Progressive processing of subspace dominating queries

Eleftherios Tiakas, Apostolos N. Papadopoulos & Yannis Manolopoulos

The VLDB Journal
The International Journal on Very Large Data Bases

ISSN 1066-8888
Volume 20
Number 6

The VLDB Journal (2011) 20:921-948
DOI 10.1007/s00778-011-0231-0

Springer

Your article is protected by copyright and all rights are held exclusively by Springer-Verlag. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your work, please use the accepted author's version for posting to your own website or your institution's repository. You may further deposit the accepted author's version on a funder's repository at a funder's request, provided it is not made publicly available until 12 months after publication.

Springer

Author's personal copy

The VLDB Journal (2011) 20:921–948
DOI 10.1007/s00778-011-0231-0

REGULAR PAPER

Progressive processing of subspace dominating queries

Eleftherios Tiakas · Apostolos N. Papadopoulos · Yannis Manolopoulos

Received: 27 June 2010 / Revised: 28 February 2011 / Accepted: 30 March 2011 / Published online: 24 April 2011
© Springer-Verlag 2011

Abstract A top-k dominating query reports the k items with the highest domination score. Algorithms for efficient processing of this query have been recently proposed in the literature. Those methods, either index based or index free, apply a series of pruning criteria toward efficient processing. However, they are characterized by several limitations, such as (1) they lack progressiveness (they report the k best items at the end of the processing), (2) they require a multi-dimensional index or they build a grid-based index on-the-fly, which suffers from performance degradation, especially in high dimensionalities, and (3) they do not support vertically decomposed data. In this paper, we design efficient algorithms that can handle any subset of the dimensions in a progressive manner. Among the studied algorithms, the Differential Algorithm shows the best overall performance.

Keywords Dominating queries · Progressive algorithms · Subspaces

1 Introduction

Preference-based query processing has been an active area of research during the last years. Two basic types of preference-based queries have been studied in the literature as follows: (1) the top-k query and (2) the skyline query. In a top-k query, users should provide a ranking function $f(x)$ that determines the score of each item x. This ranking function is used to provide an ordering of the database items in such a way that items which are more preferable to a user should have higher score values. Usually, the ranking function $f(x)$ is monotonically increasing (or decreasing). An important advantage of this query is that the cardinality of the result set is bounded by the number k.

On the other hand, in a skyline query, the system determines the best (maximal or minimal) vectors (items) such that these items are not dominated by any other item in the database. Here, domination is determined by means of the attribute values of each item. More specifically, assuming a d-dimensional data set S, an item $p = (p.x_1, p.x_2, \dots, p.x_d) \in S$ *dominates* another item $q = (q.x_1, q.x_2, \dots, q.x_d) \in S$ (and we write $p \prec q$), when: $\forall i \in \{1, \dots, d\} : p.x_i \le q.x_i \wedge \exists i \in \{1, \dots, d\} : p.x_i < q.x_i$. In other words, p dominates q, if p is at least as good as q in every dimension, and it is strictly better than q in at least one of them. In such a query, no ranking function is required, and the result is not affected by scaling operations applied to the dimensions. The most important advantage of skyline queries is that no parameters are required. However, the cardinality of the result set is arbitrary and depends on the data distribution, cardinality, and dimensionality.

In an effort to combine the advantages of the aforementioned query types (and avoid their limitations), some research works have appeared studying the problem of selecting the best items that express the highest domination power. Each item p is assigned a score, $dom(p)$, which equals the number of items that p dominates. More formally: $dom(p) = |\{q \in S : p \prec q\}|$. The intuition behind this score is that the number of items dominated by p is an indication of how well p is located in the data space. In other words, the more items p dominates, the more powerful it is. A top-k dominating

E. Tiakas (✉) · A. N. Papadopoulos · Y. Manolopoulos
Department of Informatics, Aristotle University,
54124 Thessaloniki, Greece
e-mail: tiakas@csd.auth.gr

A. N. Papadopoulos
e-mail: papadopo@csd.auth.gr

Y. Manolopoulos
e-mail: manolopo@csd.auth.gr

Author's personal copy

	A	B	C	D	E	F	G	H	I	J
Distance to beach (km)	0.8	0.5	0.1	0.9	0.2	2	0.4	1	0.3	1.3
Price (euros)	50	100	35	75	65	20	80	45	40	30
Quality (1 is best)	3	1	4	2	3	5	2	3	4	4
Age (years)	8	4	17	7	11	25	9	12	15	21

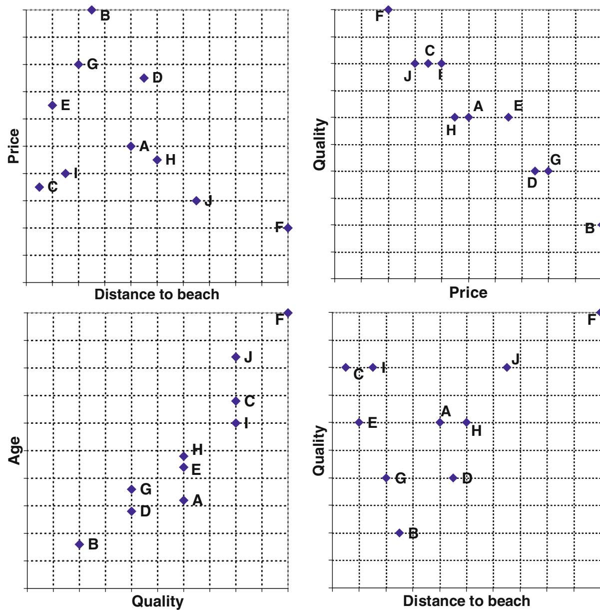

Fig. 1 The hotel example

query returns the k items with the highest scores according to the aforementioned scoring function. This query type has been proposed in [29], and it has three significant advantages in comparison with regular top-k and skyline queries: (1) no ranking function is needed, (2) the number of answers is controlled by the parameter k, and (3) the answer is not affected by possible scaling applied to dimensions.

We clarify these concepts by a simple example. Assume that there are ten hotels as depicted in Fig. 1, with attribute values scaled in [0, 1]. Consider a user who prefers to issue a top-2 dominating query based on the attributes *distance* and *price*. In this case, the domination values are 1, 0, 7, 0, 3, 0, 1, 0, 5, 0. The answer to the query consists of C (top-1) and I (top-2). Now, consider another user who prefers a top-2 dominating query based on the attributes *price* and *quality*. Then, the corresponding items follow an anti-correlated distribution and their domination values are 1, 0, 1, 1, 0, 0, 0, 2, 0, 2, respectively, and the answer to the query consists of hotels H, J (ties are broken arbitrarily). Another user may focus on *quality* and *age* attributes. Then, the corresponding items follow a correlated distribution, and their domination values are 6, 9, 2, 8, 5, 0, 6, 4, 3, 1, respectively, thus the answer to the query consists of hotel B and D. Finally, another user may prefer *distance* and *quality*. In this case, the domination values of all hotels are 3, 5, 3, 3, 5, 0, 5, 2, 2, 1. Therefore, the best hotels are B, E, and G. It is evident that the answer to a top-k dominating query does not necessarily consist of skyline items. The only exception is the top-1 item, which is guaranteed to be part of the skyline as has been shown in [29].

Although the initial number of dimensions may be large, users are not interested in all of them. In most cases, only a small subset of the available dimensions is used to express the query. Therefore, the first desirable property of a top-k dominating query processing algorithm is *to be able to operate efficiently on different subsets of dimensions*, letting users select the preferable ones. We denote by $I = \{i_1, i_2, \ldots, i_m\}$, $2 \leq m \leq d$ the subset of the user-selected dimensions. The second desirable property in a top-k dominating processing algorithm is *to provide the results in a progressive manner*. This means that as soon as the first results are available they can be returned to the user, while more answers are being prepared. A progressive algorithm determines the best item first, then the second best and so on. In contrast, a non-progressive algorithm provides the results at the end of the processing, when all relevant items have been determined. In summary, the contributions of our work are as follows:

- Novel algorithms for top-k dominating queries on vertically decomposed data are proposed, which support query processing in subsets of the available dimensions, allowing significant flexibility.
- The proposed schemes return the results in a progressive manner, enabling early termination if adequate results have been returned to the user. This way, non-blocking query plans may be produced.
- An analytical study is provided, which offers estimations for several characteristics of our methodology as well as a worst-case complexity analysis.
- A set of pruning rules is offered, being able to eliminate items that will not contribute to the final result, thus improving query performance.
- A detailed performance evaluation is provided based on real-life and synthetic data sets of various cardinalities and distributions. Moreover, our techniques are compared to state-of-the-art algorithms for top-k dominating queries, showing significant performance improvement.

The rest of the article is organized as follows. Section 2 summarizes related work in the area, whereas Sect. 3 discusses fundamental concepts required for the subsequent algorithms. Section 4 studies the proposed algorithms in detail, and Sect. 5 contains an analytical study focusing on some important aspects of the algorithms. In addition, the worst-case complexity of the algorithms is given. In Sect. 6, we suggest several pruning rules to speed-up processing. Section 7 offers performance evaluation results. Finally, Sect. 8 concludes our work.

Author's personal copy

2 Related work

2.1 Skyline queries

Since the introduction of the skyline query in database systems in [4], there was a significant effort to improve query processing efficiency as well as to provide alternatives toward eliminating the major problems inherent in skylines, which are as follows: (1) the size of the output cannot be controlled and (2) there is no ranking among skyline objects. To tackle the aforementioned limitations [5,32], ranks skyline points according to their presence in skylines of subspaces, whereas [14] offers a way to select the k most promising skyline members, in order for these k representatives to dominate the maximum number of objects. In the same lines, [23] studies techniques to select k skyline representatives in a different way, which better describe the skyline set. Another set of research efforts relaxes (or changes) the definition of dominance toward providing more flexible criteria to select skyline objects. More specifically, the works in [6,28] relax the concept of dominance, whereas a more general framework is provided in [34].

2.2 Top-k dominating queries

Top-k dominating queries have been introduced in [17] and studied further in [29,30], in order to select the most powerful objects (not necessarily skyline objects) by using the classical concept of dominance (without relaxing or changing the dominance definition). The ranking provided by this query is quite intuitive, it does not require specialized scoring functions, and the size of the output is controlled by the parameter k. Among the algorithms studied previously, CBT (cost-based traversal) [29] shows the best overall performance over aggregate R-trees, whereas FNP (fine grained with partial dominance) [30] is an interesting index-free alternative. We note that:

(i) In our case, the data is vertically decomposed and each dimension is treated separately. In order to apply CBT, an aggregate R-tree index must be constructed on-the-fly for the selected subset of dimensions using bulk loading. Then, CBT can run on top of this R-tree. Evidently, CBT requires at least one pass to the data which is not acceptable. In addition, as it is demonstrated in the experimental results, even without counting the bulk loading cost, our algorithms are superior to CBT.

(ii) Regarding FNP, the algorithm builds a grid index for the selected dimensions and therefore, it supports subspace query processing. However, for large dimensionalities, the number of cells is huge, leading to performance degradation, whereas it requires three passes over the data, which is not acceptable.

(iii) Both methods must first synthesize the data set by concatenating dimension values, which requires a pass through all data in the selected dimensions. In addition, none of these algorithms is progressive, which is a desirable property to enable incremental computation of results.

In [13], an algorithm has been proposed supporting top-k dominating queries in uncertain databases. The proposed approach has the same limitations with the previous one, since it is again based on aggregate R-trees. The novel feature of this method is that it handles uncertainty in a clear and meaningful way in applications where uncertainty cannot be avoided.

In [33], a randomized algorithm is proposed supporting probabilistic top-k dominating queries in uncertain data. Again, the proposed approach is based on aggregate R-trees. The proposed algorithm is highly accurate when the data cardinality and the dimensionality are low.

The concept of top-k dominating queries has been also used in [18]. That work studies web services discovery issues using dominance relationships through multi-criteria matching. However, the proposed techniques use ranking functions and similarity scores in order to extract the results, and these calculations differentiate from top-k dominating queries.

A problem that is concept-wise related to top-k dominating queries is to determine the k vertices in a bipartite graph with the highest degree. If we think of our data set as a bipartite graph, then a directed edge from vertex v to vertex u means that v dominates u. In [24], efficient algorithms to solve the kMCV (k most connected vertex) problem have been proposed. These algorithms can be used to solve top-k dominating queries in subspaces, and for this reason, we include them in the performance evaluation study. As it is shown in the performance evaluation in Sect. 7, each *edge probing* of kMCV corresponds to a domination check between two data items. Since these probes are performed in a random order, locality of references is not preserved, leading to a significant increase in the I/O cost.

2.3 Queries in subspaces

All previous query processing techniques for top-k dominating queries are inspired by related algorithms for skyline query processing. The BBS (branch-and-bound skyline) algorithm [17] operates on an R-tree and discovers the skyline objects in a progressive manner by a careful inspection of tree nodes. The fundamental limitation of index-based algorithms is that data organization involves the whole set of dimensions and therefore queries that focuses on specific dimensions cannot be processed efficiently. For example, building an R-tree (or any other multi-dimensional index) for 50 dimensions and letting users select any subset of these dimensions to perform

Author's personal copy

queries, it is expected to result in performance degradation, first because of the index inefficiency to handle high dimensionalities and second, due to projection operations required which destroy the spatial locality of index nodes.

A different direction has been followed in other research efforts. More specifically, subspace skyline queries have been studied in [20,27,31] and [21]. These techniques although relevant to our work cannot be used for top-k dominating queries since their construction is based on the concept of skylines, and they lack mechanisms to count the domination score of an item. In addition, these algorithms are not designed to continuously update the skyline set, and thus, they are not applicable in our setting. Finally, as shown in [29], answering dominating queries by using skyline queries is less efficient than specialized algorithms like CBT.

2.4 Threshold-based algorithms and vertically decomposed data

Our work is inspired by Fagin's pioneering work on threshold-based top-k algorithms [10]. The basic property of these algorithms is that there is a terminating condition that if met, we are sure that all possible answers have been already seen. The application of such an approach comes naturally when: (1) data are vertically decomposed [7] and therefore, each dimension is managed separately and (2) different subsets of attributes are managed by different subsystems, meaning that query processing should be performed in a middleware. Regarding the first case, systems like C-Store [1,19] and MonetDB [3] are used in place of traditional row-oriented systems toward enhanced query processing efficiency. Regarding the second case, several top-k [16] and skyline [2,15] processing techniques have been proposed to operate in a distributed environment.

In the sequel, we study algorithms for top-k dominating queries that handle each dimension separately, provide the results in a progressive manner and they are not affected by the limitations that characterize multi-dimensional indexing schemes used for data organization. Moreover, our algorithms can be used in middleware, and therefore, they enable distributed query processing when dimensions are handled by different subsystems. In addition, the progressive nature of our techniques enables the support of non-blocking query execution plans, which is important for query processing efficiency and eliminates the requirement to store intermediate results during query execution.

3 Fundamental concepts

R-trees and variants have been extensively used in the literature to support a broad range of queries over multi-dimensional data sets [4,15,17,29]. The major limitation of such an index is that since all dimensions are used to organize the data set, subspace search requires a series of projection operations which impacts efficiency due to increased I/O activity. Moreover, the resulting indexing scheme becomes inefficient even for processing queries involving the whole set of dimensions, due to the dimensionality curse.

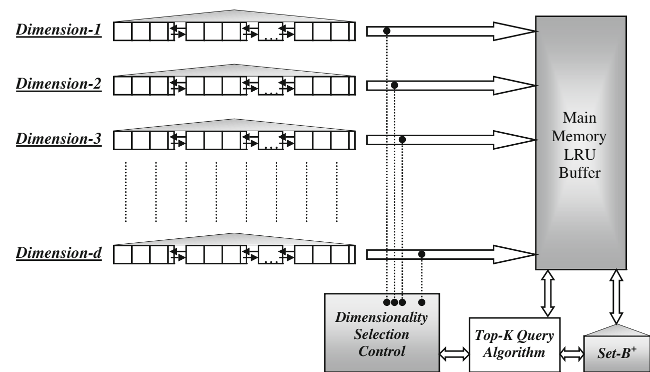

Fig. 2 Query processing components

During the last years, alternative approaches have been proposed organizing each dimension separately, resulting in a *column-oriented* (i.e., vertically decomposed) physical data organization. Two of the most widely known research efforts that have adopted the column-oriented approach is C-Store [1,19] and MonetDB [3]. Column-oriented storage shows significant performance improvements in specific types of operations and moreover, offers a completely independent treatment of dimensions, thus supporting flexible query processing involving a subset of dimensions. This way, the rest of the dimensions does not participate in query processing, and thus, query evaluation becomes more efficient. In our setting, a column-oriented organization is a natural and intuitive choice taking into account that it is impossible to provide a full-dimensional indexing scheme for every $2^d - 1$ possible subsets of dimensions. Moreover, in applications where subsets of attributes are managed by different subsystems, processing must be performed in a middleware [10,16]. In this case, our algorithms are directly applicable, provided that the different subsystems facilitate sorted access.

The architecture of the physical organization is depicted in Fig. 2. Each dimension $1 \leq i \leq d$ is organized separately by a B^+-tree, which facilitates random as well as sorted access.[1] Moreover, at any given time instance, insertions and deletions of items may take place, which are nicely handled by the trees in logarithmic time. Each B^+-tree indexes the attribute values of the specific dimension. A user may select any subset of dimensions, whereas query execution is supported by the use of an LRU buffer.

[1] It is noted that any one-dimensional access method with random and sorted access support may be used.

Author's personal copy

The B$^+$-trees are organized as follows: (1) the sorting keys are the attribute values of the items, (2) the last level record of an item contains the item attribute value, the corresponding item ID, and an additional *equality bit*, which is set to 0 if the item attribute value differs from the attribute value of its previous item in the sorting order, and it is set to 1 otherwise. Due to the sorted order of values, all items with the same value for a specific attribute as p appear sequentially in the leaf level of the B$^+$-tree. If an item p has an equality group E_p (in a specific dimension) with e equal items in the following order: $p_1, p_2, \ldots, p_i, p, p_{i+1}, \ldots, p_e$, then the corresponding records of all these equal items will have their equality bits set to 1 except from the first one (p_1) which will have an equality bit of 0. We denote by $Eb_p = \{p_1, p_2, \ldots, p_i\}$ the set of items having the same value as p in a specific dimension and stored before p, whereas $Ea_p = \{p_{i+1}, \ldots, p_e\}$ is the set of items having the same value as p in this dimension and are stored after p. Using the above, we get $E_p = Eb_p \cup \{p\} \cup Ea_p$ and $|E_p| = |Eb_p| + |Ea_p| + 1$, which always hold. In case where there are no other equal values as p, we have: $E_p = \{p\}$, $|E_p| = 1$, $Eb_p = Ea_p = \emptyset$.

An additional B$^+$-tree is used in our framework, called the *Set-B$^+$*, which is used by the top-k query algorithm for intermediate computations.[2] At the beginning of any top-k query, the Set-B$^+$ is empty. During execution, the IDs of scanned items are inserted into the Set-B$^+$ and several counters are updated. By using this approach, all algorithms' intermediate results are kept on disk (if needed) and there is no need of additional main memory storage. The Set-B$^+$ shares the same LRU buffer with all other data B$^+$-trees.

Let S be a multi-dimensional data set and I a set of selected dimensions. For the rest of the paper, we assume that small attribute values are preferable. The score of an item p is denoted as $dom(p)$ and equals the number of items dominated by p. The algorithms studied in the sequel are using the concept of *terminating items*, which has been introduced in [2,15] but studied only for skyline queries.

Definition 1 A *terminating item* is an item whose attribute values on all dimensions of interest have been retrieved (scanned) via sorted access.

Proposition 1 *If in dimension i_s items are sorted in ascending order and duplicate values are allowed, then any item $p \in S_{\{i_s\}}$ which has the same attribute value in dimension i_s with the set of items $E_p = \{p_1, p_2, \ldots, p_e\} \subset S_{\{i_s\}}$ (where $p \in E_p$) may be dominated by items that have been reached via sorted access before p and it may dominate items that will be retrieved via sorted access after p. For an item $x \in E_p$, x may dominate p, x may be dominated by p or x and p may be equivalent (i.e., they do not dominate each other).*

Proposition 2 *If in a dimension i_s the data items are sorted in ascending order and a data item p has been retrieved after $pos(p)$ sorted accesses, then $dom(p) \leq N - pos(p) + |Eb_p|$.*

During sorted accesses, we keep track of the last retrieved equality group of items located before p and compute $|Eb_p|$. Using Proposition 2, we have an upper bound for the score of any newly discovered terminating item. All discovered terminating items are maintained in a maxheap data structure prioritized on their estimated domination scores. During processing, we compute the exact domination values of the top estimated items only, updating the heap as necessary. The following proposition provides the condition for a terminating item to be reported as the next top result, enabling the progressive behavior of our algorithms.

Proposition 3 *Let t_1 and t_2 be the first two terminating items in the heap. If $dom(t_1) \geq estdom(t_2)$, then t_1 can be immediately reported as the next top result. Moreover, even if the exact domination value $dom(t_2)$ of t_2 has already been calculated and $dom(t_1) \geq dom(t_2)$, then again t_1 is the next best result.*

Proof By using Proposition 2, we have that $dom(t_2) \leq N - pos(t_2) + |Eb_{t_2}|$, and combining with the assumption $N - pos(t_2) + |Eb_{t_2}| \leq dom(t_1)$, we have: $dom(t_2) \leq dom(t_1)$. Moreover, any subsequent terminating item t satisfies one of the following:

Case 1: t is located after t_2 in the sorting order but lies into the same equality group with t_2. Then, if p is the first item exactly after the common equality group of t, t_2, we have:

$$\begin{aligned}
estdom(t) &= N - pos(t) + |Eb_t| \\
&= N - (pos(p) - |Ea_t| - 1) + |Eb_t| \\
&= N - pos(p) + |Ea_t| + |Eb_t| + 1 \\
&= N - pos(p) + |E_t| \\
&= N - pos(p) + |Ea_{t_2}| + |Eb_{t_2}| + 1 \\
&= N - (pos(p) - |Ea_{t_2}| - 1) + |Eb_{t_2}| \\
&= N - pos(t_2) + |Eb_{t_2}| = estdom(t_2)
\end{aligned}$$

Thus, using Proposition 2, we have:

$$dom(t) \leq estdom(t) = estdom(t_2) \leq dom(t_1)$$

Case 2: t lies in a next equality group after the equality group of t_2. Then, if p is the first item exactly after the equality group of t_2, we have:

$$estdom(t) = N - pos(t) + |Eb_t| \leq N - pos(p)$$

[2] We note that instead of the Set-B$^+$-tree a hashing scheme could also be applied, which is expected to show better performance. However, we selected the B$^+$-tree mainly for uniformity purposes.

Author's personal copy

The equality holds in case where p and t lie in the same equality group. Using Proposition 2, we get that:

$$\begin{aligned} dom(t) &\le estdom(t) \le N - pos(p) \\ &= N - (pos(t_2) + |Ea_{t_2}| + 1) \\ &= N - pos(t_2) - |Ea_{t_2}| - 1 \\ &= N - pos(t_2) - (|E_{t_2}| - |Eb_{t_2}| - 1) - 1 \\ &= N - pos(t_2) - |E_{t_2}| + |Eb_{t_2}| \\ &\le N - pos(t_2) + |Eb_{t_2}| \\ &= estdom(t_2) \le dom(t_1) \end{aligned}$$

In both cases, the domination value of any subsequent terminating item is less than or equal to that of t_1, thus t_1 can be reported as the next top result.

If the exact domination value $dom(t_2)$ of t_2 is available and $dom(t_1) \ge dom(t_2)$, the proposition also holds. This is because t_1 and t_2 are the two top heap items, thus any subsequent terminating item t will have an exact domination value $dom(t)$ that satisfy the property $dom(t_1) \ge dom(t_2) \ge dom(t)$, or t will have an estimated domination value $estdom(t)$ that satisfy the property $dom(t_1) \ge dom(t_2) \ge estdom(t)$. Therefore, t_1 can again be reported as the next top result. □

4 Query processing algorithms

We begin with the description of the Basic-Scan Algorithm (BSA) which demonstrates the fundamental features of our proposal, i.e., progressiveness and subspace support. Afterward, we study the Union Algorithm (UA), the Reverse Algorithm (RA), and the Differential Algorithm (DA) which apply several techniques to speed-up query processing.

4.1 The Basic-Scan Algorithm (BSA)

Figure 3 depicts the outline of BSA. The algorithm requires a maxheap HP which stores the discovered terminating items. For each terminating item t_i, the following attributes are stored: the item id, the domination value $dom(t_i)$ (estimated or exact) which is the prioritization attribute, and a flag that is true when the domination value is exact (false otherwise). Additionally, the algorithm uses the following: (1) an integer counter array $eqCnt[m]$, which stores the size of the last reported equality group (i.e., $|Eb_t|$) in each selected dimension, (2) a B$^+$-tree data node array $Bnode[m]$, which keeps the data of the current examined B$^+$-tree node in each selected dimension, (3) two variables: pos which keeps the current scan position in B$^+$-trees and a which keeps the current examined dimension.

The inner loop (lines 7–14) inspects and extracts the next best item, whereas the outer loop (lines 6,15–19) repeats the

```
BasicScanTopKDominatingQuery(I,k)
 1. m=|I|
 2. initialize LRU-Buffer, Heap HP, Set-B+, eqCnt[]
 3. for all selected dimensions do
 4.    goto first record in the corresponding B+-Tree,
       read item id, insert item in Set-B+ and update counters
 5. pos=a=j=1 ; kth=-1 ; f=true
 6. do
 7.   do
 8.     if HP.size=0 then ScanNextTerminatingItem(I)
 9.     ScanNextTerminatingItem(I)
10.     get item t1 from HP.top and HP.deheap(t1)
11.     get item t2 from HP.top
12.     if dom(t1) is not exact then CalculateDomValue(I,t1)
13.     if dom(t1)<dom(t2) then HP.enheap(t1)
14.   while dom(t1)<dom(t2)
15.   if j=k then kth=dom(t1)
16.   if j>k and kth≠dom(t1) then f=false
17.   if f=true then report id of t1 and dom(t1)
18.   j++
19. while f=true
```

Fig. 3 Outline of BSA

```
ScanNextTerminatingItem(I)
 1. h=ia ; m=|I|
 2. do
 3.    goto next record in dim h, read item p, id and eqb (eq-bit)
 4.    if eqb=0 then eqCnt[h]=0 else eqCnt[h]++
 5.    update counters of item id in Set-B+
 6.    if visit counter of id in Set-B+ is less than m then
 7.       if h=im then {a=1 ; h=i1 ; pos++} else {a++ ; h=ia}
 8.    end-if
 9. while visit counter of id in Set-B+ is less than m
10. dom(p)=N-pos+eqCnt[h]
11. HP.enheap(p) with dom(p) estimated
12. repeat line 7
13. return
```

Fig. 4 Scan next terminating item

top item extraction for at least k times. Lines 15,16 detect the k-th exact domination value and check whether the next top items have the same domination values, in order to continue reporting them (covering cases similar to that in the last example of Fig. 1). Line 17 reports the top item (in any case). Lines 8,9 scan the next terminating item using the algorithm presented in Fig. 4. Line 12 checks whether t_1 has an exact or an estimated domination value and in the second case invokes the procedure of Fig. 5. Lines 13,14 check the condition of Proposition 3 for t_1 and t_2.

Figure 4 shows the procedure for obtaining the next terminating item. Lines 2–9 implement the round-robin scan of items in the B$^+$-trees and detect the next terminating item.

When an item is found in all m dimensions, its counting closes (becomes a terminating item) and the loop exits (line 9). More specifically, line 3 moves to the next record in the B$^+$-tree of the current scanned attribute and gets the existing item p with its corresponding id and equality bit eqb directly. In case that the next record lies in a next B$^+$-

Author's personal copy


```
int CalculateDomValueBSA(I,t)
 1. backup current record position in dimension h that t closed
 2. dom(t)=0
 3. set the left-most record of the eq-group of t in h as current
 4. do
 5.     retrieve item p from the current record
 6.     if p≠t and p≺t then dom(t)++
 7.     goto next record in dimension h
 8. while not EOF in dimension h
 9. restore the record position in dimension h
10. return dom(t)
```

Fig. 5 Exact domination value calculation for BSA

Selected dimensions(I)		Sorting order in the leaves of the B+Trees									
i_1 dimension (*distance*)	id	C	E	I	G	B	A	D	H	J	F
	val	0.1	0.2	0.3	0.4	0.5	0.8	0.9	1	1.3	2
	eqbit	0	0	0	0	0	0	0	0	0	0
i_2 dimension (*price*)	id	F	J	C	I	H	A	E	D	G	B
	val	20	30	35	40	45	50	65	75	80	100
	eqbit	0	0	0	0	0	0	0	0	0	0
				t_1	t_2		t_3				

Fig. 6 BSA execution example

tree node, we first load that node in $Bnode[h]$ by replacing the previous node in memory. In the next step, the equality bit is checked, and if it is equal to 0 (line 4), then we are in a new equality group and the corresponding counter is set to 0. On the other hand, if the equality bit is 1, the counter increases by 1 in the selected dimension. Line 5 updates the number of visits of the found item and other useful counters in Set-B$^+$. If this number of visits is less than m (line 6), then the counting cannot close and before the next loop the scanning variables a, h, pos are updated in a round-robin fashion (line 7). In lines 10–11, the terminating item p is enheaped into HP together with its estimated domination value, which is determined by using the formula of Proposition 2.

Figure 5 shows the outline of the procedure for calculating the exact domination value. Let h denotes the last dimension where the terminating item has been detected. We scan sequentially the remaining values in h, we access each item and check whether it is dominated by the terminating item. The domination score is computed based on the domination check between t and all items that are in the same equality group with t and after t in dimension h (Proposition 1). The domination check in line 6 is performed by retrieving the attribute values of the scanned items using random accesses.

Subsequent algorithms change the way that scores are computed toward performance improvement.

Example 1 (BSA execution) We apply a top-2 dominating query to the data depicted in Fig. 1, on the attributes *distance* and *price*. The data set layout is given in Fig. 6.

Initially, the first two terminating items t_1, t_2 are scanned and enheaped. Six value accesses are required to retrieve t_1 by visiting the items in the B$^+$-trees for dimensions i_1, i_2 with the following order: C, F, E, J, I, C. Then, t_1 (which is item C), is enheaped with an estimated domination value of 7 (using the formula of Proposition 2). Two more value accesses are required to retrieve t_2 by visiting items G, I. Then, t_2 (which is item I), is also enheaped with an estimated domination value of 6. In the next step, t_1 is extracted from the top of the heap (because it has the maximum value), and its exact domination value is computed by checking whether it dominates items I, H, A, E, D, G, B in the B$^+$-tree of dimension i_2. As all these items are dominated by t_1, its exact domination value remains 7. The condition $dom(t_1) = 7 > 6 = dom(t_2)$ suggests that Proposition 3 is satisfied, thus C can be immediately reported as the top-1 item with domination value 7. The heap now contains only t_2 with an estimated domination value. The scanning continues, and four additional value accesses (B, H, A, A) are required to retrieve the third terminating item t_3 (which is item A). Then, t_3 is enheaped with its estimated domination value (4). In the sequel, t_2 is deheaped because it has the maximum value (6), and its exact domination value is calculated by checking whether it dominates items H, A, E, D, G, B in the B$^+$-tree of dimension i_2. Item E is not dominated by t_2, thus its exact domination value is set to 5. Then, condition $dom(t_2) = 5 > 4 = dom(t_3)$ suggests that Proposition 3 is satisfied again, and item I can be reported as the second best item with domination value 5, and the algorithm stops. The algorithm has performed in total 12 value accesses and 13 comparisons.

4.2 Union Algorithm (UA)

Although BSA is progressive, it performs a significant number of random accesses for domination checking. The next algorithm, *Union Algorithm* (UA), alleviates this problem as it does not require any explicit domination check among data items.

Proposition 4 *Let p an item and Ui_s, UE, U item sets defined as follows:*

$$Ui_s = \{p' \in S : p'.x_{i_s} < p.x_{i_s}\},\ s \in \{1, \ldots, m\}$$

$$UE = \{p' \in S - \{p\} : \forall i_s \in I,\ p'.x_{i_s} = p.x_{i_s}\}$$

$$U = \bigcup_{1 \le s \le m} Ui_s$$

Then, $dom(p)$ is given by the formula:

$$dom(p) = N - 1 - |U| - |UE|$$

Proposition 4 uses the union of Ui_1, Ui_2 which are the sets of items that lie inside the rectangular areas $yy'Oi_2$ and $xx'Oi_1$ of Fig. 7, respectively (excluding their perimeters), and the set of items (UE) in which all their attribute values are identical to p. These sets are easily computed due to the following:

Author's personal copy

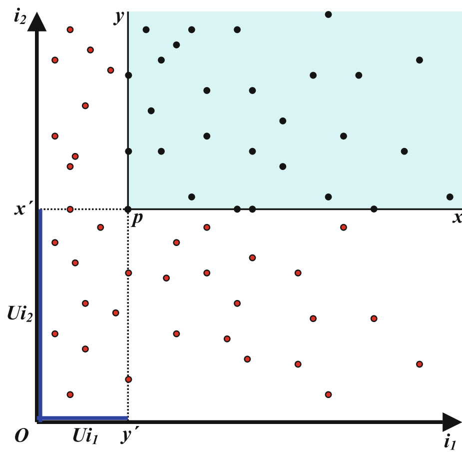

Fig. 7 UA domination rule

Proposition 5 *The union set U contains all items in the sorting order which are before the equality groups that p belongs to in each selected dimension ($E_p(i_s)$, $s = 1, \ldots, m$). Moreover, the set UE of identical items is equal to the intersection of all these equality groups, excluding p:*

$$UE = \bigcap_{1 \leq s \leq m} E_p(i_s) - \{p\}$$

Proof By the definition of Ui_s, any item $p' \in Ui_s$ has an attribute value strictly smaller than that of p, i.e. $p'.x_{i_s} < p.x_{i_s}$. Therefore, the item p' must be before the equality group $E_p(i_s)$ of p in the sorting order of dimension i_s in the corresponding B$^+$-tree.

Moreover, any item $p' \in UE$ must have all its attribute values identical to p, i.e. $p'.x_{i_s} = p.x_{i_s}$, $\forall i_s \in I$. Therefore, p' must lie in each equality group $E_p(i_s)$, $\forall i_s \in I$ and $p' \neq p$. Thus, UE is the intersection of all these equality groups without the p item, i.e. $UE = \bigcap_{1 \leq s \leq m} E_p(i_s) - \{p\}$. □

The importance of Propositions 4 and 5 is that the exact domination value of an item p can be computed by counting the items located before p in the sorted order of B$^+$-trees in each selected dimension (which we already have visited during the scan of terminating items), and the items of the equality groups that p lies. By inspecting Fig. 7, it is evident that the sorted order can be formulated by using projections to each axis. Thus, Ui_1 contains the projected items to the interval Oy' of axis i_1, without considering the items of line $y'y$ which belong to equality group $E_p(i_1)$. Similarly, Ui_2 contains the projected items to the interval Ox' of axis i_2, without taking the items of line $x'x$ which are contained in the equality group $E_p(i_2)$.

UA is an extension of BSA. The fundamental difference is that UA uses a different mechanism for exact domination value computation, which is shown in Fig. 8. Essentially, the new procedure exploits the results of Propositions 4 and 5. The main idea is to compute the union set U and the set of identical items UE, by scanning the B$^+$-trees again from the beginning in all selected dimensions, and stop when the equality group of the terminating item t has been reached. It is important to note that during scan of retrieving a terminating item, the scanning pointers stop at the same position on B$^+$-trees in all selected dimensions, thus we do not know the exact positions of the detected terminating item and the position intervals of the corresponding equality groups. To detect all these positions correctly and at the same time to compute the sets U and UE, we scan again from the beginning each dimension separately. Therefore, a direct calculation of these sets in the first scanning procedure of detection of a terminating item could not be possible, as we do not know which item will be terminated until its counting closes (to detect its positions and equality groups). The only fact that we know is that all positions of the detected terminating item in the selected dimensions are less than or equal to the position that closed the counting, thus the total number of expected second value accesses of UA in the new calculation procedure will be less than that of detecting the terminating item.

```
int CalculateDomValueUA(I,t)
 1. m=|I| ; U=∅ ; UE=∅
 2. backup current record positions in all m dimensions
 3. for any selected dimension i_s on I (s=1,…,m)
 4.   Ui_s=∅ ; UE_s=∅ ; first=false
 5.   goto first record in dim i_s and retrieve item p and its id
 6.   if p≠t then Ui_s=Ui_s∪{id} else first=true
 7.   if first=false then
 8.     do
 9.       goto next record in dim i_s, retrieve item p and its id
10.       if p≠t then Ui_s=Ui_s∪{id}
11.     while p≠t
12.     get the equality bit eqb of p from current record
13.     if eqb≠0 then
14.       do
15.         goto previous record in i_s,retrieve p,its id and eqb
16.         Ui_s=Ui_s-{id}
17.       while eqb≠0
18.     end-if
19.   end-if
20.   get the point id from the current record
21.   UE_s=UE_s∪{id}
22.   do
23.     goto next record in i_s, retrieve item p, its id and eqb
24.     if eqb≠0 then UE_s=UE_s∪{id}
25.   while eqb≠0
26.   U=U∪Ui_s
27.   if s=1 then UE=UE_s-{id of t} else UE=UE∩UE_s
28. end-for
29. dom(t)=N-1-|U|-|UE|
30. restore the record positions in all m dimensions
31. return dom(t)
```

Fig. 8 Exact domination value calculation for UA

Author's personal copy

It is also important to note that all required sets U, UE, Ui_s, UE_s are not implemented as separate structures and no additional space is required. This is because we are not interested in the specific final items that are contained, but only in their size (e.g., $|U|$, $|UE|$). This means that there is no additional work done in Lines 6, 10, 16, 21, 24, 26, and 27 of Fig. 8 to perform set operations. This also holds for algorithms RA and DA presented later. This is achieved by using specific counters stored in the Set-B$^+$ for the sizes of the sets, which are updated accordingly during the scans.

In the sequel, we describe the process of exact domination value computation of UA depicted in Fig. 8. The outer loop (lines 3–28) performs all necessary scans and set operations by dimension, and line 29 calculates the final exact domination value of the selected terminating item t using the formula of Proposition 4. Line 4 initializes the temporary variables and sets (Ui_s is used for union calculations in dimension i_s, and UE_s is used for keeping the items of the equality group of t in dimension i_s). Lines 5–7 get the first item and check whether it is equal to t. If not, then it is added to Ui_s. Before that, the first B$^+$-tree node is loaded in $Bnode[s]$ by replacing the previous node in memory and the record position is set to the leftmost record of the first equality group in dimension i_s. Lines 8–11 scan forward and add to Ui_s all items until t is found.

During forward scanning, every time the next record lies in a different B$^+$-tree node, the node is first loaded in $Bnode[s]$ by replacing the previous node in memory. In case that t lies in an equality group, then lines 12–18 scan backwards until the first item of this equality group is found and at the same time the discovered items are extracted from Ui_s (these items must not be included in the union set). During backward scanning, every time the previous record lies in a different B$^+$-tree node, the node is loaded in $Bnode[s]$ by replacing the previous node in memory. In the next step, the first item of the equality group of t is added in the UE_s set (lines 20–21), and then lines 22–25 scan all other items of this equality group and add them to UE_s. Finally, lines 26 and 27 update the final sets U, UE.

Example 2 (UA execution) The aforementioned ideas are explained by means of the example used for BSA (see Fig. 6). When t_1 (item C) is extracted from the top of the heap, the calculation phase of the UA algorithm computes the union set $U = \{F, J\}$ and the set of identical items $UE = \oslash$, by making 4 total value accesses from the beginning (see Fig. 6). Then, the domination value of t_1 is calculated as follows: $dom(t_1) = 10 - 1 - 2 - 0 = 7$. When t_2 (item I) is extracted, the calculation phase of the UA algorithm computes the union set $U = \{C, E, F, J\}$ and the set of identical items $UE = \oslash$, by making 7 total value accesses from the beginning. Then, the domination value of t_2 is calculated as follows: $dom(t_2) = 10 - 1 - 4 - 0 = 5$. As no other computations of exact domination values are required, the algorithm terminates performing in total $12+4+7 = 23$ value accesses.

4.3 Reverse algorithm (RA)

Although UA eliminates the drawbacks of BSA, it has a serious limitation: When the extracted terminating items have high positions (especially in anti-correlated data sets), or there are a lot of selected dimensions, the total number of required value accesses in set calculations is significantly increased and this may produce additional I/O cost. Our next proposal, the Reverse Algorithm (RA), takes advantage of the fact that all positions of the detected terminating item in the selected dimensions are less than or equal to the position that closed the counting. Therefore, instead of scanning the B$^+$-trees from the beginning (as UA does), RA scans backwards from the position that closed the counting and stops when the equality group of the terminating item has been scanned in each selected dimension. During this reverse scanning, all discovered items are inserted into a temporary set URi_s, and another set UR is updated by taking advantage of the current contents of the Set-B$^+$, through the following formula:

$$UR = UR\ \mathbf{OR}\ (SetB^+[s]\ \mathbf{XOR}\ URi_s)$$

in every selected dimension i_s, where $SetB^+[s]$ are the contents of Set-B$^+$ recorded from dimension i_s. After all updates, UR finally has the same items as the set U of algorithm UA. Therefore, the domination value is calculated by the same formula as in UA. Figure 9 depicts the exact domination value calculation applied in RA. Due to the similarity of UA and RA, we are not going into a detailed description of the pseudocode. However, we remind that as in UA, set operations are not performed explicitly (e.g., Line 26, Fig. 9); instead, counters are used in the Set-B$^+$ that are updated during forward and backward scanning.

Example 3 (RA execution) To illustrate the idea, we use the same example as before (see Fig. 6). When t_1 (item C) is deheaped, RA computes the XOR between the sets $\{C, E, I\}$ and $\{C, E, I\}$ in dimension i_1 which equals $\oslash$ by retrieving three records in reverse scan, and the XOR between the sets $\{F, J, C\}$ and $\{C\}$ in dimension i_2 which equals $\{F, J\}$ by retrieving one record in reverse scan. The union of the previous resulted sets is $UR = \{F, J\}$. The set of identical items is $UE = \oslash$, and therefore, the domination value of t_1 is calculated as follows: $dom(t_1) = 10 - 1 - 2 - 0 = 7$. We have 4 total reverse value accesses from position of t_1 (see Fig. 6). When t_2 (item I) is deheaped, RA computes the XOR between the sets $\{C, E, I, G\}$ and $\{I, G\}$ in dimension i_1 which equals $\{C, E\}$ by retrieving two records in reverse scan and the XOR between the sets $\{F, J, C, I\}$ and $\{I\}$ in dimension i_2 which equals $\{F, J, C\}$ by retrieving one record in reverse scan. The

Author's personal copy


```
int CalculateDomValueRA(I,t)
 1. m=|I| ; UR=∅ ; UE=∅
 2. backup current record positions in all m dimensions
 3. for any selected dimension i_s on I (s=1,…,m)
 4.    URi_s=∅ ; UE_s=∅
 5.    get the point p and its id from the current record in dim i_s
 6.    URi_s=URi_s∪{id}
 7.    if p≠t then
 8.       do
 9.          goto previous record in i_s, retrieve item p and its id
10.          URi_s=URi_s∪{id}
11.       while p≠t
12.    end-if
13.    get the equality bit eqb of p from the current record
14.    if eqb≠0 then
15.       do
16.          goto previous record in i_s, retrieve p, its id and eqb
17.          URi_s=URi_s∪{id}
18.       while eqb≠0
19.    end-if
20.    get the point id from the current record
21.    UE_s=UE_s∪{id}
22.    do
23.       goto next record in i_s, retrieve p, its id and eqb
24.       if eqb≠0 then UE_s=UE_s∪{id}
25.    while eqb≠0
26.    UR=UR∪(SetB+[s] XOR URi_s)
27.    if s=1 then UE=UE_s-{id of t} else UE=UE∩UE_s
28. end-for
29. dom(t)=N-1-|UR|-|UE|
30. restore the record positions in all m dimensions
31. return dom(t)
```

Fig. 9 Exact domination value computation for RA

union of the previous sets is $UR = \{F, J, C, E\}$. The set of identical items is $UE = \emptyset$, and therefore, the domination value of t_2 is calculated as: $dom(t_2) = 10 - 1 - 4 - 0 = 5$. We have 3 total reverse value accesses from position of t_2. Since no more score computations are needed, the algorithm terminates after performing $12 + 4 + 3 = 19$ value accesses.

4.4 Differential Algorithm (DA)

The final algorithm we study is the Differential Algorithm (DA). The main idea is that when there is a need to compute the exact domination value of a selected terminating item t, we select a previously determined *convenient* terminating item t_p whose domination value has been already computed. Forward and backward scans are performed taking into consideration the position of t_p. Therefore, if item t is positioned after t_p in a selected dimension, then the algorithm scans forward from t_p to t in the B$^+$-tree of this dimension and increases specific counters into the Set-B$^+$ structure for the visited items, otherwise it scans backwards and decreases specific counters into the Set-B$^+$ structure for the visited items. Finally, the exact domination value of the terminating item t is computed using the updated visits set counters and the same formula as in UA.

When the first terminating item is detected, its U set is calculated, as in UA algorithm, and it is stored in a temporary set denoted as UD. During scanning, between the positions of any next terminating item t and t_p, UD is updated from all i_s dimensions ($UD[1], \ldots, UD[m]$) in order to keep exactly the same items with the set U of the UA algorithm. UE is computed again with the same way as previously. The resulting formula is:

$$dom(t) = N - 1 - |UD[1] \cup \cdots \cup UD[m]| - |UE|$$

DA uses the positions of any determined terminating item t in the selected dimensions, which are retrieved during scanning and stored into the Set-B$^+$ structure. Therefore, when t is the leftmost item of its corresponding equality group in a selected dimension i_s, we record its position value $Lpos_t(i_s)$ by using the current value of the global position pointer pos, thus: $Lpos_t(i_s) = pos$. Otherwise, we record its position value by using the same $Lpos$ value with the leftmost item of its corresponding equality group. Therefore, all items of the same equality group in a selected dimension have the same $Lpos$ value.

The most "preferable" terminating item t_p that DA uses is derived from the previous terminating items in which their exact domination values have already been computed, and it is the item that minimizes the total number of required scans between t and t_p. To detect that item, we can perform an exhaustive check into the main heap between t and any other item t_h of the heap that has an exactly calculated domination value, and select the appropriate t_h that minimizes the sum $\sum_{s=1}^{m} \left|Lpos_t(i_s) - Lpos_{t_h}(i_s)\right|$. However, this approach has two basic disadvantages as follows: (a) the exhaustive check adds a significant computational and I/O cost into the algorithm and (b) the updated visit counters of all t_h items must be stored separately using a significant amount of additional space in Set-B$^+$.

Therefore, we designed an alternative solution which performs an efficient estimation on this minimization, and it is based on the rule of the *minimum absolute difference of positions preservation*: When the exact domination value of a terminating item t has been computed, we check whether its $Lpos$ positions satisfy the condition $\sum_{s=1}^{m} \left|Lpos_t(i_s) - Lpos_{t_b}(i_s)\right| < bestDiff$, where t_b is the previous selected best item, and $bestDiff$ is the previously calculated best difference, and we update and keep the corresponding values as necessary. Thus, if the condition holds, t becomes the new best item. The application of this rule lies in the fact that as the global scanning pointer pos grows, the subsequent terminating items decrease their attribute distances in a non-ascending order. Therefore, as pos grows, the previous selected best terminating item will be

Author's personal copy


```
int CalculateDomValueDifferential(I,t)
 1. m=|I| ; UE=∅ ; backup UD
 2. backup current record positions in all m dimensions
 3. if tb=∅ then     [1st Time of calculation]
 4.   execute CalculateDomValueUnion(I,t)
 5.     using UD[s] instead of U,Ui_s
 6.   tb=t
 7.   Lpos_tb(i_s)=Lpos_t(i_s), (∀s=1,…,m)
 8. else    [All Next Times of calculation]
 9.   for any selected dimension i_s on I (s=1,…,m)
10.     UE_s=∅
11.     set current record in position of tb on dim i_s
12.     if Lpos_tb(i_s)>Lpos_t(i_s) then
13.       do
14.         goto previous record in dim i_s and
15.           retrieve item p, its id and eqb
16.         UD[s]=UD[s]-{id}
17.       loop until eqb=0 and Lpos_p(i_s)=Lpos_t(i_s)
18.     end-if
19.     if Lpos_tb(i_s)<Lpos_t(i_s) then
20.       do
21.         retrieve item p, its id and eqb
22.         if eqb≠0 or Lpos_p(i_s) ≠Lpos_t(i_s) then
23.           UD[s]=UD[s] ∪{id}
24.         goto next record in dim i_s
25.       loop until eqb=0 and Lpos_p(i_s)=Lpos_t(i_s)
26.     end-if
27.     get the point p and its id from the current record
28.     UE_s=UE_s∪{id}
29.     do
30.       goto next record in i_s, retrieve item p, its id and eqb
31.       if eqb≠0 then UE_s=UE_s∪{id}
32.     while eqb≠0
33.     if s=1 then UE=UE_s-{id of t} else UE=UE∩UE_s
34.   end-for
35. end-if
36. dom(t)=N-1-|(UD[1] OR … OR UD[m])|-|UE|
37. restore the record positions in all m dimensions
38. Diff = Σ_{s=1}^{m} |Lpos_tb(i_s) – Lpos_t(i_s)|
39. if Diff<bestDiff then
40.   tb=t ; bestDiff=Diff
41.   Lpos_tb(i_s)= Lpos_t(i_s), (∀s=1,…,m)
42. else restore UD
43. return dom(t)
```

Fig. 10 Exact domination value computation for DA

closer to the current terminating item, minimizing the absolute difference of their positions and consequently, reducing the total number of required scans for the domination calculations. The best computed difference is stored in the variable $bestDiff$.

The exact domination value computation algorithm for DA is given in Fig. 10. The algorithm is composed of three parts: (1) the *initialization and backup phase* (lines 1–2), (2) the *calculation phase* (lines 3–36), and (3) the *restore and return phase* (lines 37–43).

In the *initialization and backup phase*, we initialize m and set UE, and we backup the current status of UD (line 1). We also backup the current B^+-tree record positions and the current B^+-tree nodes in all the selected dimensions, because after calculations we must return the B^+-tree's control into those record positions before we return to the main algorithm.

The *calculation phase* has two parts: (1) lines 3–7, which are called for calculations of the first determined terminating item and (2) lines 8–35, which are executed for all next discovered items. In the first part, calculations are similar to the corresponding ones of the UA algorithm (lines 3–28 of Fig. 8). Thus, the first time we scan the B^+-trees of the selected dimensions from the beginning, we update the UD sets, and the first terminating item becomes the current best item. The differences between this part and UA are as follows: (1) the UD sets have replaced the union sets U, Uis, (2) the updates of the $Lpos$ positions of the current best item have been added. Note that in this part, the UD sets contain all discovered items lying before the equality groups of the first terminating item in the selected dimensions and that in the next part of the algorithm the UE calculations (lines 27–33) remain the same as in UA. In the second part, for any selected dimension (line 9), we retrieve the $Lpos$ position values of the current terminating item (t) and the current best item (t_b) from the Set-B^+, and we initialize UE_s (line 10). Then, we set the current B^+-tree record position and the current B^+-tree node in the position and the node of the best item (line 11). There are three possible cases:

- If $Lpos_{t_b}(i_s) = Lpos_t(i_s)$, the two terminating items t_b, t lie into the same equality group of dimension i_s, thus there is no need of any updates or calculations in the UD sets.
- If $Lpos_{t_b}(i_s) > Lpos_t(i_s)$, the best terminating item (t_b) lies after the current (t) in a different equality group on dimension i_s, thus we scan backwards from the best to the current terminating item and we remove the found items from the UD set on that dimension (lines 12–18).
- If $Lpos_{t_b}(i_s) < Lpos_t(i_s)$, the best terminating item lies before the current in a different equality group on dimension i_s, thus we scan forward from the best to the current terminating item, inserting the items seen into the UD set on that dimension (lines 19–26).

In the last two cases, the UD sets are updated to hold all the items lying before the equality groups of the current terminating item. Finally, line 36 computes the exact score of the current terminating item.

In the *restore and return phase*, we restore the current record positions of all B^+-trees and all B^+-tree nodes in the selected dimensions to the one they had before the execution of this part (line 37). Then, we check whether the current terminating item (t) satisfies the rule of the *minimum absolute difference of positions preservation* in comparison

Author's personal copy

with the current best item (t_b) (lines 38,39). If this is true, it becomes the new best item (line 40) and we update all necessary variables (lines 40–41); otherwise, the best item remains unchanged and we restore the values of the UD sets (line 42). Finally, the calculated domination value of the current terminating item is returned (line 43).

Example 4 (DA execution) We use again the same running example as in BSA, UA, and RA (see Fig. 6). When t_1 (item C) is deheaped, the part-A calculation phase of the DA algorithm is executed which computes the set $UD = \{F, J\}$ (the union of the separate $UD[s]$ sets), and the set of identical items $UE = \oslash$, by making 4 total value accesses from the beginning. Then, the domination value of t_1 is calculated: $dom(t_1) = 10-1-2-0 = 7$, and t_1 becomes the best item. When t_2 (item I) is extracted, the set $UD = \{C, E, F, J\}$ is updated by performing one additional forward value access from the current best item (item C) to I (note that there are no intermediate items between C and I in dimension i_2). The set of identical items is again $UE = \oslash$. Then, the domination value of t_2 is calculated as: $dom(t_2) = 10 - 1 - 4 - 0 = 5$. Since no other computations are required, the algorithm has performed $12 + 4 + 1 = 17$ value accesses.

5 Analytical study

In this section, we provide an analytical study toward estimating the performance of the algorithms. More specifically, we study the following issues: (1) the expected number of terminating items that are likely to be discovered at a particular position, (2) the expected domination score of any terminating item discovered at a particular position, and (3) the expected storage requirements of the heap, where terminating items are organized. In addition, at the end of the section, we give the worst-case complexity of the algorithms.

Let $\mathcal{X}_j$, $1 \leq j \leq m$ be discrete random variables representing the attribute values of the objects in each dimension. To keep the analysis simple, we assume that each random variable $\mathcal{X}_j$ takes discrete values in the interval $[1, \ldots, N]$, where N is the total number of objects.[3] The most general case appears when the random variables $\mathcal{X}_j$ depend on each other. Therefore, let $F(x_1, \ldots, x_m)$ be the *joint cumulative distribution function* (CDF) of all $\mathcal{X}_j$ random variables. If the distribution is unknown, approximations may be applied (see for example [8]).

5.1 Terminating items and domination values

First, we compute the expected number of terminating items that may be discovered up to position ξ, $1 \leq \xi \leq N$, in the sorted order of attribute values. Recall that, an item is declared as a terminating item if all its attribute values have been reached up to position ξ. Consequently, the probability P_ξ that an object is a terminating item up to position ξ is determined by means of the joint CDF as follows:

$$P_\xi = P(\bigcap_{j=1}^{m} \mathcal{X}_j \leq \xi) = F(\xi, \xi, \ldots, \xi)$$

For simplicity, we use the notation $F(\xi)$ to express the fact that the joint CDF is computed for the same argument ξ in all dimensions. Let $\mathcal{T}_\xi$ denotes the number of terminating items that exist up to position ξ. Since the total number of objects is N, the expected number of terminating items $E[\mathcal{T}_\xi]$ is given by:

$$E[T_\xi] = N \cdot P_\xi = N \cdot F(\xi) \tag{1}$$

Next, we study the domination score of terminating items. Let l be the last found terminating item when the scanning position pointer has the value of ξ. Then, l has at least one rank position equal to ξ, meaning that in that particular dimension l dominates all $N - \xi$ items located at the right of ξ. However, this may not be the case for the other dimensions. If D_ξ denotes the domination score of a terminating item discovered at ξ, then clearly:

$$D_\xi \leq N - \xi \tag{2}$$

It is evident that the domination score of a terminating item at ξ is at least the number of objects whose all m attribute values are located at positions larger than ξ. Let $P_{\xi+}$ denotes the probability that all attribute values of an item are located at positions larger than ξ:

$$P_{\xi+} = P(\bigcap_{j=1}^{m} \mathcal{X}_j > \xi)$$

To compute $P_{\xi+}$, we need the *complementary* joint CDF F^c, which is generally defined as follows:

$$P(\mathcal{X}_1 > x_1, \mathcal{X}_2 > x_2, \ldots, \mathcal{X}_m > x_m) = F^c(x_1, x_2, \ldots, x_m)$$

In our case, we have:

$$P(\bigcap_{i=1}^{m} \mathcal{X}_j > \xi) = F^c(\xi)$$

One way to determine $P(\cap_{i=1}^{m} \mathcal{X}_j > \xi)$ is to use the inclusion/exclusion principle as follows:

$$P(\bigcap_{i=1}^{m} \mathcal{X}_j > \xi) = 1 - P(\cup_{i=1}^{m} \mathcal{X}_j \leq \xi) \tag{3}$$

To compute the expected number of items $E[D_\xi]$ dominated by a terminating item determined at position ξ, we may use the following:

$$E[D_\xi] \geq N \cdot P_{\xi+} = N \cdot F^c(\xi) \tag{4}$$

[3] Generalizations are possible by using results from probability theory and statistics, but such a direction is out of the scope of the paper.

Author's personal copy

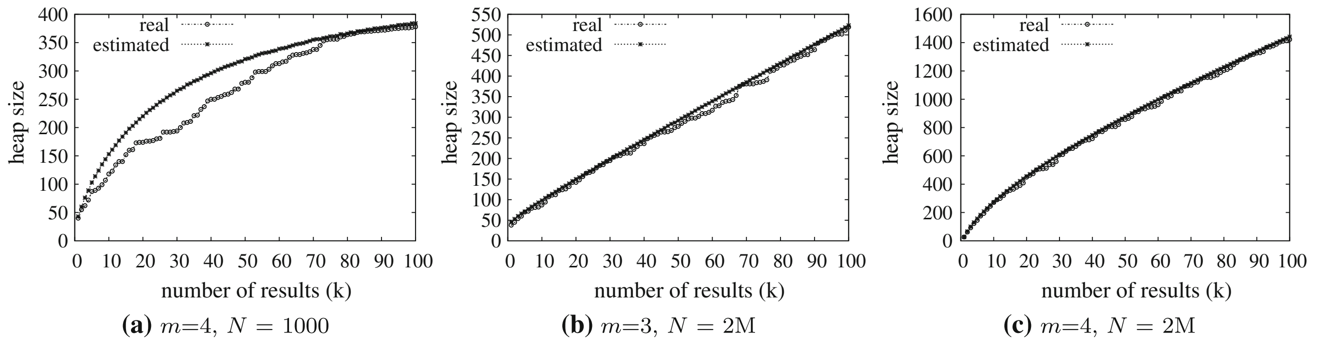

Fig. 11 Estimated and real heap size vs. k for normal multivariate distributions

5.2 Memory consumption

Next, based on the previous discussion, we estimate the expected number of terminating items maintained into the heap data structure. Note that the heap storage requirements increase as more sorted access are performed, because as shown by Eq. 1, the number of terminating items increases monotonically with the scan position ξ. Therefore, if the random variable H denotes the number of terminating items inserted into the heap, we are interested in determining its expectation $E[H]$. Recall that the heap maintains all terminating items discovered so far, minus the terminating items that have been deheaped due to their participation in the current top-k results. Let $k' \leq k$ be the number of the deheaped terminating items when we are at position ξ. Therefore, at ξ, the value of $E[H]$ satisfies the following:

$$E[H] \leq E[T_\xi] - k' \tag{5}$$

We center our focus in the case where $k = 1$. Let t_1 be the first terminating item which has been discovered at position ξ_1. Let ξ_f be the position where query processing terminates. Either t_1 is the best item and therefore it will be eventually deheaped or another terminating item t_x will be deheaped before t_1 because $dom(t_x) > dom(t_1)$. Our plan is to delay the extraction of the best item as much as possible to overestimate the expected heap size. From F, we can define the inverse (quantile) function F^{-1} [11] which determines a value corresponding to the probability given as input. Based on this, the position ξ_1 where the expected number of terminating items is 1 is determined as follows:

$$N \cdot F(\xi_1) = 1 \Rightarrow \xi_1 = F^{-1}(1/N) \tag{6}$$

According to Formula 4, the expected domination score of the terminating item discovered at ξ_1 is at least $N \cdot F^c(\xi_1)$ and substituting the value of ξ_1 from the previous equation, we have that:

$$\overline{dom(t_1)} \geq N \cdot F^c(F^{-1}(1/N)) \tag{7}$$

Our last step is to determine the position ξ_f where the terminating item will be deheaped, meaning that it is the item with the highest domination score. This will happen when $\overline{dom(t_1)} \geq N - \xi_f$. Therefore, the value of ξ_f is determined by using the equation $N \cdot F^c(F^{-1}(1/N)) = N - \xi_f$, and thus:

$$\xi_f = N - N \cdot F^c(F^{-1}(1/N)) \tag{8}$$

By substituting the value of ξ_f in Eq. 5 and using 1, we get that the expected heap size for the execution of a top-1 dominating query satisfies the following:

$$E[H] \leq N \cdot F(N - N \cdot F^c(F^{-1}(1/N))) - 1 \tag{9}$$

Using the same rationale, we generalize for any value of k, by considering the position ξ_k where the expected number of terminating items is k. As before, query processing stops when exactly k terminating items have been deheaped. The k-th best item will be deheaped at ξ_f (final position) when its score is larger than or equal to $N - \xi_f$. The expected heap size in this case satisfies the following inequality:

$$E[H] \leq N \cdot F(N - N \cdot F^c(F^{-1}(k/N))) - k \tag{10}$$

In conclusion, the expected memory consumption equals the k best items plus the expected heap size. Figure 11 depicts the estimated and the real heap size in relation to the value of k. The data follows a multivariate normal distribution in 3 and 4 dimensions. In all cases, the estimated size of the heap is very close to the computed value. More specifically, for 1K objects in the 3-d space (Fig. 11a) the average estimation error was less than 15%, whereas for 2M objects in 3-d and 4-d (Fig. 11b, c) the average error was less than 5%.

We note that a more detailed analysis could involve the following: (1) the study of specific confidence intervals in order to provide probabilistic guarantees for the results, (2) the relaxation of the assumption that attribute values are drawn from a discrete probability distribution. In addition, better bounds could be achieved by using a different approach to determine the position where at least k terminating items have been discovered. In Eq. 6, we have used the quantile function

Author's personal copy

which determines the position in order for the *expected* number of terminating items to be 1. By using Bernoulli trials, we could determine more accurately the position where *at least one* terminating item is found. However, such an approach leads to more complex equations. According to our results, the analysis presented previously is adequate, since the estimation error remains low.

5.3 Complexity analysis

In the sequel, we perform a complexity analysis regarding the number of I/O operations required by the algorithms. We will ignore the impact of the buffer, assuming that every page request results in a disk I/O operation. For the rest of the discussion, we denote by ξ_f the final scanning position where the last best item has been reported and therefore query processing terminates. We denote by b the (minimum) branching factor of all B$^+$-trees participating in query processing and also the minimum number of keys that may be hosted in a disk page. First, we analyze BSA and then we proceed with the rest of the algorithms.

Assume that BSA terminates at scanning position ξ_f. Let λ denotes the number of terminating items discovered up to position ξ_f. The number of I/Os required to reach the position ξ_f is at most $m\xi_f/b$. For the j-th terminating item, let ξ_j be the scanning position where this has been discovered. According to BSA, $(N-\xi_j)/b$ disk accesses are required to scan the corresponding dimension to the end. For each terminating item found, BSA determines its domination value by scanning one dimension to the end requiring $(N-\xi_j)/b$ I/Os and by performing random accesses in the rest m-1 B$^+$-trees. These random accesses will search all items belonging to the same equality groups in each dimension, and we denote by Q the maximum cardinality of all equality groups. If $Q=1$, then only unique values are present. Based on the previous discussion, the I/O cost of BSA has as follows:

$$C_{\mathrm{BSA}} = \frac{m\xi_f}{b} + \sum_{j=1}^{\lambda} \frac{N-\xi_j}{b} + \sum_{j=1}^{\lambda} (N-\xi_j) \times \log_b \frac{m\xi_j}{b} (m-1) \left(\log_b \frac{N}{b} + \frac{Q-1}{b} \right)$$

To derive an upper bound for the previous result, we use ξ_f in place of ξ_j and set $\lambda = \xi_f$ since the number of terminating items discovered at position ξ_f is at most ξ_f. Therefore, the worst-case complexity of BSA is given by:

$$C_{\mathrm{BSA}} \in O\left(\xi_f m N \log_b \left(\frac{m\xi_f}{b} \right) \left(\log_b \left(\frac{N}{b} \right) + \frac{Q-1}{b} \right) \right)$$

Next, we study the complexity of UA. Recall that every time a terminating item is discovered, UA rescans from the beginning in order to determine its domination score. For each item scanned, a lookup is performed in the Set-B$^+$-tree to locate the item and perform the necessary counter updates. If we are rescanning up to position ξ_j, the maximum number of items inserted in the Set-B$^+$-tree is $m\xi_j$. Thus, each lookup in the Set-B$^+$-tree costs $\log_b((m\xi_j)/b)$ I/Os. Summing for all terminating items detected and denoting (as previously) by ξ_f the position where the algorithm terminates, the I/O cost of UA is given by:

$$C_{\mathrm{UA}} = \lambda \frac{m\xi_f}{b} + \sum_{j=1}^{\lambda} m\xi_j \log_b \frac{m\xi_j}{b}$$

Again, by upper bounding the previous formula and after dropping the least significant terms, we get:

$$C_{\mathrm{UA}} \in O\left(m\xi_f^2 \log_b \frac{m\xi_f}{b} \right)$$

By using the same arguments, it is not hard to realize that the worst-case complexity of RA and DA matches that of UA. We note, however, that in a typical case DA is expected to outperform the other algorithms and also its cost is much lower than the worst-case. DA applies a more sophisticated mechanism for score computation, which has a significant impact on response time reduction. This fact is shown in the experimental results reported in Sect. 7.

6 Pruning techniques

To further decrease the number of value accesses and the corresponding I/O operations, a collection of pruning rules has been designed toward performance boost. We note however, that by using pruning, when the k bet items have been determined, it is not possible to request the (k+1)-th. However, pruning does not harm the progressiveness property of our algorithms, since items are reported in non-increasing score order.

There are three types of pruning rules that we propose, according to their applicability to the algorithm parts: (1) *Discard Rules (DRs)*, which update all items that can be discarded during the scanning for the next candidate terminating item, (2) *Early Pruning Rules (EPRs)*, which are applied before the computation of the exact domination value of the selected terminating item, and (3) *Internal Pruning Rules (IPRs)*, which are applied during exact score computation.

All pruning rules depend directly on a global pruning value G, which defines a lower bound for the exact domination values such that any candidate item t with $dom(t) \leq G$ can safely be pruned. G is initialized to 0, and it is updated every time the heap changes its exactly calculated top-k item. Thus, from the first time that the heap has k exactly calculated items (and after that point), G will be always equal to the exact domination value of the current top-k item (t_k) of the heap minus 1, i.e., $G = dom(t_k) - 1$.

Author's personal copy

To distinguish between the U and UE sets of different items, we additionally denote these sets as U_p and UE_p for the corresponding item p. We also denote as $\overline{X}$ the complement of X.

6.1 Discard Rules

Discard Rules (DRs) are using the fundamental property that if x dominates y, then $dom(x) > dom(y)$.

Rule 1 *(Discard Rule DR1): If t_k is the current exactly calculated top-k terminating item of the heap, then any item of the set $\overline{U_{t_k} \cup UE_{t_k} \cup \{t_k\}}$ can be safely discarded.*

Rule 2 *(Discard Rule DR2): If t_p is an item that has been pruned by any other rule, then any item of the set $\overline{U_{t_p} \cup UE_{t_p} \cup \{t_p\}}$ can be safely discarded.*

Rule 3 *(Discard Rule DR3): When the first exact top-k terminating item of the heap has been calculated, then any item of the set $\overline{SetB^+}$ (i.e., any item that has not been inserted into the Set-B$^+$ yet) can be safely discarded.*

To update the status of the items, an additional bit (discard bit) is used in the records of the Set-B$^+$ tree. Initially, any item that is inserted in the Set-B$^+$ takes a discard bit equal to zero (non-discarded item). But when the first exactly calculated top-k terminating item has been found, due to DR3, all next inserted items must be discarded. Therefore, after that moment, any new item that is inserted in the Set-B$^+$ takes a discard bit equal to one (discarded item).

6.2 Early Pruning Rules

Early Pruning Rules (EPRs) are applied before the calculation of the exact domination value of the selected terminating item (in line 12 of the algorithm of Fig. 3), and they are based on the main property of the *dom* function, the global pruning value G, and several other properties.

Rule 4 *(Early Pruning Rule EPR1):If t_k is the current exactly calculated top-k terminating item of the heap, t is the current selected terminating item for calculations, and $t_k \prec t$, then t cannot be included into the final top-k result, thus can be safely pruned.*

Rule 5 *(Early Pruning Rule EPR2): If t_i is any exactly calculated terminating item that is after the current exactly calculated top-k terminating item in the heap, and t is the current selected terminating item for calculations, and $t_i \prec t$, then t cannot be included into the final top-k result.*

These first two Early Pruning Rules (EPR1, EPR2) include domination checks between the specific terminating items. It is important to note that these domination checks are not performed using the attribute values of the items, but using the $Lpos$ position values in the selected dimensions as defined in Sect. 4.4, which have already been recorded in the Set-B$^+$. The equivalent condition for the domination checks is the following:

$$t_j \prec t_i \Leftrightarrow \forall i_s \in I, Lpos_{t_i}(i_s) \geq Lpos_{t_j}(i_s)$$
$$\wedge \exists i_s \in I : Lpos_{t_i}(i_s) > Lpos_{t_j}(i_s)$$

All the following early pruning rules make also use of the $Lpos$ position values.

Rule 6 *(Early Pruning Rule EPR3): If t is the current selected terminating item, it can be safely pruned if the following holds: $N - max_{i_s}\{Lpos_t(i_s)\} \leq G$.*

Rule 7 *(Early Pruning Rule EPR4): If t is the current selected terminating item and the following condition is satisfied:*

$$N - |SetB^+| - 1 + \sum_{s=1}^{m} [pos - Lpos_t(i_s)] + a \leq G$$

then t can be safely pruned. (The notations of the variables are the same with Fig. 3).

Proof When the current terminating item t is selected, its positions are as in Fig. 12 in the selected dimensions. Let A_s be the set that contains all items from the leftmost item of equality group of t on dimension i_s (i.e., from $Lpos_t(i_s)$) until the last reached position during scan (i.e., to pos). Then, using Proposition 4 for the item t, we have:

$$dom(t) = N - 1 - |U_t| - |UE_t| \leq N - 1 - |U_t|$$

Thus, the number of items into the union-set of t can be computed as: $|U_t| = |SetB^+| - |A_1 \cup A_2 \cup \cdots \cup A_m|$. Then, by combining with the inequality: $|A_1 \cup A_2 \cup \cdots \cup A_m| \leq |A_1| + |A_2| + \cdots + |A_m|$, we take:

$$dom(t) \leq N - 1 - |U_t| \Rightarrow$$
$$dom(t) \leq N - 1 - |SetB^+| + |A_1 \cup A_2 \cup \cdots \cup A_m| \Rightarrow$$
$$dom(t) \leq N - |SetB^+| - 1 + |A_1| + |A_2| + \cdots + |A_m|$$

But also, the number of items in any A_s set is equal to $pos - Lpos_t(i_s) + 1$ (if $s \leq a$), and $pos - Lpos_t(i_s)$ (if $s > a$), as

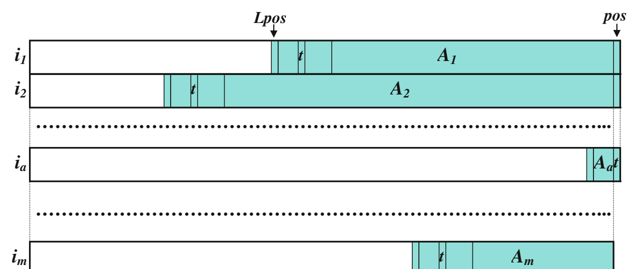

Fig. 12 Proof of EPR4

Author's personal copy

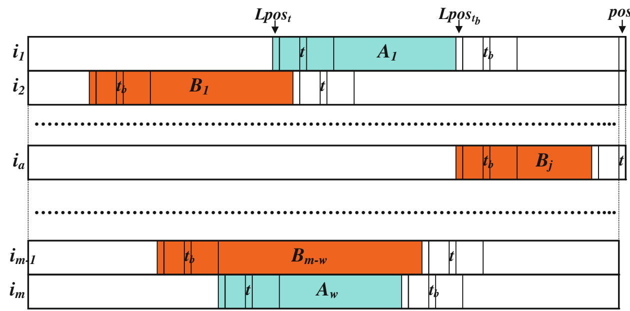

Fig. 13 Proof of EPR5

all item id's are distinct in a specific dimension. Therefore, we have:

$$dom(t) \leq N - |SetB^{+}| - 1 + \sum_{s=1}^{m} [pos - Lpos_t(i_s)] + a$$

and since $dom(t) \leq G$, t is eliminated. □

Rule 8 *(Early Pruning Rule EPR5): If t is the current selected terminating item, t_b is the current best terminating item (using the same criteria as in DA), and the following condition is satisfied:*

$$dom(t_b) + |UE_b| + \sum_{i_s : Lpos_{t_b} > Lpos_t} \left[Lpos_{t_b}(i_s) - Lpos_t(i_s) \right]$$

$\leq G$, then t can be safely pruned.

Proof When the current terminating item t is selected, its positions are as in Fig. 13 in the selected dimensions. In the same figure, the positions of the current best terminating item are depicted too. As we can see, in some dimensions t may lie before t_b and in other dimensions the opposite may hold.

Let $A_1, A_2, \ldots, A_w$ be the sets containing all items from the leftmost item of the equality group of t ($Lpos_t$), to the leftmost item of the equality group of t_b ($Lpos_{t_b}$), in the dimensions that t lies before t_b. Let also $B_1, B_2, \ldots, B_{m-w}$ be the sets containing all items from the leftmost item of the equality group of t_b ($Lpos_{t_b}$), to the leftmost item of the equality group of t ($Lpos_t$), in the dimensions that t_b lies before t.

Then, for the items t, t_b and their corresponding union sets U_t, U_{t_b} (and by using the order of all new defined sets), the following inequality holds: $|U_t| + |A| \geq |U_{t_b}|$, where ($A = A_1 \cup A_2 \cup \cdots \cup A_w$). Thus, we take:

$$-|U_t| - |A| \leq -|U_{t_b}| \Rightarrow N - |U_t| - 1 \leq N - |U_{t_b}| - 1 + |A|$$

Using Proposition 4 for the items t, t_b, we have:

$$dom(t) = N - 1 - |U_t| - |UE_t|$$
$$dom(t_b) = N - 1 - |U_{t_b}| - |UE_{t_b}|$$

Therefore, the following holds:

$$dom(t) \leq N - 1 - |U_t| \Rightarrow$$
$$dom(t) \leq N - |U_{t_b}| - 1 + |A| \Rightarrow$$
$$dom(t) \leq dom(t_b) + |UE_{t_b}| + |A| \Rightarrow$$
$$dom(t) \leq dom(t_b) + |UE_{t_b}| + |A_1| + |A_2| + \cdots + |A_w|$$

However, the number of items in any A_i set is equal to $Lpos_{t_b}(i_s) - Lpos_t(i_s)$, as all item id's are distinct in a specific dimension. Thus, we have:

$$dom(t) \leq dom(t_b) + |UE_{t_b}| + \sum_{Lpos_{t_b} > Lpos_t} \left[Lpos_{t_b}(i_s) - Lpos_t(i_s) \right]$$

and since $dom(t) \leq G$, t is eliminated. □

Rule 9 *(Early Pruning Rule EPR6): If t_i is any exactly calculated terminating item of the heap, t is the current selected terminating item, and during calculations, the following condition is satisfied:*

$$dom(t_i) + |UE_{t_i}| + \sum_{i_s : Lpos_{t_i} > Lpos_t} \left[Lpos_{t_i}(i_s) - Lpos_t(i_s) \right]$$

$\leq G$, then t can be safely pruned.

Proof Similar to the proof of EPR5. □

EPR rules enable early pruning, since they discard terminating items before computing their exact scores. All EPR rules make significant improvements to the global performance of the top-k dominating queries, as many terminating items are pruned before the calculations of their domination values. Rules EPR1, EPR2, EPR3, EPR4 can be applied in all algorithms, as they only require the $Lpos$ position values and general variable values. Rule EPR6 can be applied in all algorithms except BSA. EPR5 is applied only in algorithm DA since it uses the current best terminating item.

6.3 Internal Pruning Rules

Internal Pruning Rules (IPRs) are applied during the calculation of the exact domination value of the selected terminating item (inside the $CalculateDomValue$ functions). They are based on the previous presented early pruning rules and they detect the total number of remaining required scans for the score computation and if the derived value is less than or equal to G, then the item is discarded. More specifically, rules EPR3 through EPR6 can be transformed to IPRs using the above rationale. However, the most important IPRs that contribute the most in query performance improvement are based on EPR4:

Author's personal copy

Table 1 Algorithms and properties

Algo	Properties
BSA*	Basic Scan Algorithm, progressive, with pruning rules: G, DR3, EPR1, EPR2, EPR3, EPR4
UA*	Union Algorithm, progressive, with pruning rules: G, DR1, DR2, DR3, EPR1, EPR2, EPR3, EPR4, EPR6, IPR1
RA*	Reverse Algorithm, progressive, with pruning rules: G, DR1, DR2, DR3, EPR1, EPR2, EPR3, EPR4, EPR6, IPR1
DA*	Differential Algorithm, progressive, with pruning rules: G, DR1, DR2, DR3, EPR1, EPR2, EPR3, EPR4, EPR5, EPR6, IPR1
CBT	Cost-Based Traversal Algorithm, Non Progressive
FNP	Fine-graiNed with Partial dominance Algorithm, Non Progressive
SS	Sample and Sort Algorithm (k-Most Connected Vertex method), Non Progressive
SOE	Switch On Empty Algorithm (k-Most Connected Vertex method), Non Progressive

Rule 10 *(Internal Pruning Rule IPR1): If t is the current selected terminating item, then t can be pruned if the following holds:*

$$N - |SetB^{+}| + curDom(t) + \sum_{s=1}^{m} |curPos(i_s) - Lpos_t(i_s)| \leq G$$

The notation $curDom(t)$ expresses the current recorded domination value of item t.

In set-union calculations, an item is dominated by t if its main visits counter in Set-B^{+} becomes zero during forward or backward value accesses. The notation $curPos(i_s)$ expresses the current position of the examined record in dimension i_s during forward or backward scanning in calculations. Therefore, in every update of the variable values $curDom(t)$, $curPos(i_s)$, we check the condition of IPR1 to decide either to continue or skip the rest of the computations.

7 Performance evaluation study

All experiments have been conducted based on real-life and synthetic data sets using different parameter values. The algorithms have been implemented in C++, and all experiments have been conducted on a Pentium 4 machine with 3 GHz CPU running Windows XP. Table 1 depicts the algorithms compared and their properties. We use the symbol '*' in the algorithms' notation to emphasize that pruning is enabled. Note that efficiency depends on the order that pruning rules are applied. We have conducted several experiments, and we observed that the overall best order for applying the pruning rules is as follows: DR3,DR1, DR2, EPR4, EPR3, EPR1, EPR5, EPR6, EPR2, IPR1.

We have used three synthetic data sets, namely *correlated* (COR), *independent* (IND) and *anti-correlated* (ANT), generated using the methodology reported in [4]. Additionally, two real-life multi-dimensional data sets have been used as follows: (1) the *Forest Cover* (FC) data set (http://kdd.ics.uci.edu), which contains 581,012 forest land cells with 55 attributes, from which the first 10 attributes have been used for expressing positions, distances to roads, fireplaces, hydrology, etc and (2) the *Zillow* (ZIL) data set (http://www.zillow.com), which contains 2M 5-dimensional real estate data [26]. We selected the 1,252,208 records, because in the rest there were missing values.

Table 2 Parameters and values

Parameter	Values	Default
Data set cardinality (N)	1M, 2M, 3M, 4M	2M
Number of dimensions (m)	2, 3, 4, 5, 6, 7	4
Number of results (k)	1, 10, 25, 50, 75, 100	20
Buffer size (% of db size)	10, 15, 20, 25, 30	20

Table 2 shows the parameters and the corresponding values used in the performance evaluation. Unless otherwise specified, the parameters take the default values. We assume that the disk page capacity is 4KBytes. In all experiments, we have used buffer sizes as percentages of the total database size (the size of all B^{+}-trees on disk). Running time is measured by the sum of the CPU time and the I/O time. I/O time is computed by considering 10 ms for each page fault.[4] Dimensionality values express the number of the m out of d attributes that have been selected. We vary m from 2 to 7 in synthetic data sets, 2–10 in FC and 2–5 in ZIL. The dimensionality experimental results are averages of m random attribute selections from the total number of dimensions using all possible combinations, except in anti-correlated cases where always an anti-correlated attribute is selected.

7.1 Performance of studied algorithms

In the first experiment series, we examine the progressiveness property of our algorithms by recording the time stamps of the reported top results. Figure 14 depicts representative

[4] The same process has been followed in previous works as well, such as [29,30].

Author's personal copy

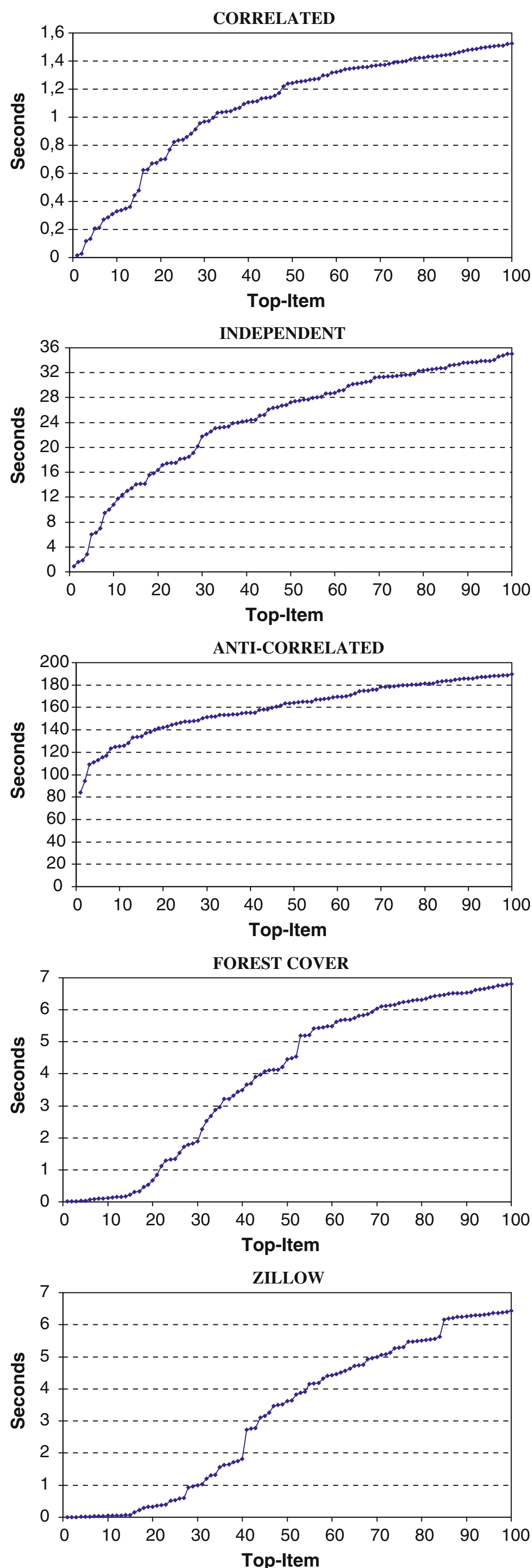

Fig. 14 Top-100 progressiveness results

time stamps of the first 100 results for the DA* algorithm (we have a similar behavior in all algorithms). By inspecting the graph, the significance of the progressiveness property becomes evident. Progressiveness enables the delivery of the first results without waiting for the computation of the complete result set. This has a direct impact to users who may inspect the first results while waiting for more. In addition, processing may be terminated at will if the user is satisfied with less than k results.

Next, we compare the performance of all algorithms by varying the cardinality and keeping all other parameters to their default values. The corresponding results are given in Fig. 15. We observe that as we increase the data set size, the performance cost increases. In COR, IND, and real data, the performance of UA*, RA*, and DA* is similar, while that of BSA* is significantly worse. In ANT data, the performance cost of UA* increases significantly and reaches that of BSA*, due to the fact that the terminating items close their counts in large scan positions, thus there are large scan intervals and a higher I/O cost. RA* and DA* perform consistently better managing to keep CPU and I/O costs at low levels. Finally, DA* outperforms all other algorithms as the size increases.

In the sequel, we compare the algorithms by varying the dimensionality of all data sets, keeping the other parameters to their default values. Figure 16 depicts the performance results. We observe that as we increase the dimensionality, the performance cost increases, as expected. This is because it is harder to discover the terminating items in higher dimensions due to the additional scans required in each dimension. BSA* is significantly worse than the other algorithms in all cases. Again, in ANT data, the performance of UA* increases significantly but it performs better than BSA*. Moreover, RA* and DA* perform significantly better, whereas DA* shows the best overall performance.

In the next series of experiments, we vary the number of top results (k), keeping the other parameters to their default values. Figure 17 depicts the results. We observe that as we increase the number of results, the performance cost increases almost linearly in all cases and for all algorithms. BSA* is again worse than the others in all cases, incurring significant CPU and I/O cost. In the real-life data sets UA*, RA* and DA* show similar performances, but DA* is consistently better. The limitation of UA* to handle ANT data is again obvious, since its cost increases significantly due to the additional accesses performed. Among all algorithms, DA* shows the best performance.

Next, we examine the buffer hit ratio by varying the buffer capacity (percentage of database size). Table 3 depicts the hit ratio values for each case. We observe that in all cases, as the buffer size increases, the hit ratio converges to 100%, as expected. For IND and real-life data sets, the starting hit ratio is very high (almost 100%) for all algorithms, except

Author's personal copy

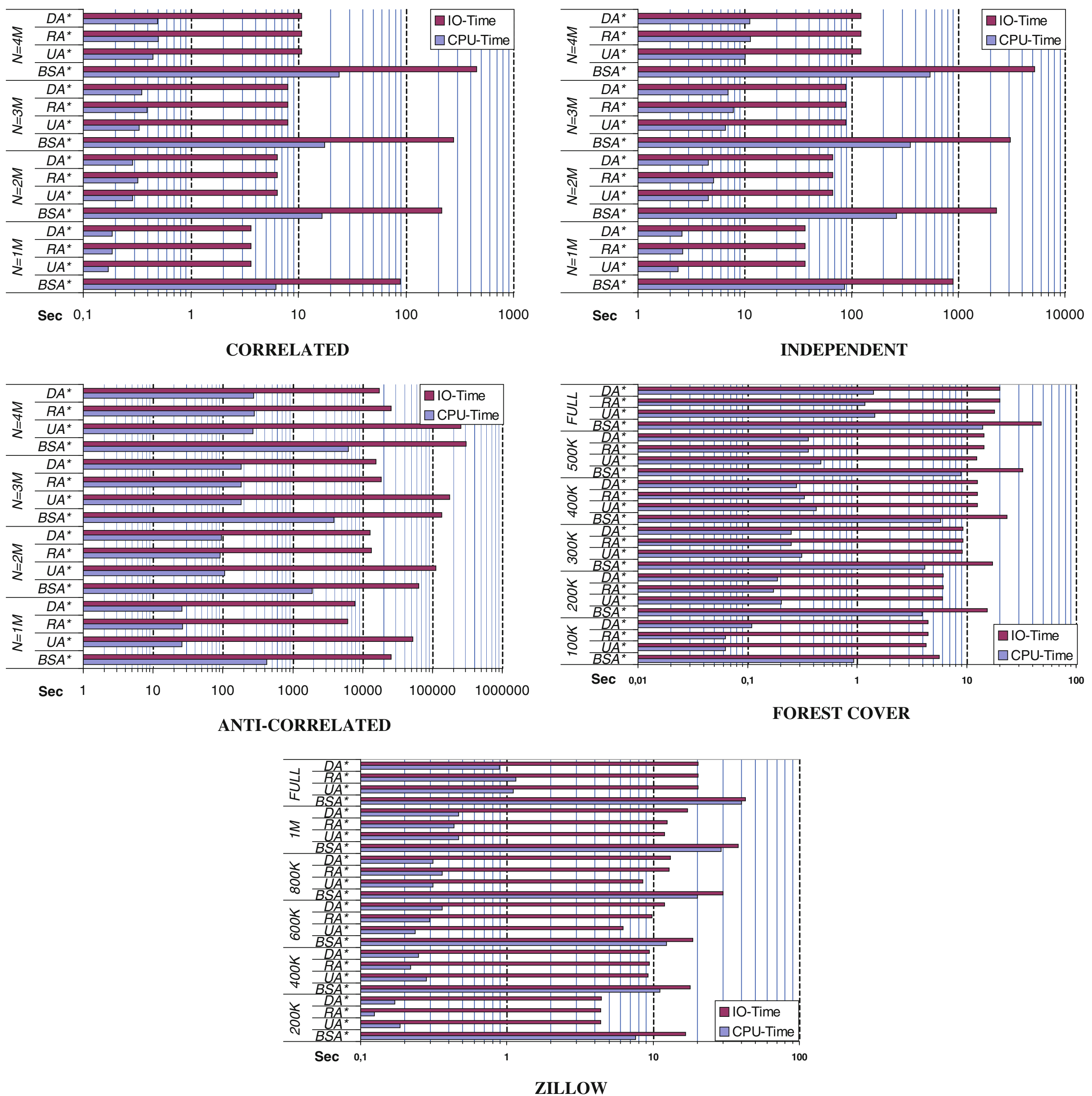

Fig. 15 Response time vs. data set cardinality ($k = 20$, $m = 4$)

BSA* where a buffer size of 20% is required to reach this hit ratio value. In ANT data, the starting hit ratio is low for all algorithms as expected. We conclude that RA* and DA* show the best performance regarding locality.

In the sequel, we study the memory requirements, by recording the maximum size that the heap may reach, by varying the dimensionality and the number of results (k). We recorded the values of the maximum heap size for the with and without using the pruning mechanisms. Tables 4 and 5 depict the results. We observe that when pruning is disabled, the heap size increases almost linearly and remains significantly small in relation to the data set cardinality. However, by enabling pruning, the heap size is reduced even further. For example, in the 5-dimensional ANT data set, when pruning is enabled the memory requirements drop by almost two orders of magnitude, as it is shown in Table 4. This behavior is an indication that pruning is very effective, as the heap size remains extremely low. Even in the ANT data set, the

Author's personal copy

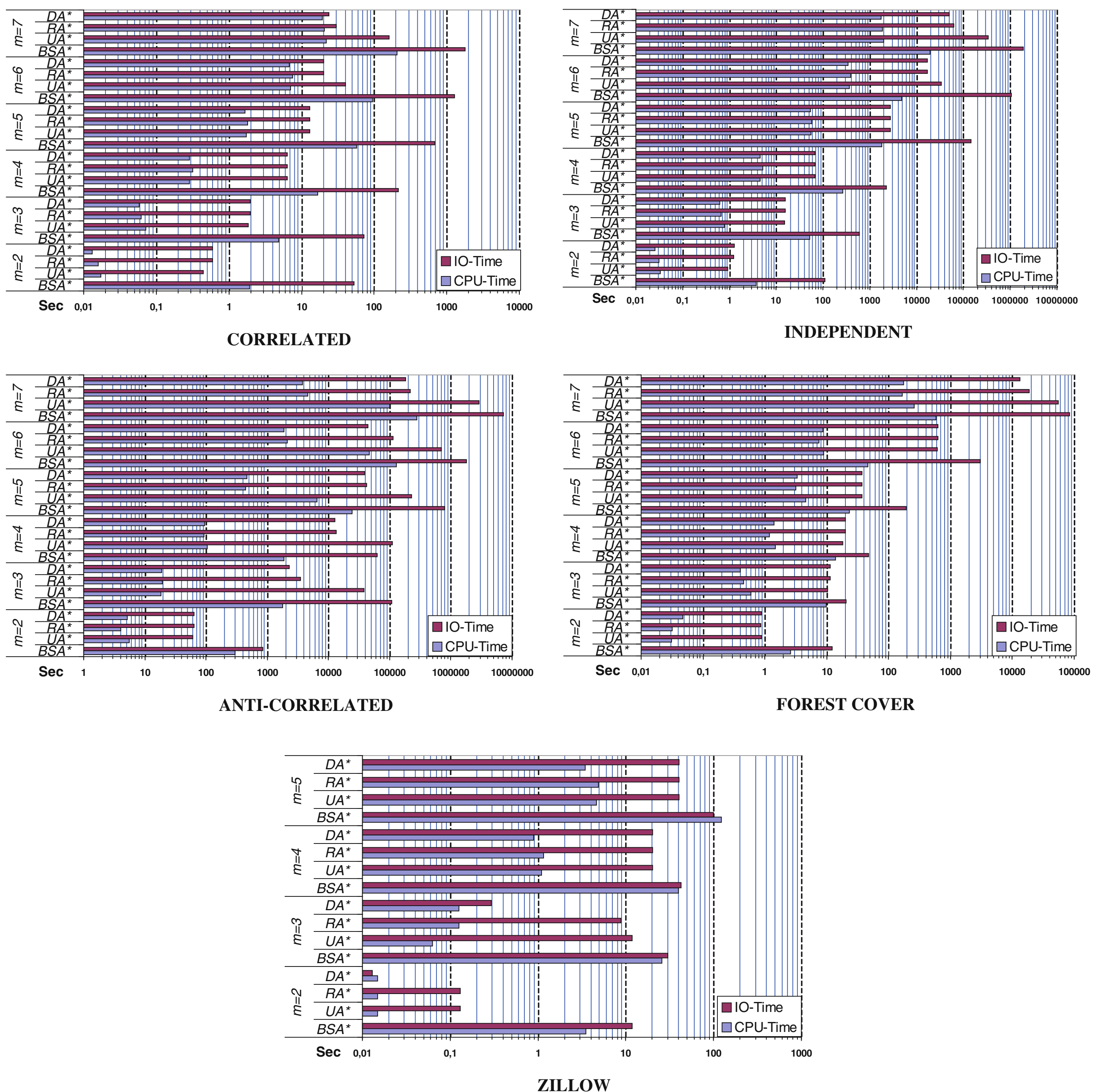

Fig. 16 Response time vs. dimensionality ($k = 20$, $N = 2M$)

heap size remains under 0.015% of the data set size in all parameter variations.

Furthermore, we have studied the effectiveness of each pruning rule by counting each discarded and pruned item. Table 6 depicts the results for the default case when the rules are applied at their most effective priority order. We observe that more than 90% of the items are discarded, meaning that pruning is very effective.

7.2 Comparison with related techniques

In the next series of experiments, we compare the performance of the proposed algorithms to that of CBT and FNP, which according to [30], shows the best performance for indexed and non-index data, respectively. In addition, we perform comparisons with the algorithms proposed in [24] to detect the k nodes with the highest degree in a bipartite graph.

Author's personal copy

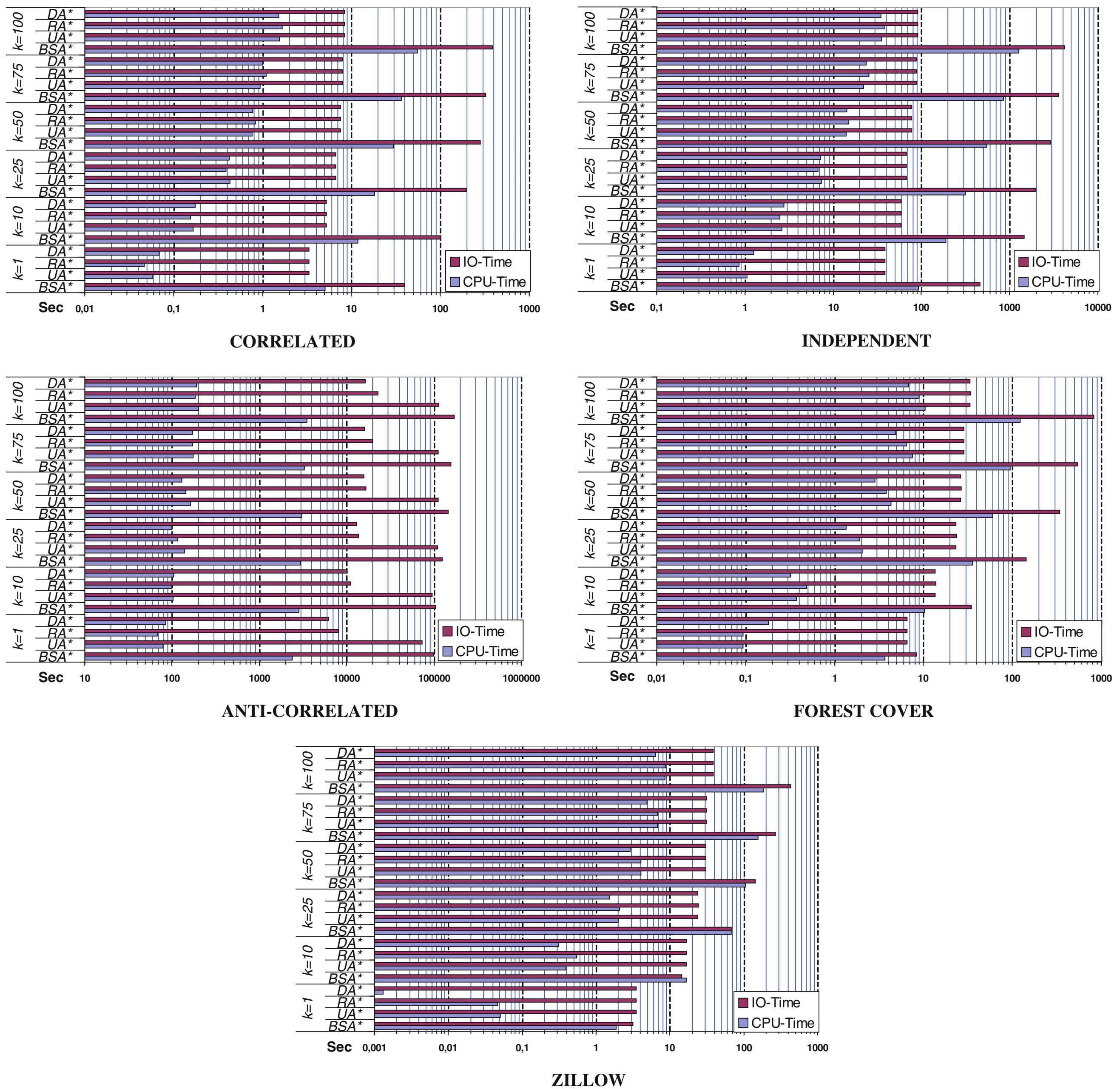

Fig. 17 Response time vs. k ($m = 4$, $N = 2M$)

We note, however, that those algorithms are not progressive and they are not designed to work on vertically decomposed data.

The intuition behind the comparison with CBT is that to process a dominating query in a subset of dimensions, one may build an R-tree index on-the-fly using bulk loading, and then run the CBT algorithm. In our results, and in favor of CBT, we did not include the index building cost, which is expected to be significant, especially for large data sets. The intuition behind the comparison with FNP is that this algorithm can be used to process a dominating query in a subset of dimensions when there is no index available. We note, however, that CBT and FNP are not designed to work on vertically decomposed data. In fact, in cases where dimensions are distributed, these algorithms are not directly applicable, since they require to synthesize multi-dimensional points from separate columns and also require multiple passes over the data.

Tables 7 and 8 depict the results, which show that UA*, RA*, DA* outperform FNP and RA*, DA* outperform CBT in most cases and parameter variations. CBT appears to have

Author's personal copy

Table 3 Performance Hit Ratio Values in Buffer ($k = 20, m = 4, N = 2M$)

	Buffer 10%				Buffer 15%				Buffer 20%				Buffer 25%				Buffer 30%			
	BSA* (%)	UA* (%)	RA* (%)	DA* (%)	BSA* (%)	UA* (%)	RA* (%)	DA* (%)	BSA* (%)	UA* (%)	RA* (%)	DA* (%)	BSA* (%)	UA* (%)	RA* (%)	DA* (%)	BSA* (%)	UA* (%)	RA* (%)	DA* (%)
Correlated	63.6	99.2	99.2	99.2	88.5	99.2	99.2	99.2	97.8	99.2	99.2	99.2	97.8	99.2	99.2	99.2	97.8	99.2	99.2	99.2
Independent	29.7	98.3	98.3	98.0	59.6	98.3	98.3	98.0	89.6	98.3	98.3	98.1	99.8	98.3	98.3	98.0	99.8	98.3	98.3	98.0
Anti-correlated	19.8	3.2	7.9	8.2	29.9	19.1	40.8	43.0	89.8	44.8	91.8	91.5	92.6	70.6	97.6	98.3	94.7	94.4	99.9	99.9
Forest cover	69.9	97.2	96.0	94.8	86.9	97.2	96.3	95.0	96.9	97.2	96.3	95.0	99.8	97.2	96.3	95.0	99.8	97.2	96.3	95.0
Zillow	86.3	96.8	96.8	94.2	94.9	96.8	96.8	94.2	99.1	96.8	96.8	94.2	99.8	96.8	96.8	94.2	99.8	96.8	96.8	94.2

a very high CPU cost for IND and ANT data, due to the existence of many duplicate values. This high CPU cost is due to the fact that in many cases CBT loads a large amount of the data set into memory to perform computations. Table 9 depicts the memory utilization of CBT as a percentage of the maximum $VList$ queue size (which CBT uses) divided by the corresponding data set cardinality N. Each test case corresponds to a different R-tree index. It is evident that the percentage of items maintained in memory is huge, and this is the main reason for the reduced I/O cost of CBT. In summary, DA* is more preferable than CBT because it is more efficient in terms of running time, it is progressive, and requires significantly less memory. Furthermore, since DA* is based on simple one-dimensional techniques and does not require a multi-dimensional index, it can be easily incorporated in existing systems. On the other hand, FNP appears to have a very high CPU cost for larger than 5 dimensions in all data sets, due to the fact that the number of cells in the grid index is very high. Therefore, FNP cannot be applied in high-dimensionality data. Moreover, FNP requires 3 full passes to the data base and a significant amount of memory for the grid structure.

In the next series of experiments, we compare the proposed algorithms to those designed in [24] for the kMCV (k Most Connected Vertex) problem. Two variants are studied as follows: SS (Sample and Sort) and SOE (Switch On Empty), which generally performs better. We applied these algorithms in our framework, by assuming that our data set constitutes a bipartite graph where both bipartitions contain the data items. Recall that SS and SOE perform edge probes to detect the k nodes with the highest degrees. In our setting, an edge probing is equivalent to a domination check between two data items. As previously, the only in-memory maintained information is the disk pages contained in the LRU buffer and the items of the heap data structure. Since both SS and SOE do not perform very well in comparison with other algorithms, we present in Table 10 only the experimental results for the FC data set in. The high I/O cost is due to the large number of domination checks, leading to I/Os without locality.

In the last series of experiments, we compare the performance of all algorithms (except FNP) to very large dimensionality data, in order to test the performance gain of the proposed methods. Therefore, we generated and tested independent data sets of 100 K items up to 35 dimensions. Table 11 depicts the results. The performance gain of RA* and DA* is significant as the dimensionality becomes high.

8 Conclusions

In this paper, we have studied progressive algorithms for top-k dominating queries, where the user expresses an interest in a subset of the available dimensions. Four algorithms

Author's personal copy

Table 4 Maximum heap size vs dimensionality ($k = 20$, $N = 2M$)

	$m = 2$		$m = 3$		$m = 4$		$m = 5$		$m = 6$		$m = 7$	
	No pruning	Pruning	No pruning	Pruning	No pruning	Pruning	No pruning	Pruning	No pruning	Pruning	No pruning	Pruning
Correlated	57	22	72	29	185	29	428	28	860	34	1337	40
Independent	30	21	109	29	298	32	951	37	2,692	45	6,550	39
Anti-correlated	281	37	889	40	1,699	50	4,718	59	13,747	57	22,118	68
Forest cover	16	14	141	28	104	22	599	43	781	43	8,708	124
Zillow	30	22	26	19	92	28	400	34	–	–	–	–

Table 5 Maximum heap size vs k ($m = 4$, $N = 2M$)

	$k = 1$		$k = 10$		$k = 25$		$k = 50$		$k = 75$		$k = 100$	
	No pruning	Pruning	No pruning	Pruning	No pruning	Pruning	No pruning	Pruning	No pruning	Pruning	No pruning	Pruning
Correlated	23	2	106	17	200	31	417	76	529	117	698	150
Independent	39	4	182	13	363	43	726	81	886	120	1,115	155
Anti-correlated	1,365	3	1,715	24	1,849	64	2,110	136	2,225	190	2,290	247
Forest cover	3	2	17	12	208	33	516	77	710	116	779	155
Zillow	4	2	42	13	228	37	483	75	769	116	861	166

Table 6 Effectiveness of pruning rules under the most efficient application order ($k = 20$, $m = 4$)

	N	Discarded	Pruned	DR3	DR1	DR2	EPR4	EPR3	EPR1	EPR5	EPR6	EPR2	IPR1	G
Independent	2,000,000	1,995,128	1,906	1,574,568	326,184	94,376	42	19	405	0	68	707	31	794
Anti-correlated	2,000,000	1,741,533	69,241	1,222,845	385,647	133,041	1	17	10,179	22	1,034	27,134	4,284	31,928
Forest cover	581,012	574,694	698	417,937	125,463	31,294	5	0	297	0	46	244	12	94
Zillow	1,252,208	1,247,640	272	1,099,510	104,513	43,617	2	1	109	0	31	60	8	61

Table 7 Comparison with CBT and FNP for different dimensionalities

		CBT	FNP	BSA*	UA*	RA*	DA*
Independent							
$m = 2$	CPU-Time	0.22	24.17	3.70	0.03	0.03	0.03
	IO-Time	5.13	16.95	109.61	0.94	1.25	1.25
$m = 3$	CPU-Time	48017.67	78.33	52.61	0.78	0.67	0.63
	IO-Time	229.77	31.17	590.70	15.03	15.82	15.82
$m = 4$	CPU-Time	55220.19	247.30	265.44	4.55	5.09	4.55
	IO-Time	285.73	49.71	2280.64	67.15	67.13	67.13
$m = 5$	CPU-Time	64346.16	571.23	1827.45	55.83	58.05	53.21
	IO-Time	344.85	68.85	146154.37	2789.52	2788.70	2788.70
$m = 6$	CPU-Time	72508.69	74883.50	4818.85	361.74	384.75	343.53
	IO-Time	400.02	89.24	1073089.08	33517.73	16755.14	16756.07
$m = 7$	CPU-Time	83792.33	972707.62	19188.46	1996.81	1891.36	1760.70
	IO-Time	465.14	96.14	1880046.12	337348.11	63230.81	50495.37
Anti-correlated							
$m = 2$	CPU-Time	1.95	57.57	296.85	5.56	4.06	5.12
	IO-Time	38.41	25.67	847.14	61.19	63.55	63.55
$m = 3$	CPU-Time	75476.42	172.53	1799.63	18.15	19.36	19.05
	IO-Time	229.91	44.28	108879.31	37794.77	3517.91	2265.95

Author's personal copy

Table 7 continued

		CBT	FNP	BSA*	UA*	RA*	DA*
$m = 4$	CPU-Time	89736.80	1821.40	1881.74	105.41	91.94	94.73
	IO-Time	285.73	62.43	63300.63	112106.27	13282.10	12864.57
$m = 5$	CPU-Time	107804.98	3958.09	24336.43	6490.82	443.92	470.59
	IO-Time	344.85	78.12	782047.06	226057.18	41852.36	39389.67
$m = 6$	CPU-Time	128033.14	60342.83	129299.29	45914.96	2132.56	1867.03
	IO-Time	400.02	92.00	1805634.48	702428.80	113218.05	44129.68
$m = 7$	CPU-Time	151877.30	922972.62	281685.74	100028.30	4645.91	3722.36
	IO-Time	465.14	97.82	7321031.15	2848031.10	218571.78	182288.32
Forest cover							
$m = 2$	CPU-Time	0.11	44.63	2.59	0.03	0.03	0.05
	IO-Time	2.37	9.08	12.30	0.89	0.87	0.89
$m = 3$	CPU-Time	323.42	89.31	9.89	0.60	0.45	0.40
	IO-Time	65.75	13.62	20.35	10.41	11.53	11.49
$m = 4$	CPU-Time	1615.45	141.49	14.04	1.46	1.19	1.42
	IO-Time	82.38	18.16	47.89	18.02	20.24	20.24
$m = 5$	CPU-Time	3048.11	638.95	23.51	4.62	3.17	3.32
	IO-Time	99.44	22.70	190.91	37.24	37.24	37.36
$m = 6$	CPU-Time	5777.94	31641.23	46.30	9.01	7.52	8.79
	IO-Time	115.69	27.23	3052.28	619.92	621.37	621.37
$m = 7$	CPU-Time	13243.91	890138.26	584.18	257.39	168.41	174.52
	IO-Time	135.04	31.83	85166.26	54908.72	18777.98	13288.88
Zillow							
$m = 2$	CPU-Time	28.69	78.47	3.52	0.02	0.02	0.02
	IO-Time	8.58	19.57	11.83	0.13	0.13	0.01
$m = 3$	CPU-Time	28.25	102.51	25.75	0.06	0.13	0.13
	IO-Time	16.78	29.35	30.15	11.72	8.79	0.29
$m = 4$	CPU-Time	2196.16	299.82	40.10	1.10	1.16	0.89
	IO-Time	168.33	39.13	42.87	20.20	20.20	20.20
$m = 5$	CPU-Time	2991.59	678.82	123.29	4.65	4.89	3.46
	IO-Time	200.18	48.91	99.84	40.64	40.71	40.64

Table 8 Comparison with CBT and FNP for different values of k

		CBT	FNP	BSA*	UA*	RA*	DA*
Independent							
$k = 1$	CPU-Time	50635.38	103.11	93.00	1.06	0.86	1.25
	IO-Time	285.11	49.71	461.04	38.62	38.59	38.61
$k = 10$	CPU-Time	53755.19	186.41	189.06	2.64	2.48	2.76
	IO-Time	285.72	49.71	1471.42	58.64	58.61	58.63
$k = 25$	CPU-Time	55267.05	274.56	314.14	7.31	6.69	7.27
	IO-Time	285.73	49.71	1982.14	68.13	68.13	68.13
$k = 50$	CPU-Time	57049.53	381.17	546.57	13.93	15.17	14.27
	IO-Time	285.73	49.71	2912.66	78.26	78.25	78.25
$k = 75$	CPU-Time	58050.55	467.80	848.95	22.09	25.58	23.64
	IO-Time	285.73	49.71	3596.27	87.21	87.19	87.20
$k = 100$	CPU-Time	58830.72	541.05	1262.00	35.74	38.08	35.06
	IO-Time	285.73	49.71	4202.99	91.45	91.43	91.44

Author's personal copy

Table 8 continued

		CBT	FNP	BSA*	UA*	RA*	DA*
Anti-correlated							
$k = 1$	CPU-Time	86800.97	621.31	2381.59	78.41	67.98	84.24
	IO-Time	285.73	62.43	98493.08	73536.06	8040.30	6201.03
$k = 10$	CPU-Time	89118.91	1385.95	2856.08	102.29	98.66	105.53
	IO-Time	285.73	62.43	104356.06	95327.44	10989.66	10104.77
$k = 25$	CPU-Time	90375.66	1929.11	2926.57	140.05	116.88	99.51
	IO-Time	285.73	62.43	123766.65	109223.90	13665.05	13052.05
$k = 50$	CPU-Time	91791.11	2527.59	3041.58	162.94	142.74	128.32
	IO-Time	285.73	62.43	144787.51	111982.33	16727.10	15905.09
$k = 75$	CPU-Time	92879.42	3208.31	3254.37	175.49	171.33	170.87
	IO-Time	285.73	62.43	157195.95	112601.71	19949.22	16166.09
$k = 100$	CPU-Time	93249.93	3741.86	3538.94	200.37	185.30	189.70
	IO-Time	285.73	62.43	172345.70	114244.91	22728.82	16403.22
Forest Cover							
$k = 1$	CPU-Time	45.89	60.63	3.67	0.09	0.09	0.18
	IO-Time	68.51	18.16	8.31	6.52	6.52	6.52
$k = 10$	CPU-Time	710.27	98.09	10.24	0.38	0.48	0.32
	IO-Time	82.25	18.16	34.50	13.64	13.80	13.67
$k = 25$	CPU-Time	2125.73	160.89	36.08	2.03	1.92	1.36
	IO-Time	82.41	18.16	142.68	23.22	23.46	23.22
$k = 50$	CPU-Time	3929.36	271.53	60.16	4.30	3.81	2.84
	IO-Time	82.56	18.16	339.68	26.41	26.54	26.41
$k = 75$	CPU-Time	5077.47	380.64	94.13	7.59	6.44	4.91
	IO-Time	82.65	18.16	543.59	28.38	28.56	28.38
$k = 100$	CPU-Time	6052.66	481.56	122.95	10.53	8.97	6.82
	IO-Time	82.66	18.16	832.90	33.58	33.89	33.58
Zillow							
$k = 1$	CPU-Time	72.52	130.88	1.86	0.05	0.05	0.00
	IO-Time	77.15	39.13	3.13	3.50	3.50	3.50
$k = 10$	CPU-Time	1494.19	220.98	16.70	0.39	0.55	0.31
	IO-Time	164.54	39.13	14.50	16.61	16.66	16.61
$k = 25$	CPU-Time	2474.94	336.74	68.43	2.00	2.06	1.53
	IO-Time	169.06	39.13	67.38	24.10	24.21	24.10
$k = 50$	CPU-Time	3498.02	542.84	104.92	4.09	4.05	2.92
	IO-Time	171.64	39.13	144.15	30.58	30.58	30.58
$k = 75$	CPU-Time	4573.97	702.35	156.01	6.77	6.83	4.93
	IO-Time	172.59	39.13	268.62	31.21	31.27	31.20
$k = 100$	CPU-Time	5483.95	843.49	183.32	8.52	8.78	6.43
	IO-Time	173.17	39.13	433.25	38.74	38.76	38.73

have been studied, namely the Basic-Scan Algorithm (BSA) which is used as the baseline, the Union Algorithm (UA), the Reverse Algorithm (RA), and the Differential Algorithm (DA). A performance evaluation has been conducted using real life as well as synthetic data sets, in order to test their behavior in terms of computational cost and memory consumption. Moreover, an analytical study has been performed providing estimates for several important quantities and reporting on the worst-case complexity of the algorithms. In addition, a set of pruning rules has been studied that can be applied toward performance boost.

All algorithms share the same characteristics regarding the way sorted accesses are enforced and operate on any subset of the dimensions. This is facilitated by providing a

Author's personal copy

Table 9 Memory utilization for CBT ($|VList|/N$)

	Independent (%)	Anti-correlated (%)	Forest cover (%)	Zillow (%)
$m = 2$	0.25110	0.86695	0.72511	6.48862
$m = 3$	95.33445	95.03355	25.82150	7.15512
$m = 4$	98.56270	95.88820	39.15496	25.54520
$m = 5$	98.96135	96.42650	49.28831	28.97394
$m = 6$	99.08865	96.99015	51.31529	–
$m = 7$	99.11505	97.04245	51.56124	–

Table 10 Comparison with SS-SOE for different dimensionalities and values of k (number of results)

		SS	SOE	BSA*	UA*	RA*	DA*
Forest cover							
$m = 2$	CPU-Time	913.94	946.73	2.59	0.03	0.03	0.05
	IO-Time	146869.54	133946.66	12.30	0.89	0.87	0.89
$m = 3$	CPU-Time	5347.95	7600.45	9.89	0.60	0.45	0.40
	IO-Time	870715.09	784550.01	20.35	10.41	11.53	11.49
$m = 4$	CPU-Time	8832.30	9125.91	14.04	1.46	1.19	1.42
	IO-Time	1131930.07	991885.59	47.89	18.02	20.24	20.24
$m = 5$	CPU-Time	15680.49	17006.03	23.51	4.62	3.17	3.32
	IO-Time	2419645.37	2016833.65	190.91	37.24	37.24	37.36
$m = 6$	CPU-Time	26864.92	28482.77	46.30	9.01	7.52	8.79
	IO-Time	3546191.08	2900638.84	3052.28	619.92	621.37	621.37
$m = 7$	CPU-Time	61480.19	65399.23	584.18	257.39	168.41	174.52
	IO-Time	8343022.56	6863947.57	85166.26	54908.72	18777.98	13288.88
$k = 1$	CPU-Time	2294.50	2381.27	3.67	0.09	0.09	0.18
	IO-Time	254684.67	236218.36	8.31	6.52	6.52	6.52
$k = 10$	CPU-Time	6894.77	6385.67	10.24	0.38	0.48	0.32
	IO-Time	789603.84	733114.40	34.50	13.64	13.80	13.67
$k = 25$	CPU-Time	9712.92	10331.67	36.08	2.03	1.92	1.36
	IO-Time	1149643.28	1067249.75	142.68	23.22	23.46	23.22
$k = 50$	CPU-Time	14935.45	15978.88	60.16	4.30	3.81	2.84
	IO-Time	1655342.76	1536694.27	339.68	26.41	26.54	26.41
$k = 75$	CPU-Time	16825.16	17972.83	94.13	7.59	6.44	4.91
	IO-Time	1910461.19	1773626.22	543.59	28.38	28.56	28.38
$k = 100$	CPU-Time	18232.91	19460.52	122.95	10.53	8.97	6.82
	IO-Time	2084172.86	1934493.77	832.90	33.58	33.89	33.58

vertical decomposition of the data set, thus organizing each dimension separately. This alternative has been proven very efficient and in several cases is the only solution (e.g., distributed databases).

The basic difference among the studied algorithms is in the way domination scores are computed. BSA* requires a significant number of random accesses, leading to increased processing costs. Its performance is inferior to that of the other algorithms. The performance of UA* deteriorates significantly when the data set distribution tends to be anti-correlated. To eliminate the problems of UA*, RA* and DA* have been designed. RA* scans backward as well as forward, reducing the CPU and the I/O cost. DA* uses an even more sophisticated mechanism to compute domination scores, which is based on the best terminating item detected so far.

Based on the results, DA* shows the best overall performance on all experiments conducted, with only a few exceptions (e.g., in the FC data set, RA* is marginally better than DA* for small values of k and m). In addition, DA* is superior to: (1) CBT, which requires an aggregate R-tree, (2) FNP, which builds a grid index on-the-fly, and (3) SS and SOE

Author's personal copy

Table 11 Comparison of all algorithms in large dimensionalities

		CBT	SS	SOE	BSA*	UA*	RA*	DA*
$m = 5$	CPU-Time	178.19	589.72	658.27	5.34	0.61	0.78	0.98
	IO-Time	51.77	404703.49	380233.39	28183.89	7.79	7.80	7.80
$m = 10$	CPU-Time	264.88	2147.58	2026.91	97.13	31.43	32.51	30.92
	IO-Time	96.89	1022706.23	972121.44	489550.10	227.83	131.81	119.35
$m = 15$	CPU-Time	344.44	2910.52	2961.41	122.77	139.64	102.62	97.15
	IO-Time	150.05	1199538.95	1131861.28	598334.95	610.08	371.56	365.33
$m = 20$	CPU-Time	1034.08	3402.05	3416.02	127.67	879.15	184.51	178.99
	IO-Time	200.12	1297931.50	1232532.57	490455.95	6171.13	786.83	671.08
$m = 25$	CPU-Time	2953.89	3844.30	3772.64	119.78	3145.15	367.85	321.49
	IO-Time	250.10	1385269.83	1318986.83	396145.15	15637.81	1318.36	1075.99
$m = 30$	CPU-Time	9089.09	4306.14	4291.80	76.16	3718.56	495.17	425.16
	IO-Time	300.08	1474294.80	1406000.00	332308.72	20403.37	1882.19	1511.52
$m = 35$	CPU-Time	22501.09	4795.69	4876.89	66.25	3934.62	525.61	485.16
	IO-Time	375.14	1563312.60	1490000.00	282728.92	22251.91	2162.81	1842.78

algorithms proposed to solve the k most connected vertex problem. The performance difference is significant in terms of running time and in-core storage requirements.

An interesting research direction is the development of approximate algorithms with probabilistic guarantees. In such a case, we are willing to sacrifice accuracy for more efficient processing. Techniques like the ones studied in [12], which utilize vector approximations, will help significantly toward this goal.

Acknowledgments We would like to express our gratitude to Y. Tao, C. Sheng, J. Li, M.L. Yiu and N. Mamoulis for sharing their source code with us.

References

1. Abadi, D.J., Madden, S.R., Hachem, N.: Column-Stores vs. Row-Stores: How Different Are They Really? In: Proceedings of the ACM SIGMOD Conference, pp. 967–980 (2008)
2. Balke, W.T., Guentzer, U., Zheng, J.X.: Efficient distributed skylining for web information systems. Proceedings of Advances in Database Technology (EDBT), pp. 256–273 (2004)
3. Boncz, P.A., Kersten, M.L., Manegold, S.: Breaking the Memory Wall in MonetDB. Communications of the ACM **51**(12), 77–85 (2008)
4. Borzsonyi, S., Kossmann, D., Stocker, K.: The Skyline Operator. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 421–430 (2001)
5. Chan, C.Y., Jagadish, H.V., Tan, K.-L., Tung, A.K.H., Zhang, Z.: On high dimensional skylines. In: Proceedings of the 10th International Conference on Extending Database Technology (EDBT), pp. 478–495 (2006)
6. Chan, C.Y., Jagadish, H.V., Tan, K.-L., Tung, A.K.H., Zhang, Z.: Finding k-dominant Skylines in High Dimensional Space. In: Proceedings of the ACM SIGMOD Conference, pp. 503–514 (2006)
7. de Vries, A.P., Mamoulis, N., Nes, N., Kersten, M.: Efficient k-NN Search on Vertically Decomposed Data. In: Proceedings of the ACM SIGMOD Conference, pp. 322–333 (2002)
8. Dvoretzky, A., Kiefer, J., Wolfowitz, J.: Asymptotic minimax character of the sample distribution function and of the classical multinomial estimator. Ann. Math. Stat. **27**(3), 642–669 (1956)
9. Fagin, R.: Combining fuzzy information from multiple systems. In: Proceedings of the Symposium on Principles of Database Systems (PODS), pp. 216–226 (1996)
10. Fagin, R.: Optimal aggregation algorithms for middleware. In: Proceedings of ACM PODS Conference (PODS), pp. 102–113 (2001)
11. Gilchrist, W.: Statistical Modelling with Quantile Functions. Chapman and Hall/CRC, London (2000)
12. Kriegel, H.-P., Kroeger, P., Schubert, M., Zhu, Z.: Efficient query processing in arbitrary subspaces using vector approximations. In: Proceedings of the 18th International Conference on Scientific and Statistical Database Management (SSDBM), pp. 184–190 (2006)
13. Lian, X., Chen, L.: Top-*k* dominating queries in uncertain databases. In: Proceedings of the 12th International Conference on Extending Database Technology (EDBT), pp. 660–671 (2009)
14. Lin, X., Yuan, Y., Zhang, Q., Zhang, Y.: Selecting Stars: The k most representative skyline operator. In: Proceedings of the 23rd IEEE international conference on data engineering (ICDE), pp. 86–95 (2007)
15. Lo, E., Yip, K.Y., Lin, K., Cheung, D.W.: Progressive skylining over web-accessible databases. Data and Knowledge Engineering **57**(2), 122–147 (2006)
16. Marian, A., Bruno, N., Gravano, L.: Evaluating top-*k* queries over web-accessible databases. ACM Transactions on Database Systems **29**(2), 319–362 (2004)
17. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive skyline computation in database systems. ACM Transactions on Database Systems **30**(1), 41–82 (2005)
18. Skoutas, D., Sacharidis, D., Simitsis, A., Kantere, V., Sellis, T.: Top-*k* dominant web services under multi-criteria matching. In: Proceedings of the 12th International Conference on Extending Database Technology (EDBT), pp. 898-909 (2009)
19. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., et al.: C-Store: a column oriented DBMS. In: Proceedings of Very Large Data Bases Conference (VLDB), pp. 553–564 (2005)

Author's personal copy

20. Tao, Y., Xiao, X., Pei, J.: SUBSKY: efficient computation of skylines in subspaces. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 65–76 (2006)
21. Tao, Y., Xiao, X.: Efficient skyline and top-*k* retrieval in subspaces. IEEE Transactions on Knowledge and Data Engineering **19**(8), 1072–1088 (2007)
22. Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient progressive skyline computation. In: Proceedings of Very Large Data Bases Conference (VLDB), pp. 301–310 (2001)
23. Tao, Y., Ding, L., Lin, X., Pei, J.: Distance-based representative skyline. In: Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE), pp. 892–903 (2009)
24. Tao, Y., Sheng, C., Li, J.: Finding maximum degrees in hidden bipartite graphs. In: Proceedings of ACM SIGMOD Conference, pp. 891–902 (2010)
25. Tong, Y.L.: Probability Inequalities in Multivariate Distributions. Academic Press, London (1980)
26. Vlachou, A., Doulkeridis, C., Kotidis, Y.: Angle-based space partitioning for efficient parallel skyline computation. In: Proceedings of the ACM SIGMOD Conference, pp. 227–238 (2008)
27. Xia, T., Zhang, D.: Refreshing the sky: the compressed skycube with efficient support for frequent updates. In: Proceedings of the ACM SIGMOD Conference, pp. 491–502 (2006)
28. Xia, T., Zhang, D., Tao, Y.: On skylining with flexible dominance relation. In: Proceedings of the 24th IEEE International Conference on Data Engineering (ICDE), pp. 1397–1399 (2008)
29. Yiu, M.L., Mamoulis, N.: Efficient processing of top-*k* dominating queries on multi-dimensional data. In: Proceedings of Very Large Data Bases Conference (VLDB), pp. 483–494 (2007)
30. Yiu, M.L., Mamoulis, N.: Multi-dimensional top-*k* dominating queries. The VLDB J. **18**(3), 695–718 (2009)
31. Yuan, Y., Lin, X., Liu, Q., Wang, W. et al.: Efficient computation of the skyline cube. In: Proceedings of Very Large Data Bases Conference (VLDB), pp. 241–252 (2005)
32. Zhang, Z., Guo, X., Lu, H., Tung, A.K., Wang, N.: Discovering strong skyline points in high-dimensional spaces. In: Proceedings of the 14th ACM Conference on Information and Knowledge Management (CIKM), pp. 247–248 (2005)
33. Zhang, W., Lin, X., Zhang, Y., Pei, J., Wang, W.: Threshold-based probabilistic top-*k* dominating queries. The VLDB Journal **19**(2), 283–305 (2009)
34. Zhang, Z., Lu, H., Ooi, B.C., Tung, A.: Understanding the meaning of a shifted sky: a general framework on extending skyline query. The VLDB Journal **19**(2), 181–201 (2010)