorted accesses less than or equal to TA. □

Lemma 2. *The number of random accesses done by BPA is always less than or equal to that of TA.*

Proof. The number of random accesses done by both BPA and TA is equal to the number of sorted accesses multiplied by *(m-1)* where m is the number of lists. Thus, the proof is implied by Lemma 1. □

Using the two above lemmas, the following theorem compares the execution cost of BPA and TA.

Theorem 2. *The execution cost of BPA is always less than or equal to that of TA.*

Proof. Considering the definition of execution cost, the proof is implied using Lemma 1 and Lemma 2. □

Lemmas 1 and 2 show that BPA always stops as early as TA. But how much faster than TA can it be? In the following, we answer this question. Assume that when BPA stops, its position in all lists is u. Then, during its execution, BPA has seen $u*m$ positions in each list, *i.e.* u positions under sorted access and $u*(m-1)$ under random access. If these are the positions from 1 to $u*m$, then the best position in each list is the $(u*m)th$ position. In other words, the best position can be m times greater than the position under sorted access. Based on this observation, we may conclude that BPA can stop at a position which is m times lower than TA. However, we did not find any case where this happens. Instead, we can prove that there are cases where BPA stops at a position which is *(m-1)* times smaller than TA. In other words, the number of sorted accesses done by BPA can be *(m-1)* times lower than TA. This is shown by the following lemma.

Lemma 3. *Let m be the number of lists, then the number of sorted accesses done by BPA can be (m-1) times lower than that of TA.*

Proof. To prove this lemma, it is sufficient to show that there are databases over which the number of sorted accesses done by BPA is *(m-1)* times lower than that of TA. In other words, under sorted access, BPA stops at a position which is *(m-1)* times lower than the position at which TA stops. Let δ be the value of TA's threshold at the moment when it stops. For each list L_i, let p_i be the position (under sorted access) at which TA stops. Without loss of generality, we assume $p_1=p_2=\ldots=p_m=j$, i.e. when TA stops its position in all lists is j. For simplicity assume that $j=(m-1)*u$ where u is an integer. Consider all cases where the two following conditions hold:

1) Each of the top-k answers have a local score at a position which is less than or equal to $j/(m-1)$, *i.e.* each of the top-k answers are seen under sorted access at a position which is less than or equal to $j/(m-1)$.

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

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

n_s = (number of seen positions in $[1..(j/(m-1))]$) + (number of seen positions in $[((j/(m-1) + 1) .. j]$)

After replacing the number of seen positions, we have:

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

After simplifying the right side of the equation, we have $n_s=m*j$. Thus, when BPA is at position $j/(m-1)$, it has seen all positions in interval $[1 .. j]$ in all lists. Therefore, the best position in each list is at least j. Hence, the best positions overall score, *i.e.* λ, is higher than or equal to the value of TA's threshold at position j, *i.e.* δ. In other words, we have $\lambda \geq \delta$. Since at position $j/(m-1)$, all top-k answers are in the set Y (see the first condition above) and their scores are less than or equal to δ (*i.e.* this is enforced by TA's stopping mechanism), the score of all data items involved in Y is less than or equal to λ. Thus, BPA stops at $j/(m-1)$,

---

[1] Choosing one of the corresponding local scores at a position greater than j allows us to adjust the local scores of top-k answers such that their overall scores do not get higher than TA's threshold at a position smaller than j, *i.e.* TA does not stop before j.

i.e. at a position which is *(m-1)* times lower than the position of TA. □

Lemma 4. *Let m be the number of lists, then the number of random accesses done by BPA can be (m-1) times lower than that of TA.*

Proof. Since the number of random accesses done by both BPA and TA is proportional to the number of sorted accesses, the proof is implied by Lemma 3. □

The following theorem shows that the execution cost of BPA can be *(m-1)* times lower than that of TA.

Theorem 3. *Let m be the number of lists, then the execution cost of BPA can be (m-1) times lower than that of TA.*

Proof. The proof is implied by Lemma 3 and Lemma 4. □

Example 3 (*i.e.* the database shown in Figure 1) is one of the cases where the execution cost of BPA is *(m-1)* times lower than TA. In that example, $m=3$ and TA stops at position 6, whereas BPA stops at position 3, *i.e.* *(m-1)* times lower than TA. For TA, the total number of sorted accesses is $6*3=18$ and the number of random accesses is $18*2=36$, *i.e.* for each sorted access *(m-1)* random accesses. With BPA, the number of sorted accesses and random accesses is $3*3=9$ and $9*2=18$, respectively.

5 BPA2

Although BPA is quite efficient, it still does redundant work. One of the redundancies with BPA (and also TA) is that it may access some data items several times under sorted access in different lists. For example, a data item, which is accessed at a position in a list through sorted access and thus accessed in other lists via random access, may be accessed again in the other lists by sorted access at the next positions. In this section, based on BPA, we propose BPA2, an algorithm which avoids re-accessing data items via sorted or random access. BPA2 is much more efficient than BPA. As we will show, the number of accesses to the sorted lists done by BPA2 can be about *(m - 1)* times lower than that of BPA.

5.1 Algorithm

Let *direct access* be a mode of access that reads the data item which is at a given position in a list. Recall from the previous section that the best position *bp* in a list is the greatest seen position of the list such that any position between 1 and *bp* is also seen. Then, BPA2 works as follows:

1. For each list L_i, let bp_i be the best position in L_i. Initially set $bp_i=0$.

2. For each list L_i and in parallel, do direct access to position $(bp_i + 1)$ in list L_i. As a data item *d* is seen under direct access in some list, do random access to the other lists to find *d*'s local score in every list. Compute the overall score of *d*. Maintain in a set *Y* the *k* seen data items whose overall scores are the highest among all data items seen so far.

3. If a direct access or random access to a list L_i changes the best position of L_i, then along with the local score of the accessed data item, return also the local score of the data item which is at the best position. Let $s_i(bp_i)$ be the local score of the data item which is at the best position in list L_i.

4. Let *best positions overall score* be $\lambda = f(s_1(bp_1), s_2(bp_2), ..., s_3(bp_m))$. If *Y* involves *k* data items whose overall scores are higher than or equal to λ, then stop doing sorted access to the lists. Otherwise, go to 1.

Position	List 1		List 2		List 3		$f = s_1 + s_2 + s_3$			
	Data item	Local score	Data item	Local score	Data item	Local score	Sum of local scores		Data item	Overall Score
1	d_1	30	d_2	28	d_3	30	88		d_1	65
2	d_4	28	d_6	27	d_5	29	84		d_2	65
3	d_9	27	d_7	25	d_8	28	80		d_3	70
4	d_3	26	d_5	24	d_4	27	77		d_4	68
5	d_7	25	d_9	23	d_2	26	74		d_5	63
6	d_8	24	d_1	22	d_6	25	71		d_6	66
7	d_{11}	17	d_{14}	20	d_{13}	15	52		d_7	61
8	d_6	14	d_3	14	d_1	13	41		d_8	64
9	d_2	11	d_4	13	d_9	12	36		d_9	62
10	d_5	10	d_8	12	d_7	11	33		…	…
…	…	…	…	…	…	…	…		…	…

Figure 2. Example database over which the number of accesses to the lists done by BPA2 is about $1/(m-1)$ that of BPA.

5. Return Y.

At each time, BPA2 does direct access to the position which is just after the best position. This position, *i.e.* $bp_i + 1$, is always the smallest unseen position in the list.

BPA2 has the same stopping mechanism as BPA. Thus, they both stop at the same (best) position. In addition, they see the same set of data items, *i.e.* those that have at least one local score before the best position in some list. Thus, they see the same set of positions in the lists.

However, there is a main difference between BPA2 and BPA. With BPA some seen positions of a list may be accessed several times, *i.e.* up to m times, but with BPA2 each seen position of a list is accessed only once. The reason is that BPA2 does direct access to the position which is just after the best position and this position is always an unseen position in the list.

Theorem 4. *No position in a list is accessed by BPA2 more than once.*

Proof. Implied by the fact that BPA2 always does direct access to an unseen position, *i.e.* $bp_i + 1$, so no seen position is accessed via direct access, and thus by random access. □

5.2 Correctness and cost analysis

The following theorem provides the correctness of BPA2.

Theorem 5. *If the scoring function is monotonic, then BPA2 correctly finds the top-k answers.*

Proof. Since BPA2 has the same stopping mechanism as BPA, the proof is similar to that of Theorem 1 which proves the correctness of BPA. □

In many systems, in particular distributed systems, the total number of accesses to the lists (composed of sorted/direct and random accesses) is a main metric for measuring the cost of a top-k query processing algorithm. Below, using two theorems we compare BPA and BPA2 from the point of the view of this metric.

Theorem 6. *The number of accesses to the lists done by BPA2 is always less than or equal to that of BPA.*

Proof. BPA and BPA2 access the same set of positions in the lists. However, BPA2 accesses each of these positions only once, but BPA may access some of the positions more than once. Therefore, the number of accesses to the lists by BPA is less than or equal to BPA. □

Theorem 7. *Let m be the number of lists, then the number of accesses to the lists done by BPA2 can be about $(m-1)$ times lower than that of BPA.*

Proof. To do the proof, we show that there are databases over which the number of accesses done by BPA is about $(m-1)$ times higher that of BPA2. For each list L_i, let bp_i be the best position at which BPA stops. Without loss of generality, we assume $bp_1=bp_2=\ldots=bp_m=j$, *i.e.* when BPA stops, the best position in all lists is j. For simplicity, assume that $j-1=(m-1)*u$ where u is an integer. We know that BPA2 stops at the same best position as BPA, so it also stops at j. Consider all databases at which the following condition holds:

1) If a data item is at a position in interval [1 .. j] in any list L_i, then $m-2$ of its corresponding local scores in other lists are at positions which are in interval [1 ..

(j-1)], and one[1] of its corresponding local scores is in a position higher than *j*.

The above condition assures that BPA does not see the data items which are at position *j* by a random access. Thus, it continues doing sorted access until the position *j*. In all databases that hold the above condition, we can argue as follows. Let n_d be the number of distinct data items which are in interval [1 .. (j-1)]. Since the total number of positions in interval [1 .. (j-1)] is m∗ (j-1), *i.e. j-1* times the number of lists, and each distinct data item occupies *(m-1)* positions in interval [1 .. (j-1)] (see the above condition), we have n_d= m∗ (j-1)/(m-1). In other words, we have $n_d = m*u$. BPA2 sees each distinct data item by doing one direct access. It also does *m* direct accesses at position *j*, *i.e.* one per list. Thus, BPA2 does a total of *(u+1)∗m* direct accesses. After each direct access, BPA2 does *(m-1)* random accesses, thus a total of *(u+1)∗m∗(m-1)* random accesses. Therefore, the total number of accesses done by BPA2 is $n_{bpa2} = (u+1)*m^2$. BPA sees all positions in interval [1 .. j] by sorted access, thus a total of *(j∗m)* sorted accesses. After each sorted access, it does *(m-1)* random accesses, thus a total of *j∗m∗(m-1)* random accesses. Therefore, the total number of accesses done by BPA is $n_{bpa} = (j)*m^2$. By comparing n_{bpa2} and n_{bpa}, we have $n_{bpa} = n_{bpa2} * (j / (u+1)) \approx n_{bpa2} * (m-1)$. □

As an example, consider the database (*i.e.* the three sorted lists) shown in Figure 2, and suppose *k=3* and the scoring function computes the sum of the local scores. If we apply BPA on this example, it stops at position 7, so it does 7∗3 sorted accesses and 7∗3∗2 random accesses. Thus, the total number of accesses done by BPA is $n_{bpa} = 63$. If we apply BPA2, it does direct access to positions 1, 2, 3 and 7 in all lists, so a total of 4∗3 direct accesses and 4∗3∗2 random accesses. Thus, the total number of accesses done by BPA2 is $n_{bpa2} = 36$. Therefore, we have $n_{bpa} \approx 2 * n_{bpa2}$.

6   Managing best positions

After each access to a list, BPA and BPA2 need to determine the best position in the list. A simple method for managing the best positions is to maintain the seen positions in a set. Then finding the best position is done by scanning the set and for each position *p*, verifying if all positions which are less than *p* belong to the set. This method is not efficient because finding the best position is done in $O(u^2)$ where *u* is the number of seen positions. Note that in the worst case, *u* can be equal to *n*, *i.e.* the number of data items in the list.

In this section, we address the problem of managing best positions in the lists and propose several efficient techniques. Each technique takes advantage of one of the following data structures: Bit array, Priority Queue, Bloom filter and B⁺tree. For each technique, we study its performance in terms of access time, space requirement and accuracy.

---

[1] This allows us to adjust the local scores of top-k answers such that BPA does not stop at a position smaller than *j*.

6.1 Bit array

In this approach, to remember the positions which are seen, each list owner uses an array of *n* bits where *n* is the size of the sorted list. Initially, all bits of the array are set to 0. There is a variable *bp* which points to the best position. Initially *bp* is set to 1. Let B_i be the bit array which is used by the owner of list L_i. After doing an access to a data item that is at position *j* in L_i, the following instructions are done by the list owner for determining the new best position:

```
Bᵢ[j] := 1;

While ((bp < n) and (Bᵢ[bp + 1] = 1))
do

    bp := bp + 1;
```

The total time needed for determining the best positions during the execution of the top-k query is *O(n)*, *i.e. bp* can be incremented up to *n*. Let *u* be the total number of seen positions in the list L_i, then the average time for determining the best position after each access is *O(n/u)*. The space needed for this approach is an array of *n* bits plus a variable, which is typically very small.

6.2 Priority queue

In this approach, we take advantage of priority queues for determining best positions. A priority queue *Q* usually has three main operations: 1) Insert(x) for adding a new element *x* to the queue; 2) Min() that returns the element with lowest priority from the queue; 3) Delete-Min() that removes from queue the element whose priority is the lowest.

For maintaining the seen positions of each sorted list, we use a priority queue. Let Q_i be the priority queue that maintains the seen positions of a list L_i. Let *bp* be a variable that points to the best position in list L_i. Initially we set *bp=0*. When a position *p* is seen in L_i, *p* is inserted to Q_i, and the following instructions are performed in order to update the best position:

```
While (Qᵢ.Min() = (bp + 1)) do

{

    bp := bp + 1;

    Qᵢ.DeleteMin();

}
```

One of the main advantages of this approach is that it maintains only the seen positions which are greater than the best position. Therefore, all positions which are lower than the current position under sorted access are deleted from the queue. In the best case, the space used by this approach is *O(1)*, i.e. when the best position is near to the position under sorted access. Let *u* be the number of seen positions in L_i. At least *1/m* of these positions are seen under sorted access. Thus, in the worst case, the number of positions which are maintained by the queue is *u∗(1 – (1/m))*.

Table 1. Performance of techniques for best position management (*n*: number of data items in each sorted list; *u*: number of positions seen by BPA).

Technique	Execution time per access	Space requirement	Accuracy
Bit array	$O(n/u)$	$O(n)$	100%
Priority queue	$O(\log \log n)$	$O(u)$	100%
Bloom filter	$O(1)$	$O(n)$	probabilistic
B$^+$tree	$O(\log u)$	$O(u)$	100%

The execution time of this approach depends on the data structure which is used for implementing the priority queue. A simple data structure that is typically used for implementing a priority queue is the heap tree [12]. Using structure heap tree, the operations Insert() and DeleteMin() are done in $O(\log n)$ where *n* is the number of queue's members, and the execution time of Min() is constant. Therefore, by using heap trees the average time per access is $O(\log u)$ where *u* is the number of seen positions. For the cases where the members of the priority queue are integer numbers, which is our case, there are more efficient data structures. For example, when the members are chosen from a set $\{1, 2, \ldots, C\}$, then the data structure proposed in [38] supports all three operations of priority queue in $O(\log \log C)$. Therefore, by using this data structure, the average time for determining best positions is $O(\log \log n)$ where n is the size of the sorted lists.

6.3 Bloom filter

In this section, we show how we can use Bloom filters in order to determine best positions. Bloom filter [6] is a probabilistic data structure that has two distinguishing features: (1) represent in a compact way the contents of a set and, (2) efficiently test whether a given item is a member of the set.

Briefly, a Bloom filter is an array of *b* bits, initially all set to 0. For representing a set $S = \{s_1, s_2, \ldots, s_n\}$ of *n* elements, the Bloom filter uses a set of *h* independent hash functions $f_1, \ldots, f_h$ with range $\{0, \ldots, b-1\}$, and for each element $s \in S$, the bits $f_i(s)$ are set to 1 for $1 \le i \le h$. A bit in the Bloom filter can be set to 1 multiple times, but only the first change has an effect. To check if an item *s* is in *S*, we check whether all $f_i(s)$ are set to 1, for $1 \le i \le h$. If not, then clearly *s* is not a member of *S*. If all $f_i(s)$ are set to 1, we assume that *s* is in *S*, although there is a small probability that we are wrong, i.e. this is referred to as a "false positive". Let IsMember(BF, *s*) be a function that returns true if *s* is assumed to be a member of the Bloom filter BF, otherwise it returns false.

In this approach, the seen positions of each sorted list L_i are represented by a Bloom filter, e.g. BF_i. A variable *bp* points to the best position in L_i. Initially *bp* is set to 0. After accessing a position *p* in L_i, *p* is inserted to BF_i. Then the following instructions are executed in order to determine the new best position:

```
While (IsMember(BF_i, bp + 1) = true)
    bp := bp + 1;
```

The advantage of using a Bloom filter is that it is efficient both in space and execution time. The time needed for executing the function IsMember is $O(h)$ where *h* is the number of hash functions used by the Bloom filter. Let *u* be the number of accesses to L_i, the total time for determining best positions is $O(h*u)$, thus the average time per access is $O(h)$. The space needed for this approach is an array of *b* bits.

The disadvantage of Bloom filters is the possibility of false positives. The probability of a false positive is equal to $PFP = (1 - e^{-hn/b})^h$ [29]. By well adjusting the parameters *h* and *b* we can minimize the possibility of false positives. Suppose we are given *b* and *n* and we want to minimize the probability of false positives by choosing an optimal value for the number of hash functions *h*. For this, we must compute the derivative of PFP with respect to *h*, and set it to zero, i.e. $d(1 - e^{-hn/b})^h / d\ h = 0$. By solving this equation, the minimum value for PFP is obtained when the number of hash functions is equal to $h = (\ln 2)*(b/n)$ [29]. Using this number of hash functions, the minimum value for the probability of false positives is $PFP = (1/2)^h \approx (0.6185)^{b/n}$. By increasing *b*, i.e. the number of bits of the Bloom filter, we can arbitrarily minimize the probability of false positives. To have a very small PFP, we have to choose a large value for *b*, e.g. if we want to have FPF <0.0001 then we must set *b* to at least $20*n$. Therefore, to provide a low false positive probability for the Bloom filter, we have to choose values for *b* which are much greater than *n*. Therefore, although the execution cost of the Bloom filter approach is much better than that of the bit array approach, its space requirement is higher.

6.4 B$^+$tree

In this approach, each list owner uses a B$^+$tree for maintaining the seen positions. In a B$^+$tree, all data are saved in the leaves, and the leaf nodes are at the same level, so any operation of insert/delete/lookup is logarithmic in the number of data items. The leaf nodes are also linked together as a linked list. Let *c* be a cell of the linked list, *i.e.* a leaf of the B$^+$tree, then

there is a pointer *c.next* that points to the next cell of the linked list. Let *c.element* be a variable that maintains the data in *c*.

Let BT_i be the B$^+$tree which the owner of list L_i uses for maintaining the seen positions of L_i. The list owner also uses a pointer *bp* that points to the cell, *i.e.* leaf of BT_i, which maintains the seen position which is the best position. After an access to a position *p* in L_i, the list owner adds *p* to BT_i. Then, it performs the following instructions to determine the new best position:

```
While     ((bp.next     ≠     null)     and
(bp.next.element = bp.element + 1)) do
bp := bp.next;
```

The above instructions assure that the pointer *bp* points always to the cell that maintains the best position. Let *u* be the total number of accesses to the list L_i during the execution of the query, then the total time for determining the best positions is *O(u)*, *i.e.* in the worst case *bp* moves from the head to the end of the linked list. Thus, the average time per access is *O(1)*. The time needed for adding a seen position to the B$^+$tree is *O(log u)*. Therefore, with the B$^+$tree approach, the average time for storing a seen position and determining the best position is *O(log u)*. The space needed for this approach is *O(u)*.

6.5 Summary

Table 1 summarizes the performance of the four techniques we propose for managing best position. In this table, *n* is the total number of data items in each sorted list, and *u* is the number of positions which are seen by BPA (or BPA2) during their execution. Note that although the space requirement for the Bloom Filter approach is *b*, i.e. the number of bits of the filter, to have a low false positive probability, b must be higher than *n*. This is why in this table, the space requirement of Bloom filter is *O(n)*.

7 Instance optimality

In this section, we discuss instance optimality which is considered as one of the main features of the TA algorithm. We first give the definition of instance optimality as in [16]. Then, we review the proof of TA's instance optimality done in [16] and show that if the positions of the data items are taken into account, then one of the main arguments of the instance optimality proof gets incorrect. In other words, we show that in this case TA may not be instance optimal.

Instance optimality is defined as follows [16]. Let *AL* be a class of algorithms, *DB* a class of databases, and *cost(A, D)* be the execution cost incurred by running algorithm *A* over database *D*. An algorithm $A \in AL$ is instance optimal over *AL* and *DB* if for every $B \in AL$ and every $D \in DB$ we have:

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

The above equation says that there are two constants c_1 and c_2 such that $cost(A, D) \leq c_1 * cost(B, D) + c_2$ for every choice of $B \in AL$ and $D \in DB$. The constant c_1 is called the *optimality ratio* of *A*.

In [16], Theorem 6.1 claims that TA is instance optimal over the class of all databases, and the class of all deterministic top-k query processing algorithms, *i.e.* those that do not make lucky guesses. This theorem assumes that *m*, i.e. the number of sorted lists, is a constant. Thus, although the execution cost of BPA can be *(m-1)* times better than TA, if *m* is a constant then we can not say that BPA is an example showing the incorrectness of TA's instance optimality.

Position	List 1		List 2		List 3		$f = s_1 + s_2 + s_3$			
	Data item	Local score s_1	Data item	Local score s_2	Data item	Local score s_3	Threshold		Data item	Overall Score
1	d_1	30	d_2	28	d_3	30	88		d_1	65
2	d_4	28	d_6	27	d_5	29	84		d_2	63
3	d_9	27	d_7	25	d_8	28	80		d_3	70
4	d_{11}	27	d_{11}	25	d_{11}	28	80		d_4	66
5	d_3	26	d_5	24	d_4	25	75		d_5	70
6	d_7	25	d_9	23	d_2	24	72		d_6	60
7	d_8	23	d_1	21	d_6	19	63		d_7	61
8	d_5	17	d_8	20	d_{13}	15	52		d_8	71
9	d_6	14	d_3	14	d_1	14	42		d_9	62
10	d_2	11	d_4	13	d_9	12	36		d_{11}	80
11	d_{14}	9	d_{14}	12	d_7	11	33		…	…
…	…	…	…	…	…	…	…		…	…

(a)   *(b)*   *(c)*

Figure 3. This database, say D', is just like the database of Figure 1, say D, except that data item d_{11}, which is not seen by BPA over D, is deposited just below the position 3 which is the stopping position of BPA over D. If we apply BPA over D' its output is different of its output over D.

However, by reviewing the proof of TA's instance optimality (i.e. Theorem 6.1 in [16]), we observed that the proof is incorrect if we memorize the positions of seen data items. In the following, we show what is wrong with that proof.

Let us first review the proof of Theorem 6.1 in [16] which proceeds as follows. Assume A is a deterministic top-k algorithm that finds correctly the top-k data items of a database D. Let d be the position (depth in [16]) at which A stops, and a be the number of distinct data items which are seen by A before its stop. Let Y be the set of the outputs of A (i.e. the top-k data items). Let τ_A be the value of threshold at d. The proof is done by showing that TA stops at most at position $(a + k)$. For this, it considers the two following cases: 1) The number of data items, which are not seen by A during its execution, is at most k; 2) There are at least $(k + 1)$ data items which are not seen by A during its execution. In the first case, it can be easily shown that TA stops at most at $(k+a)$, i.e. in the worst case TA reads all data items of the database. For the second case, the proof of Theorem 6.1 claims that every member of Y has an overall score higher than or equal to τ_A, and by using this claim it shows that TA stops at most at position $(a + k)$. To prove this claim, it argues as follows. Since at least $(k + 1)$ data items which are

not seen by A, there is some unseen data item v that is not in the output[1] of A, i.e. $v \notin Y$. Let x_i be the score of the last data item seen by A under sorted access in list L_i, for $1 \le i \le m$. Define a database D' to be just like D, except that data item v has a score x_i in list L_i, for $1 \le i \le m$. In D', put v in the list L_i below all other data items whose score in L_i is x_i (for $1 \le i \le m$). Then, the proof of Theorem 6.1 in [16] argues that: *Algorithm A performs exactly the same, and in particular gives the same output, for databases D and D'*. Therefore, algorithm A does not have v in its output for database D'. Since the overall score of v in D' is equal to τ_A, the correctness of A implies that the overall score of any member of A is at least that of v, and therefore at least τ_A.

As we see, the proof *is based on the argument that if we change the position of a data item v, which is not seen by an algorithm A, then the behaviour of A over the new database remains unchanged*, and it gives the

---

[1] Notice that there may be algorithms whose output involves data items which are not unseen by them.

same output. However, this is not true with BPA as shown as follows. Let L_i be a list, and d be the position at which BPA stops. As we know, the best position in L_i, say bp_i, is always higher than or equal to the position at which BPA stops, i.e. $bp_i \geq d$. Let p be the position of an unseen data item v in the list Li. Then the definition of best position implies that $bp_i < p$. If we change the position of v to a position q such that $d \leq q \leq bp_i$, then bp_i changes to $(q - 1)$, i.e. because q is an unseen position and all positions before it are seen. This modification in the best position may modify the best position overall score. Thus BPA may not stop at d and generate a different output. Let us illustrate this by the following example.

Example 4. As an example, suppose a top-3 query, i.e. $k=3$ such that the scoring function computes the sum of the local scores. Consider the database (*i.e.* the three sorted lists) shown in Figure 3, which we denote by D'. This database is just like the database of Figure 1, say D, except that data item d_{11}, which is not seen by BPA over D, is put just below the position 3 which is the stopping position of BPA over D. The local scores of d_{11} are exactly the same as the local scores which are at position 3 in the lists. If we apply BPA on D', it stops at position 4, and its output is $Y'=\{d_{11}, d_8, d_3\}$ which is different from the output over database D, *i.e.* $Y=\{d_3, d_5, d_8\}$. Thus, we created the database D' exactly in the same way as mentioned in the proof of Theorem 6.1 in [16], but over D' both stopping position and output of BPA are different from those over D.

As shown above, there are deterministic top-k algorithms (e.g. BPA) whose behaviour changes by changing the position of data items, which are not seen by the algorithms during their execution. As a result, Theorem 6.1 in [16] which claims the instance optimality of TA over all deterministic algorithms has to be revised.

8   Performance evaluation

In the previous sections, we analytically compared our algorithms with TA, *i.e.* BPA directly and BPA2 indirectly. In this section, we compare these three algorithms through experimentation over randomly generated databases. The rest of this section is organized as follows. We first describe our experimental setup. Then, we compare the performance of our algorithms with TA by varying experimental parameters such as the number of lists, *i.e.* m, the number of top data items requested, *i.e.* k, and the number of data items of each list, *i.e.* n. Finally, we summarize the performance results.

8.1   Experimental setup

We implemented TA, BPA and BPA2 in Java[1]. 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 database, the positions of a data item in any two lists are independent of each other.

To generate this database, the scores of the data items in each list are generated using a uniform random generator, and then the list is sorted. This is our default setting. With Gaussian database, the positions of a data item in any two lists are also independent of each other. To generate this 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 also use correlated databases, *i.e.* databases where the positions of a data item in the lists are correlated. We use this type of database for taking into account the applications where there are correlations among the positions of a data item in different lists. In real-world applications, there are usually such correlations [28]. Inspired from [28], we use a correlation parameter α ($0 \leq \alpha \leq 1$), and we generate the correlated databases as follows. For the first list, 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 create databases with stronger or weaker correlations. 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 [41] with the Zipf parameter $\theta = 0.7$. The Zipf law states that the score of an item in a ranked list is inversely proportional to its rank (position) in the list. It is commonly observed in many kinds of phenomena, *e.g.* the frequency of words in a corpus of natural language utterances.

Our default settings for different experimental parameters are as follows. In our tests, the default number of data items in each list is 100,000. Typically, users are interested in a small number of top answers, thus unless otherwise specified we set $k=20$. Like many previous works on top-k query processing, *e.g.* [8], we use a scoring function that computes the sum of the local scores. In most of our tests, the number of lists, *i.e.* m, is a varying parameter. When m is a constant, we set it to 8 which is rather small but quite sufficient to show significant performance gains of our algorithms. Note that, in some important applications such as network monitoring [8], m can be much higher.

To evaluate the performance of the algorithms, we measure the following metrics.

1) Execution cost. As defined in Section 2, the execution cost is computed as $c = a_s*c_s + a_r*c_r$ where a_s is the number of sorted accesses that an algorithm does during execution, a_r is the number of random accesses, c_s is the cost of a sorted access, and c_r is the cost of a random access. For the BPA2 algorithm, we consider each direct access equivalent to a random access. For each sorted access we consider one unit of cost, *i.e.* we set $c_s = 1$. For the cost of each random

---

[1]   The   code   is   available   at   http://www-sop.inria.fr/teams/zenith/pmwiki/pmwiki.php/Other/BPAalgorithms

access, we set $c_r = log\ n$ where n is the number of data items, *i.e.* we assume that there is an index on data items such that each entry of index points to the position of data item in the lists. The execution time which we consider here is a good metric for comparing the performance of the algorithms in a centralized system. For distributed systems, we use the next metric.

2) Number of accesses. This metric measures the total number of accesses to the lists done by an algorithm during execution. It involves the sorted, direct and random accesses. In distributed systems, particularly in the cases where message size is small (which is the case of our algorithms), the main cost factor is the number of messages communicated between nodes. The number of messages, which our algorithms (and TA) communicate between the query originator and list owners in a distributed system, is proportional to the number of accesses done to the lists. Thus, the number of accesses is a good metric for comparing the performance of the algorithms in distributed systems. For TA and BPA, the number of accesses is also a good indicator of their stopping position under sorted access, *i.e.* the number of accesses is m^2 multiplied by the stopping position.

3) Response time. This is the total time (in millisecond) that an algorithm takes for finding the top-k data items. We conducted our experiments on a machine with a 2.4 GHz Intel Pentium 4 processor and 2GB memory. In the code of BPA and BPA2, the best positions are managed using the Bit Array approach which is simpler than the B+-tree approach.

8.2 Performance results

8.2.1 *Effect of the number of lists*

In this section, we compare the performance of our algorithms with TA over the three database types while varying the number of lists.

Over the uniform database, with the number of lists increasing up to 18 and the other parameters set as in Section 8.1, Figures 4, 5 and 6 show the results measuring execution cost, number of accesses, and response time, respectively. The execution cost of BPA is much better than that of TA; it outperforms TA by a factor of approximately *(m+6)/8* for

$m>2$. BPA2 is the strongest performer; it outperforms TA by a factor of approximately *(m+1)/2* for *m>2*. On the second metric, *i.e.* number of accesses, the results are similar to those on execution cost. However, BPA2 outperforms TA by a factor which is (a little) higher than that for execution cost, *i.e.* about 1/m higher. The reason is that for measuring execution cost, we assume an expensive cost (*i.e.* log n units) for direct accesses which are done by BPA2. On response time, BPA2 (and BPA) outperforms TA by a factor which is a little lower than that on execution cost, just because of the time they need for managing the best positions.

Over the Gaussian database, with the number of lists increasing up to 18 and the other parameters set as in Section 8.1, Figures 7, 8 and 9 show the results for execution cost, number of accesses, and response time respectively. Over the Gaussian database, the performance of the three algorithms is a little better than their performance over the uniform database. BPA and BPA2 do much better than TA, and they outperform it by a factor close to that over the uniform database.

Overall, the performance results on the three metrics are qualitatively similar, in particular on execution cost and number of accesses. Thus, in the rest of this paper, we only report the results on execution cost.

Figures 10, 11 and 12 show the execution cost of the algorithms over three correlated databases with correlation parameter α set to 0.001, 0.01 and 0.1 respectively, and the other parameters set as in Section 8.1. Over these databases, the performance of the three algorithms is much better than that over Gaussian and uniform databases. In fact, the more correlated is the database; the lower is the execution cost of all three algorithms. The reason is that in a highly correlated database, the top-k data items are distributed over low positions of the lists, so the algorithms do not need to go much down in the lists, and they stop soon. However, due to their efficient stopping mechanism, BPA and BPA2 stop much sooner than TA.
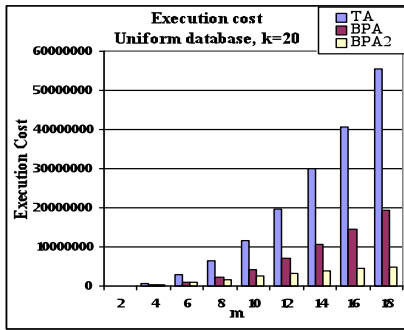
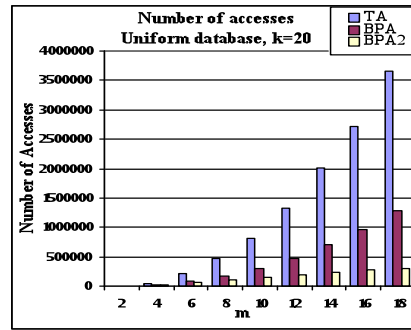**Figure 4.** *Execution cost vs. number of lists over uniform database*

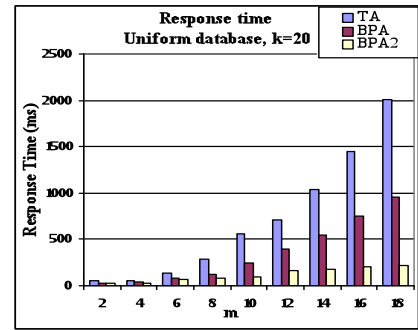**Figure 5.** *Number of accesses vs. number of lists over uniform database*

Figure 6. *Response time vs. number of lists over uniform database*

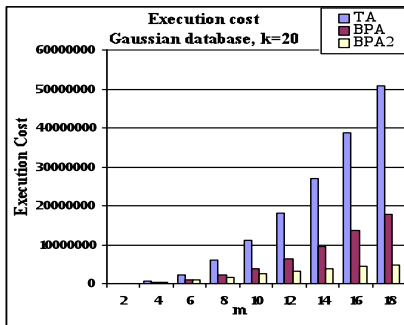**Figure 7.** *Execution cost vs. number of lists over Gaussian database*
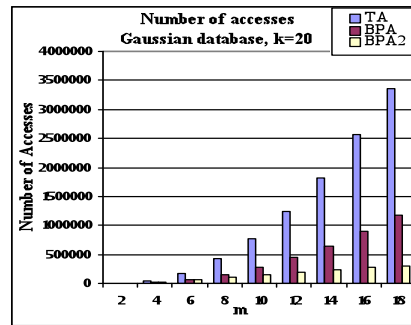

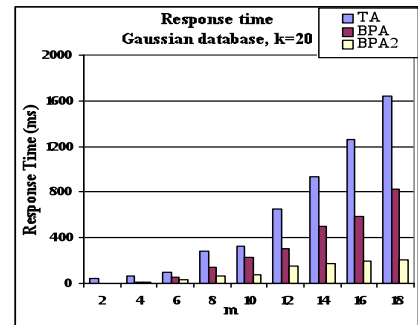**Figure 8.** *Number of accesses vs. number of lists over Gaussian database*

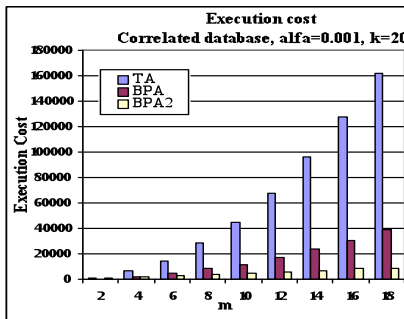**Figure 9.** *Response time vs. number of lists over Gaussian database*

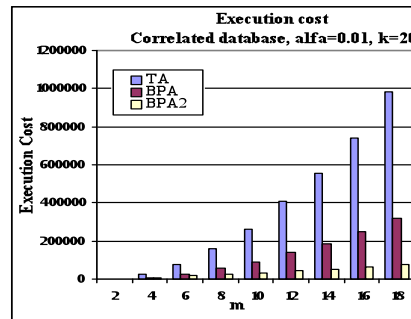**Figure 10.** *Execution cost vs. number of lists over correlated database with $\alpha=0.001$*

Figure 11. *Execution cost vs. number of lists over correlated database with $\alpha=0.01$*
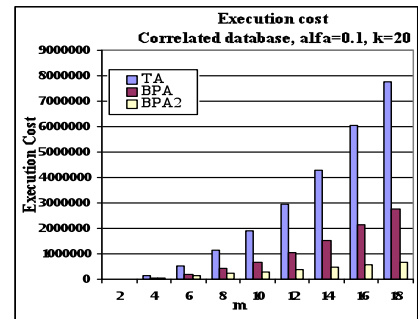

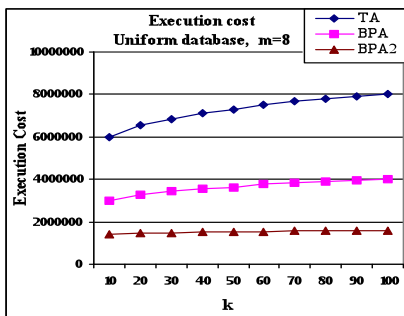**Figure 12.** *Execution cost vs. number of lists over correlated database with $\alpha=0.1$*

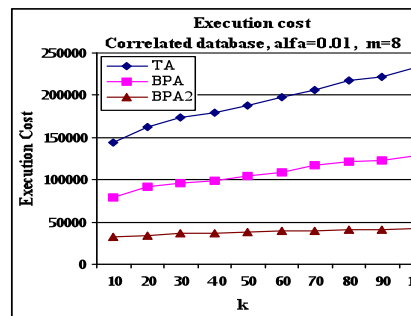**Figure 13.** *Execution cost vs. k over uniform database*

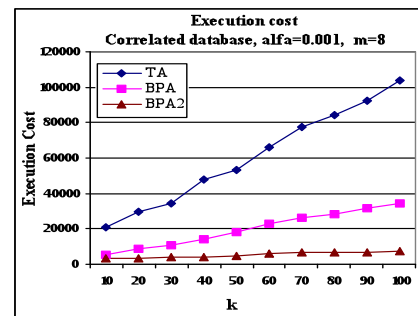**Figure 14.** *Execution cost vs. k over correlated database with $\alpha=0.01$*

Figure 15. *Execution cost vs. k over correlated database with $\alpha=0.001$*
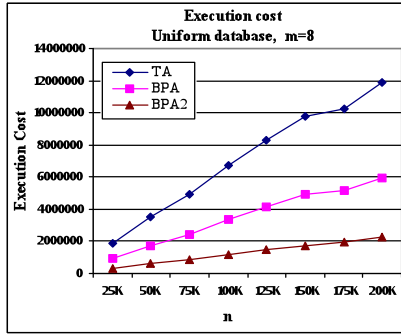
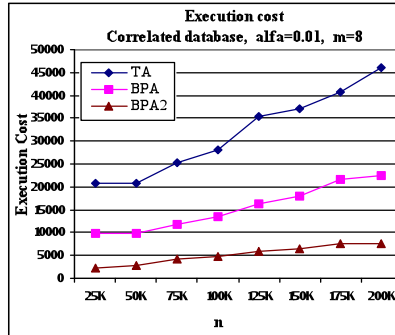**Figure 16.** *Execution cost vs. n over uniform database*

Figure 17. *Execution cost vs. n over correlated database with α=0.01*

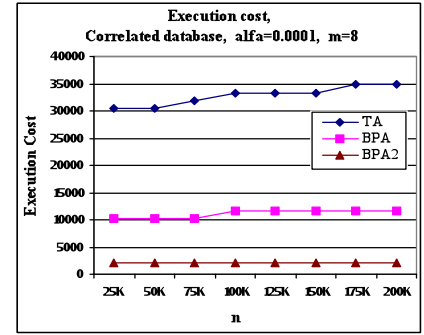**Figure 18.** *Execution cost vs. n over correlated database with α=0.0001*

8.2.2 Effect of k

In this section, we study the effect of k, *i.e.* the number of top data items requested, on performance. Figure 13 shows how execution cost increases over the uniform database, with increasing k up to 100, and the other parameters set as in Section 8.1. The execution cost of all three algorithms increases with k because more data items are needed to be returned in order to obtain the top-k data items. However, the increase is very small. The reason is that over the uniform database, when an algorithm (*i.e.* any of the three algorithms) stops its execution for a top-k query, with a high probability, it has seen also the $(k + 1)$th data item. Thus, with a high probability, it stops at the same position for a top-(k+1) query.

Figures 14 and 15 show how execution cost increases with increasing k over two correlated databases with correlation parameter set to $\alpha=0.01$ and $\alpha=0.001$ respectively. For the database with $\alpha=0.01$, *i.e.* the one which is less correlated, the impact of k is smaller. The reason is that when we run one of the three algorithms over a database with low correlation, it sees a lot of data items before stopping its execution. Thus, when it stops at a position for a top-k query, there is a high probability that it stops at the same position for a top-(k + 1) query. But, for a highly correlated database, this probability is lower because the algorithm sees a small number of data items before stopping its execution.

8.2.3 Effect of the number of data items

In this section, we vary the number of data items in each list, *i.e.* n, and investigate its effect on execution cost. Figure 16 shows how execution cost increases over the uniform database with increasing n up to 200,000, and with the other parameters set as in Section 8.1. Increasing n has a considerable impact on the performance of the three algorithms over a uniform database. The reason is that when we enlarge the lists and generate uniform random data for them, the top-k data items are distributed over higher positions in the list.

Figures 17 and 18 show how execution cost increases with increasing n over two correlated databases with correlation parameter set to $\alpha=0.01$ and $\alpha=0.001$ respectively, and the other parameters set as in Section 8.1. The results show that n has a smaller impact on a highly correlated database rather than a database with a low correlation.

8.2.4 Concluding remarks

The performance results show that, over all test databases and wrt all the metrics, the performance of our algorithms is much better than that of TA. For example, they show that wrt execution cost, BPA and BPA2 outperform TA by a factor of approximately $(m+6)/8$ and $(m+1)/2$ for $m>2$. Thus, as m increases, the performance gains of our algorithms versus TA increase significantly.

9 Related work

The problem of finding top-k data over sorted lists is a fundamental problem which can be used in both top-k selection and top-k join queries.

A first important paper is [14] which models the general problem of answering top-k queries using sorted lists, and proposes a simple yet efficient algorithm, called Fagin's algorithm (FA). The most efficient algorithm over sorted lists is the TA algorithm which was proposed by several groups[1] [15][17][31]. TA is simple, elegant and efficient [16] and provides a significant performance improvement over FA. We already discussed much TA in this paper. However, because of its stopping mechanism (based on the last seen scores under sorted access), TA can still perform useless work (see Section 3). The fundamental differences between BPA and TA are the following. BPA takes into account the positions and scores of the seen data whereas TA only takes into account their scores. Using information about the position of the seen data, BPA develops a more intelligent stopping mechanism that allows choosing a much better time to stop (such choice is correct as proved in Lemma 1). This allows BPA to gain much reduction in the number of sorted accesses and thus much reduction in the number of random accesses. Even if TA were keeping track of all seen data items, it could not stop at a smaller position under sorted access, because its threshold does not allow it.

---

[1] The second author of [15] first defined TA and compared it with FA at the University of Maryland in the Fall of 1997.

Several TA-style algorithms (*i.e.* extensions of TA) have been proposed, particularly for processing top-k selection queries in distributed environments, *e.g.* [5][13][28]. Overall, most of the TA-style algorithms focus on extending TA with the objective of minimizing communication cost of top-k query processing in distributed systems. For example, in the context of the KLEE framework [28], the authors propose an approximate TA-style algorithm aiming at gaining much reduction in communication cost at low result-quality penalties. In KLEE, the m sorted lists are distributed over m nodes, and each node divides its list into c cells and maintains statistical information describing the cells, *e.g.* lower, upper, average and frequency of local scores which fall in the cell. This information on cells contribute to reduce the number of local scores which should be communicated over the network.

The Three Phase Uniform Threshold (TPUT) [8] is an efficient algorithm to answer top-k queries in distributed systems. The algorithm reduces communication cost by pruning away ineligible data items and restricting the number of round-trip messages between the query originator and the other nodes. However, there are databases over which TPUT is not efficient. For example, if one of the lists has n data items with a fixed value that is just over the threshold of TPUT, then all data items must be retrieved by the query originator, while a more adaptive algorithm might avoid retrieving all n data items.

There have been multiple algorithms devoted to top-k join queries, i.e. top-k queries in which there are several joined relations. Most of these algorithms are designed based on the top-k selection algorithms, e.g. TA. In [20], the authors introduce an efficient top-k join algorithm and two *rank-join* operators that can be deployed in existing query execution interfaces. Proposed in [31], $J*$ is another efficient algorithm for processing 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. In [37], 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. In [24], the relational algebra is extended to support rank queries as a first-class construct. In [25], top-k join query processing is extended to aggregate queries. Approximate ranked query processing techniques, which are proposed in [7], deal with the cases where the system is not able to return a complete answer to the top-k query.

Recently, uncertain data management has received much attention in the database community (see [1] for a survey). One of the well-studied problems in uncertain databases is the evaluation of top-k queries that have more complex semantics than in exact databases. There have been multiple definitions for uncertain top-k queries (e.g. see [11][19][35]). In [35], two main first definitions, namely U-Top-k and U-kRank, have been proposed. U-Top-k returns the most probable top-k set that belong to possible worlds (possible databases), and U-kRank considers the winner in every individual rank. In [19], PT-k queries are defined that return the top-k sets whose probability of being one of the top-k results is higher than a given threshold. Another definition in [11] is based on expected rank, and returns the tuples whose expected rank in all possible worlds is less than k.

10 Conclusion

The most efficient algorithm proposed so far for answering top-k queries over sorted lists is the Threshold Algorithm (TA). However, TA may still incur a lot of useless accesses to the lists. In this paper, we proposed two algorithms which stop much sooner and thus are more efficient than TA.

First, we proposed the BPA algorithm whose stopping mechanism takes into account the seen positions in the lists. For any database instance (*i.e.* set of sorted lists), we proved that BPA stops at least as early as TA. We showed that the number of sorted/random accesses done by BPA is always less than or equal to that of TA, and thus its execution cost is never higher than TA. We also showed that the number of sorted/random accesses done by BPA can be *(m-1)* times lower than that of TA. Thus, its execution cost can be *(m-1)* times lower than that of TA.

Second, based on BPA, we proposed the BPA2 algorithm which is much more efficient than BPA. In addition to its efficient stopping mechanism, BPA2 avoids re-accessing data items via sorted and random access, without having to keep data at the query originator. We showed that the number of accesses to the lists done by BPA2 can be about *(m-1)* times lower than that of BPA.

Third, we proposed several techniques using different data structures for managing best positions in BPA and BPA2 and analyzed their performance in terms of access time, space requirements and accuracy.

Although the superiority of BPA against TA by a factor of $O(m)$ does not invalidate the TA's instance optimality (because it assumes that m is a constant), we showed that the existence of deterministic algorithms such as BPA proves that the main argument, which is used for proving the instance optimality of TA in [16], is incorrect. In other words, the proof of TA's instance optimality is incorrect, and thus TA may not be instance optimal.

To validate our contributions, we implemented our algorithms as well as TA as baseline. We evaluated the performance of the algorithms over both independent and correlated databases wrt three representative metrics (execution cost, number of accesses and response time). The performance evaluations show that, over all test databases and wrt all the metrics, our algorithms always outperform TA significantly. For example, wrt execution cost, BPA and BPA2 outperform TA by a factor of approximately *(m+6)/8* and *(m+1)/2* respectively (for $m>2$). *e.g.* for $m=10$, the factor is 2 and 5.5, respectively. Thus, as m increases, the performance gains of our algorithms versus TA increase significantly. Note that in some applications, the number of lists, *i.e.* m, is very large, *e.g.* it may range from a few tens to a few thousands [8]. For example, consider a network monitoring application that monitors the activities of the users of some specified IP locations. The specified locations may be numerous. For each location, the application maintains

a list of the accessed URLs ranked by their frequency of access. In this application, an interesting query for the network administrator is "what are the top-k popular URLs?".

References

[1] Aggarwal, C.C., S. Yu, P.: A Survey of Uncertain Data Algorithms and Applications. *IEEE Trans. Knowl. Data Eng.*, 21(5) (2009) 609-623.

[2] Akbarinia, R., Pacitti E., Valduriez, P.: Best position algorithms for top-k queries. *Int. Conf. on Very Large Databases (VLDB)*, 2007, pp. 495-506.

[3] Akbarinia, R., Pacitti E., Valduriez, P.: Reducing network traffic in unstructured P2P systems using Top-k queries. *Distributed and Parallel Databases*, 19(2) (2006) 67-86.

[4] Balke, W.-T., Nejdl, W., Siberski, W., Thaden, U.: Progressive distributed top-k retrieval in peer-to-peer networks. *Int. Conf. on Data Engineering (ICDE)*, 2005, pp. 174-185

[5] Bast, H., Majumdar, D., Schenkel, R., Theobald, M., Weikum, G.: IO-Top-k: index-access optimized top-k query processing. *Int. Conf. on Very Large Databases (VLDB)*, 2006, pp. 475-486.

[6] Bloom, B.: Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7) (1970) 422-426.

[7] Candan, K.S., Li, W.-S., Priya, M.L.: Similarity-based ranking and query processing in multimedia databases. *Data & Knowledge Engineering*, 35(3) (2000) 259-298.

[8] Cao, P., Wang, Z.: Efficient top-k query calculation in distributed networks. *ACM Symp. on Principles of Distributed Computing (PODC)*, 2004, pp. 206-215.

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

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

[11] Cormode, G., Li, F., Yi., K.: Semantics of Ranking Queries for Probabilistic Data and Expected Ranks. *Int. Conf. on Data Engineering (ICDE)*, 2009, pp. 305-316.

[12] Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to algorithms. MIT Press / McGraw-Hill (1990).

[13] Das, G., Gunopulos, D., Koudas, N., Sarkas, N.: Ad-hoc Top-k Query Answering for Data Streams. *Int. Conf. on Very Large Databases (VLDB)*, 2007, pp. 183-194.

[14] Fagin, R.: Combining fuzzy information from multiple systems. *J. of Computer and System Sciences*, 58 (1) (1999) 83-99.

[15] Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. *ACM Symp. on Principles of Database Systems (PODS)*, 2001, pp. 102-113.

[16] Fagin, R., Lotem, J., Naor, M.: Optimal aggregation algorithms for middleware. *J. of Computer and System Sciences*, 66(4) (2003) 614-656.

[17] Güntzer, U., Kießling, W., Balke, W.-T.: Towards efficient multi-feature queries in heterogeneous environments. *IEEE Int. Conf. on Information Technology, Coding and Computing (ITCC)*, 2001, pp. 622-628.

[18] Hou U, L., Mamoulis, N., Berberich, K., Bedathur, S. J.: Durable top-k search in document archives. ACM *Int. Conf. on Management of Data (SIGMOD),* 2010, pp. 555-566.

[19] Hua, M., Pei, J., Zhang, W., Lin, X.: Ranking queries on uncertain data: a probabilistic threshold approach. *ACM Int. Conf. on Management of Data (SIGMOD)*, 2008, pp. 673-686.

[20] Ilyas, I.F., Aref, W.G., Elmagarmid, A.K.: Supporting Top-k Join Queries in Relational Databases. *Int. Conf. on Very Large Databases (VLDB)*, 2003, pp. 754-765.

[21] Jin, C., Yi, K., Chen, L., Xu Yu, J., Lin, X.: Sliding-window top-k queries on uncertain streams. *VLDB Journal*, 19(3) (2010) 411-435.

[22] Kimelfeld, B., Sagiv, Y.: Finding and approximating top-k answers in keyword proximity search. *ACM Symp. on Principles of Database Systems (PODS)*, 2006, pp. 173-182.

[23] Koudas, N., Ooi, B.C., Tan, K.L., Zhang, R.: Approximate NN queries on streams with guaranteed error/performance bounds. *Int. Conf. on Very Large Databases (VLDB)*, 2004, pp. 804-815.

[24] Li, C., Chang, K. C.-C., Ilyas, I.F., Song, S.: RankSQL: Query Algebra and Optimization for Relational Top-k Queries. *ACM Int. Conf. on Management of Data (SIGMOD)*, 2005, pp. 131-142.

[25] Li, C., Chang, K. C.-C., Ilyas, I.F.: Supporting ad-hoc ranking aggregates. *ACM Int. Conf. on Management of Data (SIGMOD)*, 2006, pp. 61-72.

[26] Li, F., Yi, K., Le, W.: Top- queries on temporal data. *VLDB Journal*, 19(5) (2010) 715-733.

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

[28] Michel, S., Triantafillou, P., Weikum, G.: KLEE: A framework for distributed top-k query algorithms. *Int. Conf. on Very Large Databases (VLDB)*, 2005, pp. 637-648.

[29] Mitzenmacher, M.: Compressed bloom filters. *ACM Symp. on Principles of Distributed Computing (PODC)*, 2001, pp. 144-150.

[30] Mouratidis, K., Bakiras, S., Papadias, D.: Continuous monitoring of top-k queries over sliding windows. *ACM Int. Conf. on Management of Data (SIGMOD)*, 2006, pp. 635-646.

[31] Natsev, A., Chang, Y.-C., Smith, J. R., Li, C.-S., Vitter, J. S.: Supporting incremental join queries on ranked input. *Int. Conf. on Very Large Databases (VLDB)*, 2001, pp. 281-290.

[32] Persin, M., Zobel, J., Sacks-Davis, R.: Filtered document retrieval with frequency-sorted indexes. *J. of the American Society for Information Science*, 47(10), pp. 749-764 (1996)

[33] Re, C., Dalvi, N.N., Suciu, D.: Efficient Top-k Query Evaluation on Probabilistic Data. *Int. Conf. on Data Engineering (ICDE)*, pp. 886-895 (2007)

[34] Silberstein, A., Braynard, R., Ellis, C.S., Munagala, K., Yang, J.: A sampling-based approach to optimizing top-k queries in sensor networks. *Int. Conf. on Data Engineering (ICDE)* (2006)

[35] Soliman, M.A., Ilyas, I.F., Chang, K. C.-C.: Top-k Query Processing in Uncertain Databases. *Int. Conf. on Data Engineering (ICDE)*, 2007, pp. 896-905.

[36] Tran, T., Wang, H., Rudolph, S., Cimiano, P.: Top-k Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data. *Int.*

Conf. on Data Engineering (ICDE), 2009, pp. 405-416.

[37] Tsaparas, P., Palpanas, T., Kotidis, Y., Koudas, N., Srivastava, D.: Ranked Join Indices. *Int. Conf. on Data Engineering (ICDE)*, 2003.

[38] Van Emde Boas, P.: Preserving order in a forest in less than logarithmic time. *16th Annual Symposium on Foundations of Computer Science (FOCS)*, 1975, pp. 75-84.

[39] Wu, M., Xu, J., Tang, X., Lee, W-C.: Monitoring top-k query in wireless sensor networks. *Int. Conf. on Data Engineering (ICDE)*, 2006.

[40] Zhang, W., Lin, X., Zhang, Y., Pei, J., Wang, W.: Threshold-based probabilistic top- dominating queries. *VLDB Journal*, 19(2) (2010) 283-305.

[41] Zipf, G.K.: Human Behavior and the Principle of Least Effort. Addison-Wesley Press (1949).