2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing

An Experimental Evaluation of Aggregation Algorithms for Processing Top-*K* Queries*

Liang Zhu[1], Qin Ma[2, #], Weiyi Meng[3], Mingqian Yang[4], Fang Yuan[4]
[1]School of Computer Science and Technology, Hebei University, Baoding, Hebei 071002, China
[2]Department of Foreign Language Teaching and Research, Hebei University, Baoding, Hebei 071002, China
[3]Department of Computer Science, State University of New York at Binghamton, Binghamton, NY 13902, USA
[4]School of Mathematics and Information Science, Hebei University, Baoding, Hebei 071002, China
{zhu, maqin, yuanfang}@hbu.edu.cn; ymq655@163.com; meng@cs.binghamton.edu

***Abstract*—For processing top-*K* queries with monotone aggregation functions, the threshold algorithm (TA) and its family are important methods in many scenarios. From 1996 to 2003, Fagin et al. proposed a variety of TA-like algorithms such as the FA, TA, TAz, NRA and CA algorithms for different access methods as well as various data resources, but they did not report the experimental results of the TA-like algorithms in their seminal papers. Since then, some of the original TA-like algorithms have been implemented, improved or adapted in different situations and/or applications; however, the original algorithms have not been thoroughly compared and analyzed under the same experimental framework. To address this problem, in this paper, we carry out extensive experiments to measure the performance of the original aggregation algorithms and the slight adaptations of TA, TAz and NRA, and then we provide comprehensive surveys on the natures of the TA-like algorithms based on our experimental results.**

I. INTRODUCTION

From 1996 to 2003, Fagin et al. proposed the aggregation algorithms FA, TA, TAθ, TAz, NRA and CA in [5-8] in order to deal with top-*K* queries (also ranked, ranking, top-*k*, or top-*N* queries), where the TAθ is an approximation algorithm while the others are exact algorithms. Those algorithms that can be used in various scenarios are remarkably simple, database-friendly and powerful, which are called TA-like algorithms in this paper. In the seminal papers [5-8], Fagin et al. defined the TA-like algorithms, proved their correctness, presented the size of the buffer used by each algorithm, and discussed the optimality for each of the algorithms under certain assumptions; however, the experimental results of the algorithms are not reported in [5-8]. Notice that we address exact algorithms without discussing TAθ in this paper.

There are many variations and improvements of some of the original TA-like algorithms for various situations and/or applications [13]. For example, Nepal and Ramakrishna [17] and Güntzer et al. [9] presented their own algorithms independently that are equivalent to TA for processing queries over multimedia databases. Using index structures, [10] presented the MPro algorithm that ensures every probe performed is necessary for evaluating the top-*K* tuples. Tracking the "best positions" in each sorted list and reducing the number of accesses, [1] proposed techniques BPA and BPA2 to optimize the TA algorithm in [8]. [15] proposed the algorithm LARA to optimize the algorithm NRA by employing a lattice to reduce the computational cost of NRA. Building a table $\boldsymbol{R}'$ to maintain the sorted access order in the lists such that the grade computation of an object using sorted lists can be reduced by $\boldsymbol{R}'$, the STA method in [14] is an improvement of the TA algorithm. The previous research works focused on performance evaluations and comparisons of a small number of the TA-like algorithm(s) related to the proposal(s). However, the original TA-like algorithms have not been thoroughly compared under the same experimental framework. We address this problem and make the following contributions in this paper: (1) Using the same experimental framework, we provide a performance evaluation of the aggregation TA-like algorithms FA, TA, TAz, NRA, CA, TA1, TAz1, and NRA1. (2) Based on a variety of real-world and synthetic datasets with low, moderate and high dimensions, we analyze the eight TA-like algorithms and compare them with the baseline Naïve Algorithm (NA) [8] through extensive experiments. (3) We report comprehensive surveys on the natures of the TA-like algorithms based on our experimental results.

This paper is organized as follows. Section II introduces some notations and concepts. Section III presents the algorithms TA1, TAz1, and NRA1. Section IV provides the experimental results. Finally, Section V concludes the paper.

II. PROBLEM DEFINITION

Let $\Re$ be the set of all real numbers, $\boldsymbol{R} \subset \Re^n$ be a finite relation. In general, the schema of $\boldsymbol{R}$ is $\boldsymbol{R}(tid, A_1, \cdots, A_n)$ with n attributes $(A_1, \cdots, A_n)$ corresponding to $\Re^n = \Re_1 \times \cdots \times \Re_n$ where the ith axis $\Re_i = \Re$ for every i, each tuple in $\boldsymbol{R}$ is associated with a *tid* (*t*uple *id*entifier), and $\boldsymbol{R}$ is stored as a base relation in a database system. We do not distinguish between $\boldsymbol{R}(tid, A_1, \cdots, A_n)$ and $\boldsymbol{R}(A_1, \cdots, A_n)$ if there is no need to refer to *tid*.

Denoting $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$ and $\boldsymbol{y} = (y_1, y_2, \cdots, y_n)$ in $\Re^n$, an aggregation function $f(\cdot)$: $\Re^n \to \Re$ is *monotone* if $f(\boldsymbol{x}) \le f(\boldsymbol{y})$ whenever $x_i \le y_i$ for every i; $f(\cdot)$ is *strictly monotone* if $f(\boldsymbol{x}) < f(\boldsymbol{y})$ whenever $x_i < y_i$ for every i; $f(\cdot)$ is *strictly monotone in each argument* if whenever one argument is strictly increased and the remaining arguments are held fixed, then the value of the aggregation function $f(\cdot)$ is strictly increased, that is, $f(\cdot)$ is

*This work is supported in part by NSFC (61170039) and the NSF of Hebei Province (F2012201006).
Corresponding author.

978-1-5090-0154-5/15 $31.00 © 2015 IEEE
DOI 10.1109/CIT/IUCC/DASC/PICOM.2015.47

IEEE computer society

strictly monotone in each argument if $x_i < x'_i$ implies that $f(x_1, \cdots, x_{i-1}, x_i, x_{i+1}, \cdots, x_n) < f(x_1, \cdots, x_{i-1}, x'_i, x_{i+1}, \cdots, x_n)$ [8]. In this paper, aggregation functions will be *monotone* or *strictly monotone in each argument*.

Let a tuple $\boldsymbol{t} = (t_1, t_2, \cdots, t_n) \in \boldsymbol{R}$. For simplicity, t_i is called the *individual grade* of $\boldsymbol{t}$ for each attribute A_i ($1 \le i \le n$), and $f(\boldsymbol{t}) = f(t_1, t_2, \cdots, t_n)$ is the (*overall*) *grade* of $\boldsymbol{t}$. A top-K query against $\boldsymbol{R}$ is to find a sorted set of K tuples $\langle \boldsymbol{t}_1, \boldsymbol{t}_2, \cdots, \boldsymbol{t}_K \rangle$ in $\boldsymbol{R}$ that have the highest grades of $f(\boldsymbol{t})$ for all $\boldsymbol{t} \in \boldsymbol{R}$. The results of a top-$K$ query are called top-K tuples. For each attribute/column A_i of $\boldsymbol{R}(tid, A_1, \cdots, A_n)$, its values are sorted in descending order from $\max(A_i)$ to $\min(A_i)$, $1 \le i \le n$, then we obtain n sorted lists $L_1, \cdots, L_n$. Each entry of L_i is of the form (tid, a), where $a = \boldsymbol{t}[A_i]$ is the value of $\boldsymbol{t}$ with tid under attribute A_i, i.e., the *individual grade* of $\boldsymbol{t}$ with respect to the ith attribute A_i.

Two modes of access to data, *sorted access* and *random access*, will be used in aggregation algorithms. Sorted (or sequential) access is to obtain the individual grade of a tuple's attribute in some sorted list by going through the list sequentially from the top. Random access is to request the individual grade of a tuple's attribute in a list and to obtain it in one step. Aggregation algorithms answer the top-K queries with the following three cases of data access methods [8, 13]. (1) Both sorted and random accesses: assume the availability of both sorted and random accesses in all the underlying data sources. The algorithms FA, TA, TA1 and CA belong to this case. (2) Restricting sorted access: assume the availability of at least one sorted access source. Random accesses are used in a controlled manner to reveal the overall grades of candidate answers. The TAz and TAz1 belong to this category. (3) No random access: assume the underlying sources provide only sorted access to tuples based on their grades. The NRA and NRA1 belong to this situation. Furthermore, aggregation algorithms are defined in the context of "*no wild guesses*" [8], that is, a tuple must be encountered under sorted access before it can be discovered by random access.

The middleware cost [8]: Let c_s be the cost of a sorted access, and c_r be the cost of a random access. If an algorithm $\mathcal{A}$ does s sorted accesses and r random accesses to find the top-K tuples, then its *middleware cost* is $cost(\mathcal{A}) = s \cdot c_s + r \cdot c_r$, for some positive constants c_s and c_r. Usually, a single sorted access is probably much more expensive than a single random access as described in [5], therefore, we assume that $c_s \ge c_r$. Additionally, Fagin defined the *database access cost* in [5] (called *unweighted middleware cost* in [6]) is $dacost(\mathcal{A}) = s + r$.

Instance optimality [8]: Let $\mathbf{D}$ be a class of relations, let $\mathbf{A}$ be a class of algorithms, and let $cost(\mathcal{A}; \boldsymbol{D})$ be the middleware cost incurred by running algorithm $\mathcal{A}$ over relation $\boldsymbol{D}$ for $\mathcal{A} \in \mathbf{A}$ and $\boldsymbol{D} \in \mathbf{D}$. An algorithm $\mathcal{B} \in \mathbf{A}$ is *instance optimal* over $\mathbf{A}$ and $\mathbf{D}$ if there are constants c and c' such that for every $\mathcal{A} \in \mathbf{A}$ and every $\boldsymbol{D} \in \mathbf{D}$ we have $cost(\mathcal{B}, \boldsymbol{D}) \le c \cdot cost(\mathcal{A}, \boldsymbol{D}) + c'$. The constant c is called the *optimality ratio*.

Fagin et al. [8] proved that TA, TAz, NRA, and CA are instance optimal, under natural assumptions. For the instance optimality of the four algorithms TA, TAz, NRA, and CA, the middleware costs (i.e., the access costs) are only considered; however, internal computation costs are ignored, which might well be expensive in practice in some cases, especially, for the algorithm NRA. The FA finds the top-K tuples with middleware cost $O(|\boldsymbol{R}|^{(n-1)/n} K^{1/n})$ over a relation $\boldsymbol{R}$ with $|\boldsymbol{R}|$ tuples if the orderings in the sorted lists are probabilistically independent [5, 6, 8], where $|\boldsymbol{R}|$ indicates the cardinality of $\boldsymbol{R}$. Moreover, TA will never access more *distinct* tuples than FA, but TA may perform more random accesses than FA [8, 9].

III. ALGORITHMS

Obviously, there is a Naïve Algorithm (NA) for obtaining the top-K tuples [8]: Under sorted access, NA probes every entry in each of the n sorted lists, computes the overall grade of every tuple by the aggregation function $f(\cdot)$, ranks all overall grades, and then returns the top-K tuples with their overall grades. In this paper, we will use the performance of NA as a baseline to compare with the other algorithms.

The algorithms TA and TAz obtain the top-K tuples with their overall grades and require only a small constant-size buffer to remember the top-K tuples and their overall grades, which needn't cache other seen tuples. However, the result set of the algorithm NRA consists of the top-K tuples without their overall grades, meanwhile it no longer suffices to have bounded buffers, because it has to use a buffer to cache all seen tuples, and the buffer size may be linear in the relation size.

In order to evaluate top-K queries over Web-accessible databases, TA and TAz need to, like NRA, cache all seen tuples [3, 16]. Moreover, NRA requires, like TA, that the top-K tuples be obtained with their overall grades in order to implement pipelined execution plans [11, 12]. Based on the main ideas of the algorithms, e.g., TA-Adapt in [3] and NRA-RJ in [11, 12], we present three algorithms TA1, TAz1, and NRA1 in this section, which are slightly modified from TA, TAz, and NRA, respectively. The algorithms TA1, TAz1, and NRA1 obtain the top-K tuples with their overall grades and cache all seen tuples in a buffer, whose size is linear in the number of seen tuples. We only present TA1, TAz1, and NRA1 and the respective differences between them and TA, TAz, and NRA in this paper, while the details of FA, TA, TAz, NRA and CA can be found in [8].

For both sorted and random accesses, we present the algorithm TA1 below in a style similar to that in [8].

Algorithm TA1

(1) For each L_i, i=1 to n
  Do sorted access in parallel to L_i. As a tuple $\boldsymbol{t}$ is seen under sorted access in list L_i,
  /* check whether $\boldsymbol{t}$ has already been in the buffer ***tSEEN*** */
  If $\boldsymbol{t}$ has been seen /* i.e., $\boldsymbol{t} \in$ ***tSEEN*** */
    Continue;
  Else /*$\boldsymbol{t}$ has not been seen, i.e., $\boldsymbol{t} \notin$ ***tSEEN***/
    Store this new seen $\boldsymbol{t}$ in the buffer ***tSEEN***. Do random access to the other lists to find the attribute value t_i of tuple $\boldsymbol{t}$ in every list. Then compute the overall grade $f(\boldsymbol{t})$ of the tuple $\boldsymbol{t}$. If this grade is one of the K highest we have seen, then remember tuple $\boldsymbol{t}$ and its grade $f(\boldsymbol{t})$ (ties are broken arbitrarily);
  End If
End For

(2) For each list L_i, let p_i be the grade of the last tuple seen under sorted access. Define the *threshold point* $\boldsymbol{p} = (p_1, p_2, \cdots, p_n)$, compute the *threshold value* $\tau = f(\boldsymbol{p})$;
If there are at least K tuples that have been seen with $f(\boldsymbol{t}) \geq \tau$, then halt;
Else goto (1).
(3) Let $\boldsymbol{Y}$ be a set containing the K tuples that have been seen with the highest grades $f(\boldsymbol{t})$'s. The output is then the sorted set $\{(\boldsymbol{t}, f(\boldsymbol{t})) \mid \boldsymbol{t} \in \boldsymbol{Y}\}$ according to the grades $f(\boldsymbol{t})$'s.

==========

The main differences between TA1 and TA are that TA1 uses the buffer ***tSEEN*** to contain all seen tuples, and employs the if-else structure "If $\boldsymbol{t}$ has been seen…Else…" in Step(1) to determine whether a tuple $\boldsymbol{t}$ has already been seen, before it is seen under the current sorted access in list L_i. In general, TA1 will never do more random accesses than TA; however, TA1 need maintain the buffer ***tSEEN***, and then may perform more maintenance than TA. Thus, TA is usually more efficient than TA1, whereas TA1 may be more efficient than TA in some special cases.

Let $Z = \{i_1, i_2, \cdots, i_m\}$, $1 \leq m = |Z| < n$, be a nonempty proper subset of $\{1, 2, \cdots, n\}$, i.e., the set of indices i of those lists L_i that can be accessed under sorted access. Without loss of generality, we assume that $Z = \{1, 2, \cdots, m\}$, $1 \leq m < n$. The algorithm TAz1 is shown as follows, which is a natural modification of TA1.

==========

Algorithm TAz1

---

(1) For each L_i, i=1 to m
Do sorted access in parallel to L_i. As a tuple $\boldsymbol{t}$ is seen under sorted access in list L_i,
/* check whether $\boldsymbol{t}$ has already been in the buffer ***tSEEN*** */
If $\boldsymbol{t}$ has been seen /* i.e., $\boldsymbol{t} \in$ ***tSEEN*** */
Continue;
Else /*$\boldsymbol{t}$ has not been seen, i.e., $\boldsymbol{t} \notin$ ***tSEEN***/
Store this new seen $\boldsymbol{t}$ in the buffer ***tSEEN***. Do random access to the other lists to find the attribute value t_i of tuple $\boldsymbol{t}$ in every list. Then compute the overall grade $f(\boldsymbol{t})$ of the tuple $\boldsymbol{t}$. If this grade is one of the K highest we have seen, then remember tuple $\boldsymbol{t}$ and its grade $f(\boldsymbol{t})$ (ties are broken arbitrarily);
End If
End For
(2) For each list L_i, let p_i be the grade of the last tuple seen under sorted access. Define the *threshold point* $\boldsymbol{p} = (p_1, \cdots, p_m, \max(A_{m+1}), \cdots, \max(A_n))$, compute the *threshold value* $\tau = f(\boldsymbol{p})$;
If there are at least K tuples that have been seen with $f(\boldsymbol{t}) \geq \tau$, then halt;
Else goto (1).
(3) Let $\boldsymbol{Y}$ be a set containing the K tuples that have been seen with the highest grades $f(\boldsymbol{t})$'s. The output is then the sorted set $\{(\boldsymbol{t}, f(\boldsymbol{t})) \mid \boldsymbol{t} \in \boldsymbol{Y}\}$ according to the grades $f(\boldsymbol{t})$'s.

==========

TAz1 need employ the buffer ***tSEEN*** to cache all seen tuples, and to check whether a tuple $\boldsymbol{t}$ has already been seen as it is seen under the current sorted access in list L_i. Thus, TAz1 may perform more maintenance than TAz, and then TAz is usually more efficient than TAz1.

The TA and TA1 do both sorted and random accesses; in this situation, obviously, TAz and TAz1 can also be utilized to handle top-K query. Intuitively, TAz and TAz1 may do less sorted accesses than TA and TA1 because $Z = \{i_1, i_2, \cdots, i_m\}$ is a nonempty proper subset of $\{1, 2, \cdots, n\}$; hence TAz and TAz1 may be more efficient than TA and TA1, respectively. In order to obtain better performance of TAz or TAz1, it is critical to choose the "optimal" subset Z_o of $\{1, 2, \cdots, n\}$, which is difficult since there are $2^n - 2$ nonempty proper subsets in terms of the set $\{1, 2, \cdots, n\}$. It is important to consider the case where $|Z| = 1$ as described in [3]; thus, we will report the best performance of TAz or TAz1 for $Z = \{i\}$, $i = 1, 2, \cdots, n$, in our experiments, which is indeed better than that of TA or TA1 for several datasets.

In order to obtain the best performances of TAz and TAz1 when $Z = \{i\}$, $i = 1, 2, \cdots, n$, we try to find a way to determine the "*optimal list*" of TAz and TAz1. Unfortunately, we have not yet come up with a general method that is suitable for every dataset and every aggregation function (at least three) in our experiments. The issue requires further investigation.

For the situations where random accesses are impossible, the algorithm NRA1 is shown below.

==========

Algorithm NRA1

---

(1) Let $p^L_1, \cdots, p^L_n$ be the smallest possible values in lists $L_1, \cdots, L_n$.
(2) Do sorted access in parallel to lists $L_1, \cdots, L_n$, and at each step do the following:
(2.1) Maintain the last seen values $p^U_1, \cdots, p^U_n$ in the n lists.
(2.2) For every tuple $\boldsymbol{t} = (t_1, \cdots, t_n)$ with some unknown attribute values, compute a lower bound for $f(\boldsymbol{t})$, denoted $f_L(\boldsymbol{t})$, by substituting each unknown attribute value t_i with p^L_i, and compute an upper bound for $f(\boldsymbol{t})$ denoted $f_U(\boldsymbol{t})$, by substituting each unknown attribute value t_i with p^U_i. For tuple $\boldsymbol{t}$ that has not been seen at all, $f_L(\boldsymbol{t}) = f(p^L_1, \cdots, p^L_n)$, and $f_U(\boldsymbol{t}) = f(p^U_1, \cdots, p^U_n)$.
(2.3) Let $\boldsymbol{Y}$ be the set of K tuples with the largest lower bound values $f_L(\boldsymbol{t})$ seen so far. If two tuples have the same lower bound, then ties are broken using their upper bounds $f_U(\boldsymbol{t})$, and arbitrarily among tuples that additionally tie in $f_U(\boldsymbol{t})$. Let M_K be the Kth largest $f_L(\boldsymbol{t})$ value in $\boldsymbol{Y}$ (i.e., $M_K \leq f_L(\boldsymbol{t})$ for every $\boldsymbol{t}$ in $\boldsymbol{Y}$).
(3) Call a tuple $\boldsymbol{t}$ viable if $f_U(\boldsymbol{t}) > M_K$. Halt when (a) at least K distinct tuples have been seen, and (b) there are no viable tuples outside $\boldsymbol{Y}$. That is, if $f_U(\boldsymbol{t}) \leq M_K$ for $\boldsymbol{t} \notin \boldsymbol{Y}$. Else goto step (2).
(4) For every tuple $\boldsymbol{t} = (t_1, \cdots, t_n)$ in $\boldsymbol{Y}$ with some unknown attribute values, do sorted access to obtain the attribute values from the corresponding lists.
(5) Return the graded set $\{(\boldsymbol{t}, f(\boldsymbol{t})) \mid \boldsymbol{t} \in \boldsymbol{Y}\}$.

==========

The NRA in [8] has only steps (1), (2), (3) and (5) in the above NRA1. In contrast, NRA1 can get the top-K tuples with

their overall grades by Step-(4), while NRA1 never does less sorted accesses than NRA.

IV. EXPERIMENTS

We report our experimental results of the nine algorithms NA, FA, TA, TA1, TAz, TAz1, NRA, NRA1 and CA with eleven datasets. The experiments are carried out using Visual C++ on a PC with Windows XP, Intel(R) Core(TM) i5-2400 CPU @ 3.10 GHz 3.09GHz, and 2.98GB memory.

A. Datasets and Preparations

The low-dimensional datasets coming from [2] involve both synthetic and real datasets. The real datasets include Census2D and Census3D (both with 210,138 tuples), and Cover4D (581,010 tuples); the synthetic datasets are Gauss3D (500,000 tuples) with Gaussian distribution, and Array3D (507,701 tuples) with Zipfian distribution. The attribute values in low-dimensional datasets are all integers. The moderate- and high-dimensional real datasets are the same as in [4]. Their attribute values are double precision floating point numbers. The moderate datasets House8D, House16D and House20D are derived from the U.S. Household Census dataset with 22,784 tuples, and their attribute values are normalized in the interval [0,1]. High-dimensional datasets Lsi25D, Lsi50D and Lsi104D are derived from Telcordia LSI Engine with 20,000 tuples, and their attribute values are in the domain $[-3.3991\times10^{38},-8.01543\times10^{-43}]\cup[1.03108\times10^{-41}, 3.40237\times10^{38}]$.

In the following discussion, the names of datasets Census2D, Census3D, Gauss3D, Array3D, Cover4D, House8D, House16D, House20D, Lsi25D, Lsi50D, and Lsi104D are abbreviated to C2D, C3D, G3D, A3D, C4D, H8D, H16D, H20D, L25D, L50D, and L104D, respectively. In the name of a dataset, suffix "*n*D" indicates that the dataset has *n* dimensions.

We use the following default settings: (1) the aggregation function is Sum-Function $f(\boldsymbol{t}) = sum(\boldsymbol{t}) = t_1 + t_2 + \cdots + t_n$; (2) the program will be executed 10 times individually for each query, and its measures is the averages of the results of the 10 executions; (3) $K = 100$ for all eleven datasets; (4) all sorted lists are in main memory. When a different setting is used, it will be explicitly specified.

The following measures are used in our experiments.

- *The elapsed time (millisecond, ms) used to obtain the top-K tuples*: The time needed to find the top-*K* tuples from the respective dataset.
- *The number of sorted accesses (denoted by #SA)*: The number of sorted accesses for a query to find the top-*K* tuples from the respective dataset.
- *The number of random accesses (#RA)*: The number of random accesses for a query to find the top-*K* tuples from the respective dataset.
- *The number of seen tuples (#SeT)*: The number of seen tuples for a query to find the top-*K* tuples from the respective dataset.

For a top-*K* query over an *n*-dimensional dataset $\boldsymbol{R}$, the TA or TAz needs only a small bounded buffer to remember the top-*K* tuples and their grades, and then the buffer size $O(nK)$ is independent of the size of the dataset. For the other algorithm $\mathcal{A}$, however, $\mathcal{A}$ may need a large buffer to store its seen tuples with their related information, and the buffer size depends on $\#SeT(\mathcal{A}) = |\boldsymbol{tSEEN}|$ the number of seen tuples by $\mathcal{A}$ over $\boldsymbol{R}$, i.e., $BufferSize(\mathcal{A}, \boldsymbol{R}) = O(\#SeT(\mathcal{A})*n) = O(n|\boldsymbol{tSEEN}|)$. Thus, we will report *#SeT* rather than the memory overhead.

B. Experiments with Default Setting

1) Elapsed Time

We use the two *API* functions *QueryPerformanceCounter()* and *QueryPerformanceFrequency()* to measure the elapsed time of each algorithm. Table I lists the elapsed times of nine algorithms for eleven datasets. If a value v is at least 1*ms* in Table I, it will be converted to an integer by ROUND(v, 0). Note that, as $Z = \{i\}$, $i = 1, 2, \cdots, n$, the elapsed times of TAz and TAz1 are the minimum values $\min_Z\{time(\text{TAz})\}$ and $\min_Z\{time(\text{TAz1})\}$ respectively in Table I for each dataset.

Firstly, from Tables I, FA, TA, TA1, TAz, TAz1, and CA outperform NA (the Naïve Algorithm) for all eleven datasets; furthermore, we can see that the performances of the TA, TA1, TAz, TAz1 and CA are stable since they are *instance optimal*, and their internal computation costs are low. NRA and NRA1 are better than NA when the dimensionality of a dataset is at most 3; however, they underperform NA if dimensionality is at least 4.

TABLE I. THE ELAPSED TIMES OF THE NINE ALGORITHMS FOR THE ELEVEN DATASETS

***time* (*ms*)**	**C2D**	**C3D**	**G3D**	**A3D**	**C4D**	**H8D**	**H16D**	**H20D**	**L25D**	**L50D**	**L104D**
NA	167	231	628	628	873	32	67	90	94	281	940
FA	30	77	206	133	389	29	45	55	59	105	223
TA	5	5	63	30	5	4	13	33	2	14	45
TA1	26	31	101	82	101	7	16	22	14	30	69
TAz	0.24	0.24	47	10	5	11	14	13	14	21	37
TAz1	25	31	62	60	102	11	17	23	24	44	71
NRA	116	201	541	469	963	73	234	312	415	1359	5218
NRA1	119	204	547	469	982	74	234	316	415	1366	5182
CA	25	30	112	84	123	8	29	60	14	34	95

Next, we compare the performances among the TA-like algorithms. Recall that CA is the Combined Algorithm by combining TA and NRA. Based on $\lfloor time(\text{NRA1})/time(\text{TA1}) \rfloor$ in Table I, we assume that $h = 4, 6, 5, 5, 10, 7, 14, 14, 29, 45$, and 75 for C2D, C3D, G3D, A3D, C4D, H8D, H16D, H20D, L25D, L50D, and L104D, respectively. Obviously, TA and TAz are better than CA from Table I since CA has to maintain all seen tuples, i.e., CA is not suitable for the case $c_s \geq c_r$.

TA, TA1, TAz, TAz1 and CA outperform FA except for CA over the dataset H20D, where $time(\text{CA, H20D}) = 60ms > time(\text{FA, H20D}) = 55ms$. There are two reasons for the exception: (1) as shown in [8, 9], TA will never access more

distinct tuples than FA, but TA may perform more random accesses than FA. From Table III, the *#RA* with TA is 1124800 for the dataset H20D, while the *#RA* is only 137660 by FA; (2) the elapsed time of NRA is much larger than that of FA (312*ms* versus 55*ms*) over the dataset H20D. Notice that the number of seen tuples in Table IV for TA includes the count of repeating seen tuples; in fact, the number of *distinct* seen tuples by TA is the same as that by TA1 in Table IV.

In Table I, the elapsed time TAz (or TAz1) is the minimum value with the *optimal list* for each dataset. Thus, TAz is highly competitive with TA, and the performance difference between TAz and TA is small for every dataset. Comparing TAz with TA, *time*(TAz) is longer than *time*(TA) for datasets H8D, H16D, L25D and L50D, while *time*(TAz) is at most *time*(TA) for other datasets. Similarly, TAz1 is highly competitive with TA1. Comparing TAz1 with TA1, *time*(TAz1) is longer than *time*(TA1) when the dimensionality of a dataset is at least 4, but *time*(TAz1) is at most *time*(TA1) if the dimensionality of a dataset is at most 3. Generally, TAz and TAz1 have the same numbers of sorted and random accesses as shown in Tables II and III respectively, while TAz1 will incur more costs than TAz to maintain the all seen tuples. Therefore, the elapsed time of TAz is at most that of TAz1 as shown in Table I.

From the definition of TA and TA1, we know that TA1 never does more random accesses than TA, while TA1 incurs additional costs to maintain all of its seen tuples. In general, TA is slightly better than TA1; however, there is one dataset H20D in which TA1 is better than TA in Table I. This is because the number of random accesses (*#RA*) for TA is much larger than TA1 (see the following Table III).

2) Numbers of Sorted and Random Accesses

In this subsection, we report the number of sorted accesses (*#SA*) in Table II and the number of random accesses (*#RA*) in Table III. Note that no random access is performed for the algorithms NA, NRA and NRA1.

TABLE II. THE NUMBER OF SORTED ACCESSES OF THE NINE ALGORITHMS FOR THE ELEVEN DATASETS

#SA	C2D	C3D	G3D	A3D	C4D	H8D	H16D	H20D	L25D	L50D	L104D
NA	420276	630414	1500000	1523103	2324040	182272	364544	455680	500000	1000000	2080000
FA	17080	75309	177699	86682	360216	90872	238000	318020	402425	893250	1975792
TA	202	303	124857	45357	10812	5440	34480	59200	4625	10350	24128
TA1	202	303	124857	45357	10812	5440	34480	59200	4625	10350	24128
TAz	101	101	116956	24284	11768	22784	22784	22784	20000	20000	20000
TAz1	101	101	147629	28943	11768	22784	22784	22784	20000	20000	20000
NRA	417958	626844	1499997	1522320	2303232	180760	364416	455520	499575	999350	2079896
NRA1	417958	626844	1499997	1522320	2303348	180760	364431	455534	499575	999350	2079896
CA	202	296	161870	40371	125644	4411	38024	79326	9975	10461	24456

TABLE III. THE NUMBER OF RANDOM ACCESSES OF THE NINE ALGORITHMS FOR THE ELEVEN DATASETS

#RA	C2D	C3D	G3D	A3D	C4D	H8D	H16D	H20D	L25D	L50D	L104D
NA	0	0	0	0	0	0	0	0	0	0	0
FA	16880	134292	305556	159480	808308	90688	126544	137660	97575	106750	104208
TA	202	606	249714	90714	32430	38080	507165	1124800	108576	499163	2454181
TA1	202	604	235348	87618	32307	23758	214515	357105	99432	396851	1449107
TAz	101	202	233912	48568	35304	159488	341760	432896	480000	980000	2060000
TAz1	101	202	295258	57886	35304	159488	341760	432896	480000	980000	2060000
NRA	0	0	0	0	0	0	0	0	0	0	0
NRA1	0	0	0	0	0	0	0	0	0	0	0
CA	150	333	203	205	507	716	1446	1826	2178	4792	10083

Generally, more middleware cost or database access cost ($dacost(\mathcal{A}) = s + r =$ *#SA* + *#RA*) will lead to more elapsed time in terms of an algorithm. From Table II, the numbers of sorted accesses (*#SA*'s) of NA, NRA and NRA1 are large; moreover, NA has to compute the overall grade of every tuple by the aggregation function and rank the overall grades of all tuples in the dataset; while NRA and NRA1 need to maintain and update the information of all their seen tuples. Therefore, NA, NRA and NRA1 have long elapsed times in Table I.

As described in Section III, NRA1 will obtain the overall grade for each top-*K* tuple, and then its number of sorted accesses is at least that of NRA. Usually, NRA and NRA1 have the same number of sorted accesses as shown in Table II. Indeed, there are datasets (e.g., C4D, H16D and H20D) with *#SA*(NRA1) > *#SA*(NRA). Therefore, NRA can obtain some top-*K* tuple(s) with unknown individual grade(s). Furthermore, the difference between *#SA*(NRA1) and *#SA*(NRA) is little. In fact, the maximum difference is 116 for C4D. Hence, NRA and NRA1 have almost the same elapsed times in Table I.

From Table II, the *#SA* of FA is much more than that of TA, TA1, TAz, TAz1, or CA for all datasets. We know that $c_s \geq c_r$ [5] (see Section II) and FA needs to maintain all of its seen tuples. Thus FA has large elapsed times in Table I and underperforms TA, TA1, TAz, TAz1 and CA (except CA over H20D), though the number of random accesses (*#RA*) of FA is smaller than that of TA, TA1, TAz, or TAz1 when the dimensionality is at least 16 (see Table III).

Table II shows that TA and TA1 do the same sorted accesses, but they often make different random accesses as illustrated in Table III since TA1 remembers all its seen tuples. Thus, the *#RA* of TA1 is at most that of TA. If the *#RA* of TA1 is much less than that of TA, the elapsed time of TA1 may be less than that of TA (e.g., for dataset H20D in Table I).

From Tables II and III, the *#SA* and *#RA* of TAz are equal to those of TAz1 respectively for each dataset except for two synthetic datasets G3D and A3D. The reason for the exception is that the optimal list of TAz differs from that of TAz1 for the dataset G3D or A3D. Each of G3D and A3D has lots of

duplicate tuples. TAz needn't insert a duplicate tuple into the buffer containing top-K tuples, while TAz1 has to remember all its seen tuples; thus, they have different optimal lists.

For the datasets C2D and C3D, the *#SA* and *#RA* of TA, TA1, TAz, TAz1 and CA are much less than those of other algorithms (Notice that no random access is performed for NA, NRA and NRA1). Since $1 \le Age \le 99$, $-25897 \le Income \le 347998$ and $0 \le WeeksWorkedPerYear \le 52$ in C2D and/or C3D, the top- K values of the attribute *Income* are much larger than those of the other attribute(s), and *Income* is the dominative factor of the function $f(\boldsymbol{t}) = t_1 + \cdots + t_n$. In fact, the tuples with top-K values of *Income* may be the top-K tuples.

TABLE IV. THE NUMBER OF SEEN TUPLES OF THE NINE ALGORITHMS FOR THE ELEVEN DATASETS

#SeT	C2D	C3D	G3D	A3D	C4D	H8D	H16D	H20D	L25D	L50D	L104D
NA	210138	210138	500000	507701	581010	22784	22784	22784	20000	20000	20000
FA	16980	69867	161085	82054	292131	22695	22784	22784	20000	20000	20000
TA	202	303	124857	45357	10810	5440	33811	59200	4524	10187	23827
TA1	202	302	117674	43809	10769	3394	14301	18795	4143	8099	14069
TAz	101	101	116956	24284	11768	22784	22784	22784	20000	20000	20000
TAz1	101	101	147629	28943	11768	22784	22784	22784	20000	20000	20000
NRA	210132	210138	500000	507700	581010	22784	22784	22784	20000	20000	20000
NRA1	210132	210138	500000	507700	581010	22784	22784	22784	20000	20000	20000
CA	202	296	147906	39136	119649	2876	15274	20954	7935	8200	14476

3) Number of Seen Tuples

NA accesses all tuples in each dataset $\boldsymbol{R}$, therefore, its *#SeT* is equal to $|\boldsymbol{R}|$ in Table IV. The *#SeT* of FA is large, and FA will see all tuples when the dimensionality is at least 16 in Table IV. Regarding the TA, we count *#SeT*(TA, $\boldsymbol{R}$) repeatedly for duplicate seen tuples; thus, there is some $\boldsymbol{R}$ in Table IV such that *#SeT*(TA, $\boldsymbol{R}$) > $|\boldsymbol{R}|$ (e.g., the dataset H20D). In fact, the number of distinct seen tuples by TA is identical with that by TA1, satisfying *#SeT*(TA1, $\boldsymbol{R}$) < $|\boldsymbol{R}|$ in Table IV.

TAz and TAz1 define the threshold point $\boldsymbol{p} = (p_1, \cdots, p_m, \max(A_{m+1}), \cdots, \max(A_n))$, and compute the threshold value $\tau = f(\boldsymbol{p})$; thus, the final τ may be too large such that $f(\boldsymbol{t}) < \tau$ for all $\boldsymbol{t} \in \boldsymbol{R}$. In this situation, TAz or TAz1 will not halt until it has seen the grade of every tuple in every list. However, this situation cannot happen with TA generally [8]. The results in Table IV confirm this. From Table IV, TAz and TAz1 will see a small number of tuples for low-dimensional datasets, but they will see all tuples for moderate- and high-dimensional datasets.

Consequently, except for TA and TAz with small bounded buffers, a larger number of seen tuples for the other algorithms means larger buffers as well as larger elapsed times generally.

C. Experiments with Different Setting

In this section, we repeat the same experiments as in Section IV.B for various settings, but we report only parts of the results of each experiment due to space limitation.

1) Effect of Different Result Size K

Let K = 5, 10, 20, 40, 100, and 200, respectively. We discuss the effect of K on the performance of the algorithms in this subsection. For $f(\boldsymbol{t}) = t_1 + \cdots + t_n$, we report the elapsed times in Figure 1 for the three original algorithms TA, TAz and NRA in [8] with three datasets C4D, H20D and L104D. In Figure 1, TA4D, TA20D and TA104D are used to indicate the values of TA for the datasets C4D, H20D and L104D, respectively. The other signs of TAz and NRA are alike. the three curves for TA grow slowly as K grows. The changes of TAz are similar to those of TA. Moreover, *time*(TAz4D, top-100) < *time*(TAz4D, top-200). The reason is that *BufferSize*(TAz4D, top-100) < *BufferSize*(TAz4D, top-200), and TAz incurs more costs to handle the *Buffer* for a top-200 query than that for a top-100 query. From Figure 1, the TA-like algorithms are stable when K changes. The reason is as follows: when TA or TAz obtains the top-K tuples, there is a high probability that the top-(K+1)th tuple has already been obtained, but it is not inserted into the buffer that stores only top-K tuples and their grades.

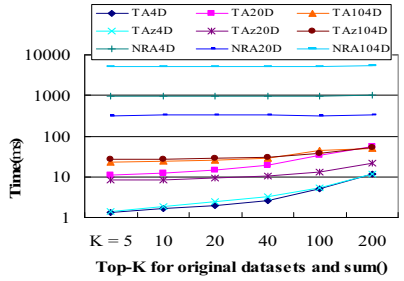


Fig. 1. Elapsed times for $5 \le K \le 200$

2) Dataset with Normalization

For each $\boldsymbol{R}$ of low- and high-dimensional datasets, we normalize it such that $\boldsymbol{R} \subset [0, 1]^n$, the Cartesian product of n closed intervals [0, 1]. For a tuple $\boldsymbol{t} = (t_1, \cdots, t_n) \in \boldsymbol{R}$, we also use the Sum-Function $f(\boldsymbol{t}) = sum(\boldsymbol{t}) = t_1 + \cdots + t_n$. Notice that we only discuss low- and high-dimensional datasets in this subsection, as the moderate-ones are already normalized [4] as described in Section IV.A. We report the elapsed time in Figure 2.

Comparing Table I in Section IV.B.1 with Figure 2, the trends of elapsed times of all algorithms over the *normalized* datasets are similar to those over the *original* datasets except for the TA and CA for the high-dimensional datasets. TA and CA are much better than NA for the high-dimensional *original* datasets; however, they underperform NA significantly for the high-dimensional *normalized* datasets. From Figure 2, we can also see that: *time*(TAz) < *time*(TAz1) < *time*(TA1) < *time*(FA)

< *time*(NA) < *time*(TA) < *time*(CA) < *time*(NRA) ≈ *time*(NRA1) for the three high-dimensional normalized ones.

To our surprise, "TA underperforms NA significantly", as we know that TA is an elegant algorithm with instance optimality [8].

TA stores only the K tuples with the highest overall grades. It may perform many sorted and random accesses to the tuples, many of which probed again and again. Therefore, the *#SeT* of TA is much more than that of NA for the three high-dimensional normalized datasets; meanwhile, TA does much more sorted and random accesses than NA, i.e., *dacost*(TA) is much more than $n|\boldsymbol{R}|$ when $\boldsymbol{R} \in$ {L25D, L50D, L104D}. These factors are the reasons behind "TA underperforms NA significantly".

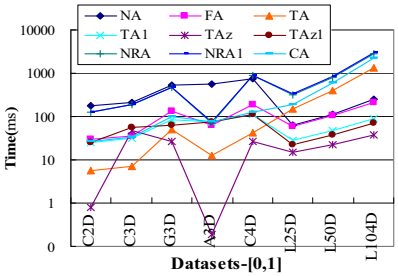


Fig. 2. Elapsed times for normalized datasets

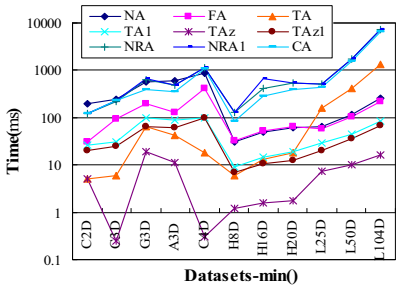


Fig. 3. Elapsed times for *min*()

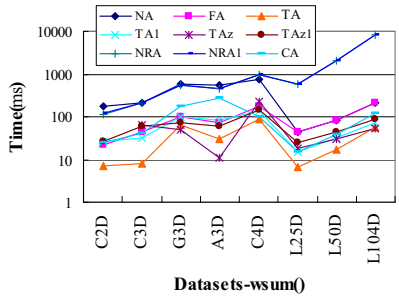


Fig. 4. Elapsed times for *wsum*()

3) *Effect of Different Aggregation Functions*

The performance of TA-like algorithms may change depending on the aggregation function used. In this subsection, we conduct the experiments for the other two widely used functions Min and Weighted-Sum: $f_1(\boldsymbol{t}) = min(\boldsymbol{t}) = min(t_1, \cdots, t_n)$ and $f_2(\boldsymbol{t}) = wsum(\boldsymbol{t}) = \omega_1 t_1 + \cdots + \omega_n t_n$, for $\boldsymbol{t} = (t_1, \cdots, t_n) \in \boldsymbol{R}(A_1, \cdots, A_n)$, where $\omega_i = 1/\max(|\max(A_i)|,|\min(A_i)|)$, $i = 1, 2, \cdots, n$.

Figure 3 shows the results of $min(\boldsymbol{t})$. Figure 4 demonstrates the results with $wsum(\boldsymbol{t})$ for low- and high-dimensional original datasets. Notice that $wsum(\boldsymbol{t})$ become $sum(\boldsymbol{t})$ since all $\omega_i = 1$ ($i = 1, \cdots, n$) for moderate-ones.

From Table I in Section IV.B.1, and Figures 3 and 4, it can be seen that the elapsed times of the nine algorithms are sensitive to the aggregation functions. Regarding high-dimensional datasets, for example, the algorithm TA underperforms NA significantly when $min(\boldsymbol{t})$ is used in Figure 3; the reasons are similar to those of the normalized datasets in Section IV.C.2. However, TA will be the best among all nine algorithms if $sum(\boldsymbol{t})$ and $wsum(\boldsymbol{t})$ are applied to the original high-dimensional datasets. Also considering high-dimensional datasets, FA is better than NA for $sum(\boldsymbol{t})$ from Table I, but NA is slightly better than FA for $wsum(\boldsymbol{t})$ from Figure 4.

4) *Min and Max Elapsed Times of TAz and TAz1*

The elapsed time of the TAz or TAz1 in the subsections above is the minimum value with the "*optimal list*" for each dataset. In this case, TAz and TAz1 are highly competitive with TA and TA1, respectively. Because we have not found a way to obtain the "*optimal list*", we use the minimum and the maximum of elapsed times to estimate TAz and TAz1.

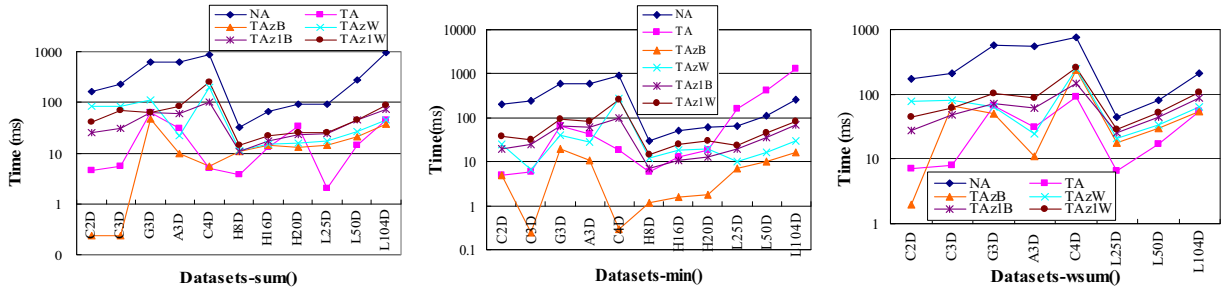
Over each original dataset with the three aggregation functions, Figure 5 depicts the elapsed times of NA, TA, TAzB, TAzW, TAz1B and TAz1W, where TAzB indicates the *B*est (or minimum) value and TAzW means the *W*orst (or maximum) value for TAz, so do TAz1B and TAz1W for TAz1.

For $sum(\boldsymbol{t})$, Figure 5(a) shows that the NA underperforms all the others. The TAz and TAz1 are stable for moderate- and high-dimensional datasets. TAz1 is also stable for low-dimensional datasets since the difference between TAz1W and TAz1B is at most 37*ms* for all low-dimensional datasets except for the dataset C4D with 148*ms*. The changes of the curves with TAz are large for low-dimensional datasets since the difference (TAzW – TAzB) is from 62*ms* to 195*ms* for all low-dimensional datasets except for the dataset A3D with 12*ms*. Comparing the worst values, TAz runs slower than TAz1 for the datasets C2D, C3D, and G3D since TAzW–TAz1W > 0. The values of TAzW, TAzB, TAz1W, and TAz1B are all smaller than that of TA for dataset H20D, while their five values are almost the same for dataset H16D.

For $min(\boldsymbol{t})$, Figure 5(b) shows that: (1) the elapsed times of TAzB are smaller than the others over the eleven original datasets, (2) NA underperforms all the others for low- and moderate-dimensional datasets, (3) TA underperforms the others significantly for high-dimensional datasets, and (4) the values of TA are between the minimum and the maximum elapsed times of TAz or TAz1 for low- and moderate-dimensional datasets. When $min(\boldsymbol{t})$ is used, TAz, TA, and TAz1 are suitable for low- and moderate-dimensional datasets, while TAz and TAz1 are good choices for high-dimensional datasets since TA halts only if it performs a lot of sorted and random accesses in order to satisfy the termination condition $f_1(\boldsymbol{t}_K) = min(\boldsymbol{t}_K) \geq \tau$.

Figure 5(c) shows the results for only low- and high-dimensional original datasets with $f_2(\boldsymbol{t}) = wsum(\boldsymbol{t})$, since $wsum(\boldsymbol{t})$ becomes $sum(\boldsymbol{t})$ for moderate-ones over which the results are

shown in Figure 5(a). From Figure 5(c), we can see that NA has significantly longer elapsed time than the others. The difference between TAz1W and TAz1B is not large, which is from 4*ms* to 30 *ms* for all datasets except for C4D with 110*ms*. The difference between TAzW and TAzB is small, which is from 3*ms* to 16*ms* for all datasets except for C2D with 76 *ms*. Furthermore, TA works well with *wsum*(***t***), since its elapsed times are smaller than the others for almost all datasets except for C2D, G3D and A3D in Figure 5(c).



(a) Aggregation function $f(t) = sum(t)$ (b) Aggregation function $f_1(t) = min(t)$ (c) Aggregation function $f_2(t) = wsum(t)$

Fig. 5. Elapsed times for NA and TA, and Min and Max elapsed times for TAz and TAz1

V. Conclusion

For processing top-*K* queries with monotone aggregation functions, this paper provides an experimental evaluation of nine algorithms NA, FA, TA, TA1, TAz, TAz1, NRA, NRA1 and CA, where NA is the Naïve Algorithm as a baseline and the others are TA-like algorithms. We conduct extensive experiments with three aggregation functions based on eleven datasets with different characteristics. From our experimental results, we summarize the following observations. (1) There is no single winner for all experiments. TAz with the "optimal list" outperforms all the other algorithms when the aggregation function is *min*(***t***) over all eleven datasets; however, TAz with the "worst list" cannot beat for instance TA for all datasets. (2) To our surprise, for the three original high-dimensional datasets, TA underperforms NA significantly when the aggregation function is *min*(***t***), in this case, TAz, TAz1 and TA1 are the most suitable algorithms, while TAz, TA, TAz1 and TA1 are preferable for low- and moderate-dimensional datasets with the function *min*(***t***). (3) The performances of TA-like algorithms are sensitive to the attribute values of tuples in a dataset. In reference to high-dimensional datasets, for example, TA is the best algorithm for original datasets, but it is not attractive for normalized datasets since TA runs considerably slower than NA in these cases. With respect to normalized datasets, TAz and TA are good choices if a dataset has at most 8 dimensions; otherwise, TAz, TAz1 and TA1 may be the suitable algorithms. (4) In general, if aggregation function is *sum*(***t***) or *wsum*(***t***) over original datasets, TAz and TA are the best algorithms. There are two exceptions: (a) *sum*(***t***) with H20D, TA1 and TAz1 are better than TA in this case, and (b) *wsum*(***t***) with C4D, all algorithms excluding NRA, NRA1 and NA are better than TAz.

In the future, we plan to evaluate the performances of existing state-of-the-art variations and improvements of the original TA-like algorithms under the same experimental framework. Another interesting issue is to devise a (heuristic) strategy to find an optimal subset $Z_o \subset \{1, 2, \ldots, n\}$, such that the elapsed time of TAz or TAz1 with Z_o is the minimum among all subsets Z's of $\{1, 2, \ldots, n\}$.

References

[1] R. Akbarinia, E. Pacitti, and P. Valduriez, Best position algorithms for top-k queries. In *VLDB*, Vienna, Austria, 2007, pp. 495-506.

[2] N. Bruno, S. Chaudhuri, and L. Gravano, Top-k selection queries over relational databases: mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27, 2 (2002), 153-187.

[3] N. Bruno, L. Gravano, and A. Marian, Evaluating top-k queries over web-accessible databases. In *ICDE*, San Jose, USA, 2002, pp. 369-380.

[4] C. Chen, and Y. Ling, A sampling-based estimator for top-k selection query. In *ICDE*, San Jose, USA, 2002, pp. 617-627.

[5] R. Fagin, Combining fuzzy information from multiple systems. In *PODS*, Montreal, 1996, pp. 216-226.

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

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

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

[9] U. Güntzer, W. Balke, and W. Kießling, Optimizing multi-feature queries for image databases. In *VLDB*, Cairo, Egypt, 2000, pp. 419-428.

[10] S.-W. Hwang, and K.C.-C. Chang, Optimizing top-k queries for middleware access: a unified cost-based approach. *ACM Trans. Database Syst.*, 32, 1 (2007), NO.5.

[11] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, Joining ranked inputs in practice. In *VLDB*, 2002, pp. 950-961.

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

[13] I. F. Ilyas, G. Beskales, and M. A. Soliman, A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40, 4 (2008), NO.11.

[14] W. Jin, and J. Patel, Efficient and generic evaluation of ranked queries. In *SIGMOD*, Athens, Greece, 2011, pp. 601-612.

[15] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung, Efficient top-k aggregation of ranked inputs. *ACM Trans. Database Syst.* 32, 3 (2007), NO.19

[16] A. Marian, L. Gravano, and N. Bruno, Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29, 2 (2004), 319-362.

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