3B2v8.06a/w (Dec 5 2003).51c + model | IS : 488 | Prod.Type:FTP pp.1–22(col.fig.:10–15,18,19) | ED:PreethiNS PAGN:NMN SCAN:puspha

ARTICLE IN PRESS

Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Information Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮

www.elsevier.com/locate/infosys

Branch-and-bound processing of ranked queries

Yufei Tao[a], Vagelis Hristidis[b], Dimitris Papadias[c,*], Yannis Papakonstantinou[d]

[a]*Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong*
[b]*School of Computing and Information Sciences, Florida International University, Miami, FL 33199, USA*
[c]*Department of Computer Science, Hong Kong University of Science and Technology, Clearwater Bay, Hong Kong*
[d]*Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA, USA*

Received 22 April 2003; received in revised form 15 December 2005; accepted 21 December 2005

Recommended by Y. Ioannidis

Abstract

Despite the importance of ranked queries in numerous applications involving multi-criteria decision making, they are not efficiently supported by traditional database systems. In this paper, we propose a simple yet powerful technique for processing such queries based on multi-dimensional access methods and branch-and-bound search. The advantages of the proposed methodology are: (i) it is space efficient, requiring only a single index on the given relation (storing each tuple at most once), (ii) it achieves significant (i.e., orders of magnitude) performance gains with respect to the current state-of-the-art, (iii) it can efficiently handle data updates, and (iv) it is applicable to other important variations of ranked search (including the support for non-monotone preference functions), at no extra space overhead. We confirm the superiority of the proposed methods with a detailed experimental study.

Keywords: Databases; Ranked queries; R-tree; Branch-and-bound algorithms

1. Introduction

The ability to efficiently rank the "importance" of data, is crucial to many applications that involve multi-criteria decision making. Consider a database of mutual funds, where each fund has two attributes (i) "growth", indicating the recent increase of its asset, and (ii) "stability", representing the overall volatility of its growth (low stability indicates high volatility). Fig. 1 shows the attribute values (normalized to [0,1]) of 12 funds. Customers select the "best" funds for investment based on, however, different preferences. For example, an investor whose primary goal is capital conservation with minimum risk would prefer funds with high stability, while another client may prioritize both attributes equally. To express these requests in a uniform manner, the ranking system adopts a *preference function* $f(t)$ which computes a *score* for every record t, and rates the relative importance of various records by their scores. Consider, for example, the linear preference function $f(t) = w_1 \cdot t.growth + w_2 \cdot t.stability$ for Fig. 1, where w_1 and w_2 are specified by a user to indicate her/his priorities

*Corresponding author. Tel.: +852 23586971; fax: +852 23581477.

E-mail addresses: taoyf@cs.cityu.edu.hk (Y. Tao), vagelis@cis.fiu.edu (V. Hristidis), dimitris@cs.ust.hk (D. Papadias), yannis@cs.ucsd.edu (Y. Papakonstantinou).

URL: http://www.cs.ust.hk/~dimitris/.

0306-4379/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2005.12.001

fund id	1	2	3	4	5	6	7	8	9	10	11	12
growth	0.2	0.1	0.3	0.2	0.3	0.5	0.4	0.6	0.7	0.6	0.7	0.7
stability	0.2	0.5	0.3	0.9	0.8	0.7	0.3	0.1	0.2	0.5	0.6	0.5

Fig. 1. An example dataset.

on the two attributes. For $w_1 = 0.1$, $w_2 = 0.9$ (stability is favored), the best three funds have ids 4, 5, 6 since their scores (0.83, 0.75, 0.68, respectively) are the highest. Similarly, if $w_1 = 0.5$, $w_2 = 0.5$ (both attributes are equally important), the ids of the best three funds become 11, 6, 12.

The above operation, known as *top-k ranked search*, cannot be efficiently supported by conventional databases, and has received considerable attention in recent years. Informally, a top-k query specifies a preference function $f(t)$, and returns the k tuples with the highest scores (a formal definition appears in Section 3). In particular, the preference function is *not* known in advance (otherwise, the problem is trivially solved by simply sorting the dataset according to the given function), and different queries may adopt distinct functions. In practice, a "good" ranking system should (i) answer any query with low cost, (ii) incur minimum space overhead, (iii) support database updates, and (iv) efficiently process variations of ranked searches (e.g., different types of preference functions, etc.).

All the existing methods [1–4] (reviewed in the next section) satisfy only part of the above requirements, and hence are inadequate for practical applications. Particularly, they require pre-computing and materializing significant amount of information, whose size can be several times larger than the original database. As a result, considerable re-computation is needed (to modify the materialized data) for each update. Furthermore, these methods focus exclusively on traditional *top-k* queries, and cannot be efficiently adapted to other variations of ranked search.

Motivated by these shortcomings, we provide a methodology for ranked retrieval that indeed satisfies all the "practical" requirements mentioned earlier, and has significantly wider applicability than the previous methods. Specifically, our technique uses only a *single* multi-dimensional index (e.g., R-trees [5,6]) that stores each tuple at most once, to answer all types of top-k queries (for all k, preference functions, and variations). Further, the index required is currently available in existing DBMS (Oracle, Informix, etc.), and hence, the proposed algorithms can be implemented with minimum effort. Specifically, our contributions are:

- We reveal the close relationship between ranked search and the well-studied *branch-and-bound* processing framework. In particular, this framework significantly reduces the difficulty of the problem, and leads to novel solutions that are much simpler, but more powerful, than the previous ones.
- We develop a new algorithm, BRS, which pipelines continuously the data records in descending order of their scores. We provide a detailed performance analysis of BRS, including a technique to estimate its query cost (in terms of the number of disk accesses).
- We discuss several important variations of ranked retrieval, including (i) the *constrained top-k query*, which returns the k tuples with the highest scores among the records satisfying some selection conditions, (ii) the *group-by ranked search*, which retrieves the top-k objects for each group produced by a group-by operation, and (iii) the support of "non-monotone" preference functions (to be elaborated in Section 3).
- We evaluate BRS using extensive experiments, and show that it outperforms the existing methods significantly on all aspects (including the query cost, space overhead, applicability to alternative forms of ranked search, etc.).

The rest of the paper is organized as follows. Section 2 surveys the previous work on top-k search and other related queries. Section 3 formally defines the problem, and motivates its connection with the branch-and-bound paradigm. Section 4 presents BRS, analyzes its performance and describes a method for reducing the space requirements. Section 5 extends our methodology to other variations of top-k retrieval. Section 6 contains an extensive experimental evaluation, and Section 7 concludes the paper with directions for the future work.

2. Related work

Section 2.1 surveys methods for processing ranked queries, focusing primarily on the direct competitors of our technique. Then, Section 2.2 introduces branch-and-bound algorithms on R-trees that motivate our work.

2.1. Ranked queries

To the best of our knowledge, *Onion* [1] is the first ranked search algorithm in databases. Specifically, given a relational table T with d attributes A_1, $A_2, \ldots, A_d$, *Onion* is optimized for linear preference functions in the form $f(t) = \sum_{i=1-d}(w_i \cdot t.A_i)$, where $w_1, w_2, \ldots, w_d$ are d constants specified by the user. Each tuple is converted to a d-dimensional point, whose ith ($1 \leqslant i \leqslant d$) coordinate equals $t.A_i$. The motivation of *Onion* is that the result of a top-1 query must lie in the convex hull CX_1 of the (transformed) points. Let CX_2 be the convex hull of the points in $T - CX_1$ (i.e., points that do not belong to CX_1). Then, objects satisfying a top-2 query can always be found in the union of CX_1 and CX_2. In general, if CX_i is the "layer-i" convex hull, a top-k query can be answered using only $CX_1, \ldots, CX_k$. Based on these observations, *Onion* pre-computes the convex hulls of all layers, and materializes them separately on the disk. Given a query, it first scans the most-exterior hull, progresses to the inner layers incrementally, and stops when it detects that the remaining hulls cannot contain any other result.

Onion is not applicable to non-linear preference functions (in which case the top-1 object, for example, does not necessarily lie in CX_1). Even for linear functions, *Onion* incurs expensive pre-processing and query costs. Specifically, it is well-known that the cost of computing the convex hull is $O(n^{d/2})$ for n points in the d-dimensional space. Thus, *Onion* is impractical for large relations with more than three attributes, especially when updates are allowed (as they trigger the re-computation of the convex hulls). To answer a query, *Onion* needs to access at least one full hull $CX_1(T)$, whose size may be large in practice. In the worst case, when all the points belong to $CX_1(T)$, the whole database must be scanned.

Currently, the most efficient method for ranked queries is *Prefer* [2,4]. Assume that all the records have been sorted in descending order of their scores according to an *arbitrary* preference function f_V. The sorted list is materialized as a view V (which has the same size as the dataset). Consider a top-k query q with preference function f_q. Obviously, if $f_V = f_q$, its result consists of exactly the first k tuples in the sorted list V, in which case the query cost is minimal (i.e., the cost of sequentially scanning k tuples). The crucial observation behind *Prefer* is that, even in the general case where $f_V \neq f_q$, we can still use V to answer q *without* scanning the entire view. Specifically, the algorithm examines records of V in their sorted order, and stops as soon as the *watermark* record is encountered. A watermark is the record such that, tuples ranked after it (in V) cannot belong to the top-k of f_q, and hence do not need to be visited (see [4] for the watermark computation). Evidently, the number k_V of records that need to be accessed (before reaching the watermark) depends on the similarity between f_V and f_q. Intuitively, the more different f_V is from f_q, the higher k_V is. When f_V and f_q are sufficiently different, using V to answer q needs to visit a prohibitive number of records, even for a small k. To overcome this problem, *Prefer* materializes multiple views (let the number be m) $V_1, V_2, \ldots, V_m$, which sort the dataset according to *different* preference functions $f_{V_1}, f_{V_2}, \ldots, f_{V_m}$. Given a query q, *Prefer* answers it using the view whose preference function is most similar to f_q. Hristidis and Papakonstantinou [4] propose an algorithm that, given m, decides the optimal $f_{V_1}, f_{V_2}, \ldots, f_{V_m}$ to minimize the expected query cost.

Prefer is applicable to many "monotone" preference functions (discussed in Section 3), but requires that the function "type" (e.g., linear, logarithmic, etc.) should be known in advance. Views computed for one type of functions *cannot* be used to answer queries with other types of preferences. Therefore, to support h preference types, totally $h \cdot m$ views need to be stored, requiring space as large as $h \cdot m$ times the database size. Further, for a particular type, satisfactory query performance is possible only with a large number m of views. As an example, the experiments of [4] show that, to achieve good performance for top-10 queries on a relation with four attributes, $m = 40$ views must be created! Since each tuple is duplicated in every view, when it is updated, all its copies must be modified accordingly, resulting in high overhead. Hence, this technique is advocated only if the data are static, and the system has huge amount of available space.

Numerous papers (see [7–10] and the references therein) have been published on top-k search when

the information about each object is distributed across *multiple* sources. As an example, assume a user wants to find the k images that are most similar to a query image, defining similarity according to various features such as color, texture, pattern, etc. The query is submitted to several retrieval engines, each of which returns the most similar images based on a *subset* of the features, together with their similarity scores (e.g., the first engine will output images with the best matching color and texture, the second engine according to pattern, and so on). The problem is to combine the multiple outputs to determine the top-k images in terms of the *overall* similarity, by reading as few results from each source as possible. In this paper, we consider all the data reside on a single local repository, as with *Onion* and *Prefer*. Nevertheless, our technique is complementary to the distributed top-k retrieval since it can be deployed to efficiently find the (partial) results at each source.

Finally, Tsaparas et al. [3] propose a join index to efficiently rank the results of joining multiple tables. The key idea is to pre-compute the top-K results for *every* possible preference function, where K is a *given* upper bound on the number of records returned. Although this technique can be adapted to ranked search on a single table (as is our focus), its applicability is seriously limited since: (i) no top-k query with $k>K$ can be supported, and (ii) it applies to tables with *only* two (but not more) attributes. Furthermore, even in its restricted scope (i.e., $k<K$ and two dimensions), this method suffers from similar pre-computation problems as *Prefer*, or specifically, large space consumption and poor update overhead. In this paper, we discuss general top-k techniques applicable to arbitrary k and dimensionalities, and thus exclude [3] from further consideration.

2.2. Branch-and-bound search on R-trees

The R-tree [5,6] is a popular access method for multi-dimensional objects. Fig. 2a shows part of a 2D point dataset, and Fig. 2b illustrates the corresponding R-tree, where each node can contain at most three entries. Each leaf entry stores a point, and nearby points (e.g., a, b, c) are grouped into the same leaf node (N_4). Each node is represented as a *minimum bounding rectangle* (MBR), which is the smallest axis-parallel rectangle that encloses all the points in its sub-tree. MBRs at the same level are recursively clustered (by their proximity) into nodes at the higher level (e.g., in Fig. 2b, N_4, N_5, N_6 are grouped into N_7), until the number of clusters is smaller than the node capacity. Each non-leaf entry stores the MBR of its child node, together with a (child) pointer.

The branch-and-bound framework has been applied extensively to develop efficient search algorithms based on R-tree for numerous problems, including NN search [11,12], convex hull computation [13], skyline retrieval [14,15], moving object processing [16], etc. In the sequel, we introduce the framework in the context of NN retrieval, which is most related to ranked search as elaborated in the next section.

Specifically, a k NN query retrieves the k points closest to a query point. The *best-first* (BF) [12] algorithm utilizes the concept of *mindist* defined as follows. The *mindist* of an intermediate entry equals

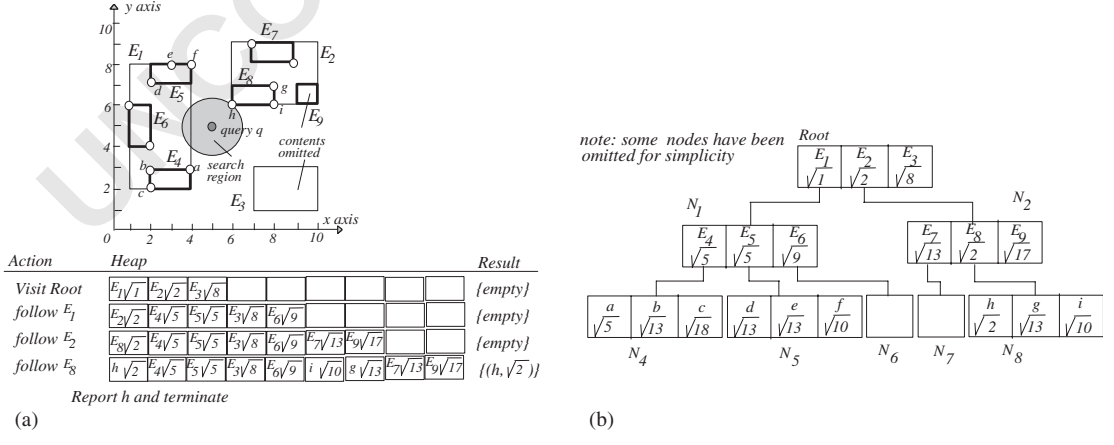

Fig. 2. Nearest neighbor (NN) processing using R-trees. (a) The dataset, node MBRs, and heap content in NN search, (b) the R-tree.

the minimum distance between its MBR and the query point q, while for a leaf entry, *mindist* equals the distance between the corresponding data point and q. Fig. 2b shows the *mindist* values of all the entries in the tree (these numbers are for illustrative purposes only, and are *not* actually stored) with respect to the query in Fig. 2a ($k = 1$). BF keeps a *heap* H that contains the entries of the nodes visited so far, sorted in ascending order of their *mindist*. Initially, H contains the root entries, and BF repeatedly de-heaps and processes the heap entry with the smallest *mindist*. Processing an intermediate entry involves fetching its child node, and inserting all its entries into H. In Fig. 2, since E_1 has the smallest *mindist* (among the root entries), BF removes it from H, visits its child node N_1, and en-heaps entries E_4, E_5, E_6 together with their *mindist*. The next entry processed is E_2 (it has the minimum *mindist* in H now), followed by E_8. The next entry de-heaped is data point h, which is guaranteed to be the first NN of q [12], and hence the algorithm terminates. Fig. 2a demonstrates the changes of the heap contents in the above process. In general, for a k NN query, the algorithm continues until k data points have been removed from H.

BF is *optimal* in the sense that it only visits the nodes "necessary" for discovering k NN. As discussed in [12,17] the "necessary" nodes include those whose MBRs intersect the *search region*, which is a circle centering at the query point q, with radius equal to the distance between q and its kth NN (Fig. 2a shows the region for $k = 1$). The performance of BF has been extensively studied, and several cost models [17–19] are proposed to predict the number of R-tree nodes accesses in processing a query.

3. Problem definition

Let T be a relational table with d numerical attributes A_1, A_2,…,A_d. For each tuple $t \in T$, denote $t.A_i$ as its value on attribute A_i. Without loss of generality [1,4], we consider the permissible values of each attribute distribute in the unit range [0,1] (i.e., $t.A_i \in [0, 1]$). We convert each tuple t to a d-dimensional point whose ith ($1 \leqslant i \leqslant d$) coordinate equals $t.A_i$, and index the resulting points with an R-tree. Fig. 3 shows the transformed points for the fund records of Fig. 1 (e.g., t_1 corresponds to the tuple with id 1, t_2 for id 2, etc.), as well as the corresponding R-tree. In the sequel, we will use this dataset as the running example to illustrate the proposed algorithms. Although our discussion involves 2D objects, the extension to higher dimensionality is straightforward.

A *preference function* f takes as parameters the attribute values of a tuple t, and returns the *score* $f(t)$ of this tuple. Given such a function f, a *top-k ranked* query retrieves the k records $t_1, t_2, \ldots, t_k$ from table T with the highest scores. We denote the score of t_i ($1 \leqslant i \leqslant k$) as s_i, and without loss of generality, assume $s_1 \geqslant s_2 \geqslant \cdots \geqslant s_k$. Special care should be taken when multiple objects achieve the kth score s_k (e.g., for the top-1 query with $f(t) = t.A_2$ in Fig. 3, t_{11}, t_{12}, t_9 have the same score $s_1 = 0.7$). In this paper, we assume the query simply returns *any* of them. This assumption is made purely for simplicity: the discussion for other choices (e.g., reporting them all) is fundamentally identical, but involves unnecessary complications.

A function f is *increasingly monotone on the ith dimension* ($1 \leqslant i \leqslant d$), if $f(p_1) < f(p_2)$, for any two points p_1 and p_2 such that $p_1.A_j = p_2.A_j$ on dimensions $j \neq i$, and $p_1.A_i < p_2.A_i$ (i.e., the coordinates of p_1 and p_2 agree on all the axes except the ith

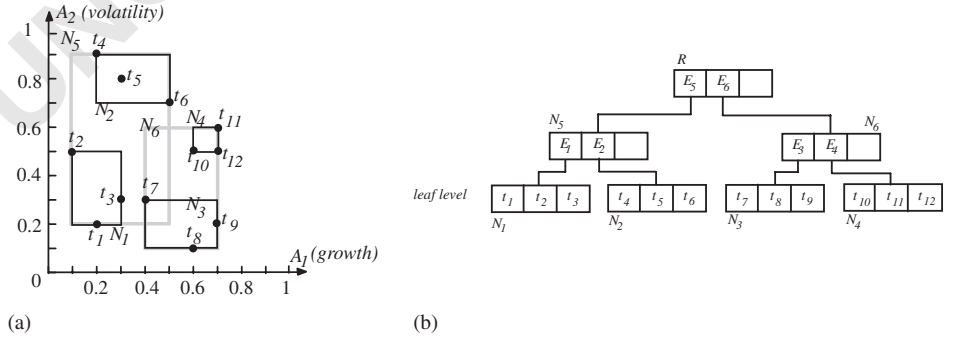

Fig. 3. The multi-dimensional representation of data in Fig. 1.

one). Similarly, f is *decreasingly monotone on the ith dimension* if, given any two points p_1, p_2 as above, $f(p_1) > f(p_2)$ always holds. We say f is *monotone*, if it is (either increasingly or decreasingly) monotone on *all* dimensions; otherwise, f is a *non-monotone*.

A popular monotone preference is the linear function $f(t) = \sum_{i=1-d}(w_i \cdot t.A_i)$. Further, if $w_i > 0$ (<0), then $f(t)$ is increasingly (decreasingly) monotone on the ith axis. For example, $f(t) = t.A_1 - t.A_2$ is increasingly monotone on A_i but decreasingly monotone on A_2 (recall that a monotone function can be increasingly monotone on some attributes, but decreasingly monotone on the others). Some instances of non-monotone functions include non-linear polynomials (e.g., $f(t) = t.A_1^2 - t.A_1 + 2 \cdot t.A_2$), and functions with absolute operators (e.g., $f(t) = |t.A_1 - 0.5| + |t.A_2 - 0.5|$). We point out that, as will be discussed in Section 5.3, *Onion* and *Prefer* do not support non-monotone preferences, which are inherently more difficult to process than the monotone functions.

A concept closely related to monotonicity is the *dominance* relationship between a pair of points p_1, p_2. Specifically, we say p_1 *dominates* p_2 *with respect to a monotone function* f, if the following condition holds on every dimension i ($1 \leqslant i \leqslant d$) : $p_1.A_i \geqslant p_2.A_i$ ($p_1.A_i \leqslant p_2.A_i$) if f is increasingly (decreasingly) monotone on this axis. For example, given $f(t) = t.A_1 + t.A_2$ (increasingly monotone on both attributes), point t_{11} (coordinate (0.7, 0.6)) dominates t_7 (0.4, 0.3) in Fig. 3, while, for $f(t) = t.A_1 - t.A_2$, t_9 dominates t_7. Note that, the dominance relationship does not necessarily exist for all pairs of points. For instance, t_{11} and t_7 do not dominate each other with respect to $f(t) = t.A_1 - t.A_2$. Note that the concept of "dominance" is *not* applicable to non-monotone functions.

We use a single R-tree on T to efficiently process all top-k queries, regardless of the value of k and the preference function used. Since the R-tree is a dynamic index, data updates are efficiently supported. Unlike *Prefer*, we do not assume any particular "type" of preferences, but aim at all preference "types" using the same tree. We achieve this by applying the branch-and-bound framework. To intuitively explain why the framework is useful in this scenario, we show the connection between top-k search and k NN retrieval. Consider a top-k query q with a preference function $f(t)$ that is increasingly monotone on all dimensions. Let us formulate a k NN query q_{nn}, whose query point lies at the "maximal corner" of the data space (the corner has coordinate 1 on all axes). Unlike a traditional NN query, however, the distance between a data point p and q_{nn} is computed as $dist(p, q_{nn}) = f(q_{nn}) - f(p)$. Obviously, the k objects that minimize $dist(p, q_{nn})$, maximize function f. Therefore, we have reduced the top-k query into a k-NN one, which, as discussed in Section 2.2, can be solved using the BF algorithm.

Motivated by this, in the next section we present a new top-k algorithm BRS (branch-and-bound ranked search) which is similar to BF, but fixes several problems that prevent its immediate application in ranked search. First, observe that in the $dist(p, q_{nn})$ formulated earlier, the term $f(q_{nn})$ is a constant, which implies that the introduction of q_{nn} is essentially "dummy". Namely, we needed it to clarify the connection between top-k and k NN, but it is not necessary in query processing. This is captured in BRS, which does not transform a ranked query to any k NN search, but solves it *directly* utilizing the characteristics of the problem. Second, the distance function $dist(p, q_{nn})$ invalidates the *mindist* definition in BF (which is for the Euclidean distance). BRS is based on an alternative metric *maxscore*, and a novel algorithm that evaluates *maxscore* for intermediate entries. In the next section, we elaborate the details of BRS, focusing on monotone functions. Then, Section 5 extends the technique to other problem variations, including the support for non-monotone functions.

4. Ranked search on monotone preference functions

Section 4.1 presents BRS for top-k queries, supporting arbitrary monotone functions, and Section 4.2 analyzes its performance and proves its optimality. Section 4.3 introduces a technique that estimates the retrieval cost with the aid of histograms. Section 4.4 proposes a technique that reduces the space consumption.

4.1. Problem characteristics and BRS

We aim at reporting the top-k objects in descending order of their scores (i.e., the tuple with the highest score is returned first, then the second highest, and so on). Towards this, we formulate the concept of *maxscore*, which replaces *mindist* in BF. For a leaf entry (a data point), *maxscore* simply equals its score evaluated using the given preference function f. The *maxscore* of an intermediate entry E, on the other hand, equals the largest score of any

Algorithm get_*maxscore* (M=(l_1,h_1, l_2,h_2,…, l_d,h_d), f)
/* M is a MBR with extent [l_i, h_i] along the i-th dimension ($1 \leq i \leq d$), and f the preference function */
1. initiate a point p whose coordinates are not decided yet
2. for i=1 to d /* examine each dimension in turn */
3. if f is increasingly monotone on this dimension
4. the i-th coordinate of p is set to h_i
5. else the i-th coordinate of p is set to l_i
6. return $f(p)$

end get_*maxscore*

Fig. 4. Algorithm for computing *maxscore* for monotone functions.

point that *may* lie in the subtree of E. Similar to *mindist*, *maxscore* is conservative since it may not be necessarily achieved by a point that *actually* lies in E. Next we explain how to compute *maxscore* (for intermediate entries), utilizing the following property of the dominance relationship.

Lemma 4.1. *Given two points p_1, p_2 such that p_1 dominates p_2 with respect to a monotone function f, then $f(p_1) \geqslant f(p_2)$.*

Proof. Due to symmetry, it suffices to prove the case where f is increasingly monotone on all axes. Since p_1 dominates p_2, $p_1.A_i \geqslant p_2.A_i$ on all dimensions $1 \leqslant i \leqslant d$. By the definition of "increasingly monotone", we have $f(p_1.A_1, p_1.A_2, \ldots, p_1.A_d) \geqslant f(p_2.A_1, p_1.A_2, \ldots, p_1.A_d) \geqslant f(p_2.A_1, p_2.A_2, \ldots, p_1.A_d) \geqslant \cdots \geqslant f(p_2.A_1, p_2.\ A_2, \ldots, p_2.A_d)$, thus completing the proof. □

In fact, given the MBR of an intermediate entry E and any monotone function f, we can always find a corner of the MBR, which dominates all points in the MBR. The "dominating corner", however, is not fixed, but instead varies according to f. As an example, consider $f(t) = A_1 + A_2$ and entry E_5 in Fig. 3. Since $f(t)$ is increasingly monotone on both dimensions, the dominating corner is the top-right corner of E_5. For $f(t) = A_1 - A_2$, however, the dominating corner becomes the bottom-right one. As a result, combining with Lemma 4.1, the *maxscore* of an intermediate entry is simply the score of its dominating corner.

A naïve solution for identifying the dominating corner is to evaluate the scores of all the corners which, however, scales exponentially with the dimensionality d (i.e., there are totally 2^d corners of a d-dimensional box). In order to decrease the CPU time, we present an alternative method, which requires $O(d)$ time. The idea is to decide the dominating corner by analyzing the "monotone direction" of f on each dimension (i.e., whether it is increasingly or decreasingly monotone). The monotone direction can be checked in $O(1)$ time per axis by first arbitrarily choosing two points p_1, p_2 whose coordinates differ only on the dimension being tested, and then comparing $f(p_1)$ and $f(p_2)$. Further, the test needs to be performed only once, and the monotone directions can be recorded, using $O(d)$ space, for future use. Fig. 4 shows the pseudo-code of algorithm *get_maxscore* for computing *maxscore*, which decides the dominating corner p of a MBR E as follows. Initially, the coordinates of p are unknown, and we inspect each dimension in turn. If f is increasingly/decreasingly monotone on the ith dimension, then we set the ith coordinate of p to the upper/lower boundary of E on this axis. When the algorithm terminates, all the coordinates of p are decided, and the algorithm *get_maxscore* simply returns the score of p.

Lemma 4.2. *Let intermediate entry E_1 be in the subtree of another entry E_2. Then the maxscore of E_1 is no larger than that of E_2, for any monotone or non-monotone function f.*

Proof. The correctness of the lemma follows from the fact that, every point in E_1 lies in E_2, too. Therefore, the *maxscore* of E_2 is at least as large as that of E_1. □

Based on these observations, BRS traverses the R-tree nodes in descending order of their *maxscore* values. Similar to BF (which accesses the nodes in ascending order of their *mindist*), this is achieved by maintaining the set of entries in the nodes accessed so far in a heap H, sorted in descending order of their *maxscore*. At each step, the algorithm de-heaps the entry having the largest *maxscore*. If it is a leaf entry, then the corresponding data point is guaranteed to have the largest score, among all the records that have not been reported yet. Hence, it is reported directly. On the other hand, for each

```
Algorithm BRS (RTree, f, k}
/* RTree is the R-tree on the data set, f is the preference function, and k denotes how many points to return */
1.  initiate the candidate heap H  /* H takes entries in the form (REntry, key) and manages them in descending
    order of key (REntry is an entry in RTree) */
2.  initiate a result set S with size k
3.  load the root of RTree, and for each entry e in the root
4.      e.maxscore=get_maxscore(e.MBR, f)  // invoke the algorithm in Figure 4
5.      insert (e, e.maxscore) into H
6.  while (S contains less than k objects)
7.      he=de-heap(H)
8.      if he is a leaf entry, then add he to S, and return S if it contains k tuples
9.      else for every entry e in he.childnode
10.         e.maxscore=get_maxscore(e.MBR, f)
11.         insert (e, e.maxscore) into H
12. return S
end BRS
```

Fig. 5. The BRS algorithm.

de-heaped intermediate entry, we visit its child node, and en-heap all its entries. The algorithm terminates when k objects have been de-heaped (they constitute the query result). Fig. 5 formally summarizes BRS.

As an example, consider a top-1 query q with $f(t) = A_1 + A_2$ in Fig. 3. BRS first loads the root (node R) of the R-tree, and inserts its entries to the heap H with their *maxscore* (1.4, 1.3 for E_5 and E_6, respectively). The next node visited is the child N_5 of E_5 (since E_5 has higher *maxscore* than E_6), followed by E_2, after which the content of H becomes $\{(E_6, 1.3), (t_6, 1.2), (t_5, 1.1), (t_4, 1.1), (E_1, 0.8)\}$. The next entry removed is E_6, and then E_4, and at this time $H = \{(t_{11}, 1.3), (t_{12}, 1.2), (t_6, 1.2), (t_{10}, 1.1), (t_5, 1.1), \ldots\}$. Since now the top of H is a data point t_{11}, the algorithm returns it (i.e., it has the largest score), and terminates. Note that, similar to BF for NN search, BRS can be modified to report the tuples in descending score order, without an input value of k (e.g., the user may terminate the algorithm when satisfied with the results).

4.2. I/O optimality

Similar to BF, BRS (of Fig. 5) is *optimal*, since it visits the smallest number of nodes to correctly answer any top-k query. The *optimal cost* equals the number of nodes, whose *maxscore* values are larger than the kth highest object score s_k. Note that, a node visited by BRS may not necessarily contain any top-k result. For example, consider an intermediate node E_1 with *maxscore* larger than s_k, while the *maxscore* of all nodes in its subtree is smaller than s_k (this is possible because $E_1.maxscore$ is only an upper bound for the *maxscore* of nodes in its subtree). In this case, none of the child nodes of E_1 will be accessed, and therefore, no point in E_1 will be returned as a top-k result. However, even in this case, access to E_1 is inevitable. Specifically, since $E_1.maxscore > s_k$, there is a chance for some point in E_1 to achieve a score larger than s_k, which cannot be safely ruled out unless the child node of E_1 is actually inspected. Based on this observation, the next lemma establishes the optimality of BRS.

Lemma 4.3. *BRS algorithm is optimal for any top-k query, i.e., it accesses only the nodes whose maxscore values are larger than* s_k *(the kth highest object score).*

Proof. We first show that BRS always de-heaps (i.e., processes) the (leaf and intermediate) entries in descending order of their *maxscore*. Equivalently, let E_1, E_2 be two entries de-heaped consecutively (E_1 followed by E_2); it suffices to prove $E_1.maxscore \geqslant E_2.maxscore$. There are two possible cases: (i) E_2 already exists in the heap H when E_1 is de-heaped, and (ii) e_2 is in the child node of e_1. For (i), $E_1.maxscore \geqslant E_2.maxscore$ is true because H removes entries in descending order of their *maxscore*, while for (ii) $E_1.maxscore \geqslant E_2.maxscore$ follows Lemma 4.2. To prove the original statement, consider any node N whose *maxscore* is smaller than s_k, which is the kth highest object score (let o denote the object achieving this score). According to our discussion earlier, BRS processes o before the parent entry E of N, meaning that at the time o is reported, N has not been visited. □

The optimality of BRS can be illustrated in a more intuitive manner, using the concept of "identical score curve" (ISC). Specifically, the ISC

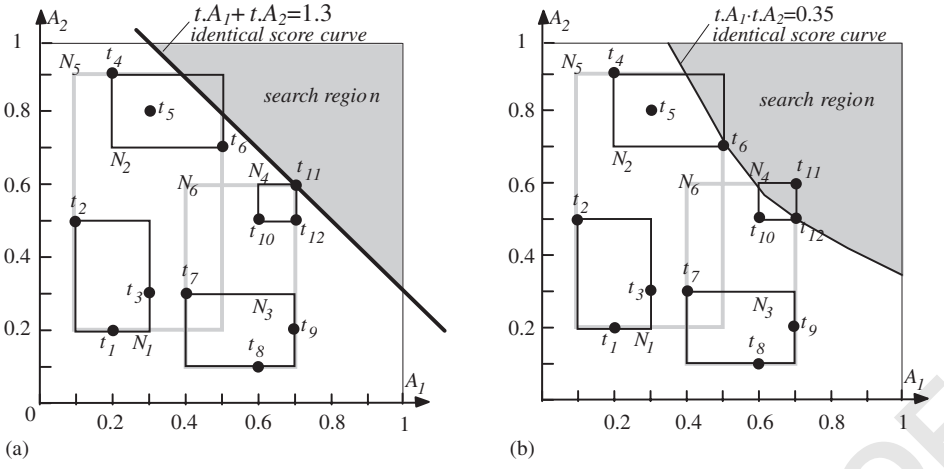

Fig. 6. The identical score curves. (a) For top-1 query with $f(t) = t.A_1 + A_2$, (b) for top-3 query with $f(t) = t.A_1 + t.A_2$.

is a curve corresponding to equation $f(t) = v$, which consists of points in the data space whose scores equal v. Figs. 6a and b illustrate the ISCs of $f(t) = A_1 + A_2 = 1.3$ and $f(t) = A_1 \cdot A_2 = 0.35$, respectively. An important property of ISC $f(t) = v$ is that it divides the data space into two parts, referred to as "large" and "small" parts in the sequel, containing the points whose scores are *strictly* larger and smaller than v, respectively. In Figs. 6a and b, the larger parts are demonstrated as the shaded areas. Let s_k be the kth highest object score of a top-k query. Then, the ISC $f(t) = s_k$ defines a "search region", which corresponds to the larger part of the data space divided by $f(t) = s_k$. According to Lemma 4.3, to answer the query, BRS accesses only the nodes whose MBRs intersect the search region (the *maxscore* of any other node must be smaller than s_k). In Fig. 6a (b), the plotted ISC $f(t) = 1.3$ (0.35), where 1.3 (0.35) is the highest (third highest) object score according to $f(t)$. Thus, to answer the top-1 (-3) query in Fig. 6a (b), BRS visits the root, N_5, N_2, N_6, N_4, as their MBRs intersect the shaded area.

An interesting observation is that, for top-k queries with small k, BRS visits only the nodes whose MBRs are close to the corners of the data space, i.e., its node accesses clearly demonstrate a "locality" pattern. This has two important implications. First, nodes whose MBRs are around the center of the data space may never be retrieved, which motivates a space-reduction technique in Section 4.4. Second, consecutive executions of BRS are likely to access a large number of common nodes. Hence, the I/O cost of BRS can be significantly reduced by introducing a buffer (for caching the visited disk pages), as confirmed in the experiments.

4.3. Query cost estimation

In practice, the system should be able to estimate the cost of a top-k query, in order to enable query optimization for this operator. Motivated by this, we develop a method that can predict the I/O overhead of BRS for any monotone preference function. For simplicity, we aim at estimating the number of leaf node accesses because (i) it dominates the total cost, as is a common property of many algorithms based on indexes [20], and (ii) the extension to the other levels of the tree is straightforward.

Our technique adopts multi-dimensional histograms [21–24]. Specifically, a histogram partitions the data space into disjoint rectangular *buckets*, the number of which is subject to the amount of the available main memory. The goal of partitioning is to make the data distribution *within each bucket* as uniform as possible (i.e., the overall distribution is approximated with piecewise-uniform assumption). We maintain two histograms: the first one HIS_{data} on the given (point) dataset, and the other HIS_{leaf} on the MBRs of the leaf nodes. In HIS_{data}, each bucket b is associated with the number $b.num$ of points in its extent, while for HIS_{leaf}, we store in each bucket b (i) the number $b.num$ of leaf MBRs whose centroids fall inside b, and (ii) the average extent $b.ext$ of these MBRs.

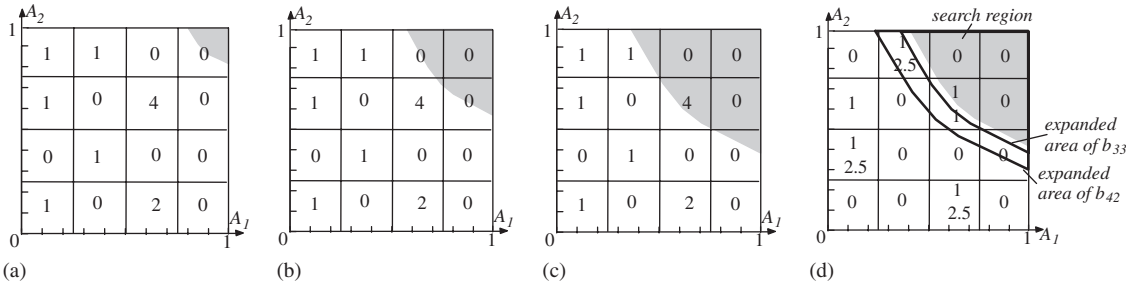

Fig. 7. Equi-width histograms and procedures in query estimation. (a) HIS_{data} and the first tested search region, (b) the second tested search region, (c) the third tested search region, (d) HIS_{leaf} and the final estimated search region.

For simplicity, our implementation adopts the *equi-width histogram* [21], which partitions the data space into c^d regular buckets (where c is a constant called the *histogram resolution*). Figs. 7a and d demonstrate the buckets of HIS_{data} and HIS_{leaf} ($c = 4$) for the dataset and leaf MBRs in Fig. 3, respectively. In Fig. 7d, for each non-empty bucket, the upper and lower numbers correspond to $b.num$ and $b.ext$, respectively (e.g., the number 2.5 of the non-empty bucket in the first row is the average of the width and height of MBR N_2). It is worth mentioning that, both HIS_{data} and HIS_{leaf} can be maintained efficiently: whenever there is a change in the dataset or leaf level, it is intercepted to update HIS_{data} or HIS_{leaf}, respectively (histogram updates are well-studied in [23]).

To estimate the query cost, we first predict the size of the search region SR based on HIS_{data}. Consider the top-3 query with preference function $f(t) = t.A_1 \cdot t.A_2$ in Fig. 6b. To predict its SR, we first sort the centroids of all the buckets in HIS_{data}, in descending order of their scores. In Fig. 7a, the sorted order is b_{44} (the subscript indicates the 4th row, 4th column, counting from the bottom and left, respectively), $b_{43}, b_{33}, b_{32}, \ldots$. Then, we incrementally increase SR by examining the buckets in the sorted order, until SR contains expectedly k points. For example, to process the first bucket b_{44}, we focus on the ISC passing its centroid, i.e., $f(t) = t.A_1 \cdot t.A_2 = f(b_{44}) = 0.81$ (we denote the centroid using the same symbol as the bucket). The shaded area in Fig. 7a illustrates the SR decided by this ISC. In this case, since b_{44} is the only bucket intersecting SR, the number of points in SR can be estimated as $b_{44}.num \cdot area(b_{44} \cap SR)/area(b_{44}) = 0$ (since $b_{44}.num = 0$).

In general, to estimate the expected number of points in a search region, we inspect all the buckets that intersect it. In particular, for each such bucket b, we compute its *contribution* (i.e., the number of points in b that fall in SR), and sum up the contributions of all buckets as the final estimate. Since the point distribution in a bucket is uniform, the contribution of a bucket b equals $b.num \cdot area(b \cap SR)/area(b)$, where $area(b \cap SR)$ and $area(b)$ denote the areas of b and the intersection between b and SR, respectively. Unfortunately, for general preference functions, $b \cap SR$ is usually an irregular region whose area is difficult to compute. We solve this problem numerically using the Monte-Carlo method as follows. A set of points (let the number be α) is first generated randomly inside the bucket. Then, the number β of points falling in SR is counted (this can be done by evaluating the score of each point), after which $area(b \cap SR)/area(b)$ can be roughly calculated as β/α.

Continuing the example, since the current SR contains too few points (less than $k = 3$), it needs to be expanded. Thus, we examine the second bucket b_{43} in the sorted list, take the ISC passing its centroid, and estimate the number of points in the new SR as described above. As shown in Fig. 7b, the SR intersects a non-empty bucket b_{33}, and hence, the estimated number is a non-zero value e_{43} (the subscript indicates the value is obtained when inspecting b_{43}). Fig. 7c shows the further expanded SR according to the next bucket b_{33}. Assuming that the estimated value $e_{33} > 3$, the current SR needs to be shrunk based on e_{33} and e_{43} (i.e., the previous estimate). We obtain the final SR according to the ISC $f(t) = (k - e_{43})(f(b_{33}) - f(b_{43}))/e_{33} - e_{43} + f(b_{43})$, i.e., the position of this ISC is obtained by interpolating linearly[1] the ISCs

[1]Strictly speaking, linear interpolation is not the best interpolation in all cases. Nevertheless, we apply it anyway since it is simple and produces satisfactory estimation as shown in the experiments.

that produced e_{33} and e_{43}. Fig. 7d illustrates the final SR.

Having obtained the estimated SR, we proceed to predict the number of leaf accesses based on Lemma 4.3, utilizing the following observation: if an MBR E intersects an SR, then the centroid of E lies in an "expanded search region", which is obtained by extending SR towards the negative direction of each dimension by $l/2$, where l is the projection length of E on that dimension. Based on the idea, for each bucket b in HIS_{leaf}, we expand SR by $b.ext/2$ towards the negative direction of all dimensions. Let SR' be the expanded search region. Then, the number of leaf MBRs (whose centroids fall) in b that intersect SR can be estimated as $b.num \cdot area(b \cap SR')/area(b)$ (i.e., the contribution of b to the final estimate).

To compute $area(b \cap SR')/area(b)$, we again resort to the Monte-Carlo method (for the same reasons as in computing $area(b \cap SR)/area(b)$). Particularly, to test if a point (generated randomly in b) is in SR', we increase its coordinates by $b.ext/2$ on all axes, and check if the resulting point falls in SR (by computing its score). The total number of leaf nodes visited equals the sum of the contributions of all the buckets. Fig. 7d demonstrates the expanded SR with respect to buckets b_{33} and b_{42}, where SR is expanded by 0.5 and 1.25, respectively (the estimate of the other buckets is 0). Judging from the intersection areas (between the expanded SR and the corresponding buckets), the final predicted cost is close to 2 leaf accesses, which is the real query cost as shown in Fig. 6.

Although we used an equi-width histogram as an example, the above discussion can be easily extended to any histogram with the same bucket representation. In particular, if the first few levels of the underlying R-tree can be pinned in memory (as is often the case in practice), we can treat the MBRs of the memory-resident nodes as the buckets, and perform cost estimation as described earlier.

4.4. Reducing the size of the R-tree

In practice, the number k of objects requested by a query is expected to be small compared to the cardinality of the dataset. Interestingly, if all the queries aim at obtaining no more than K tuples (i.e., $k < K$), where K is an appropriate constant (for most applications, in the order of 100 [3]), some records may never be retrieved, regardless of the concrete (monotone) preference functions. These "inactive" records can be safely removed from the R-tree (without affecting the query results), thus reducing the space requirements. In Fig. 6a, for example, it can be verified (as explained shortly) that t_{10}, t_3, t_7 are inactive for top-1 queries, while the inactive records for $K = 2$ include t_{10} and t_7 (all points are active for $K \geqslant 3$).

To analyze the properties of active data, let us first consider, without loss of generality, preference functions that are increasingly monotone on all dimensions. Recall that, for such functions, a point p_1 dominates p_2 if the coordinates of p_1 are larger than those of p_2 on all dimensions. A crucial observation is that, an object is inactive for top-K, if and only if it is dominated by *at least* K other objects. For instance, if $K = 1$, then t_{10} in Fig. 6a is inactive since it is dominated by t_{11}. In fact, the active records for $K = 1$ consist of t_4, t_5, t_6, t_{11}, which constitute the *skyline* [15] of the dataset.

We present a solution that is applicable to any K. To apply the proposed algorithm, we need to select a *representative* function, which can be any function increasingly monotone on all axes, e.g., $f(t) = t.A_1 \cdot t.A_2$. Similar to BRS, we maintain all the entries in the nodes visited so far using a heap H, and process them in descending order of their *maxscore* according to $f(t)$. Unlike BRS, however, new heuristics are included to prune nodes that cannot contain any active record. To illustrate, assume that we want to discover the active set for $K = 2$, on the dataset in Fig. 6a. Initially, H contains root entries E_5, E_6 with *maxscore* 0.45, 0.42, respectively. Since E_5 has higher *maxscore*, its child N_5 is retrieved, leading to $H = \{(E_2, 0.45), (E_6, 0.42), (E_1, 0.15)\}$. The nodes visited next are N_2 and N_6, after which $H = \{(t_{11}, 0.42), (t_{12}, 0.35), (t_6, 0.35), (t_{10}, 0.3), (t_5, 0.24), (E_3, 0.21), (t_4, 0.18), (E_1, 0.15)\}$. The entry t_{11} that tops the heap currently, is the first active object, and is inserted into the active set AS. Similarly, the next object t_{12} is also added to AS, which contains $K = 2$ records now, and will be taken into account in the subsequent processing. Specifically, for each leaf (intermediate) entry de-heaped from now on, we add it to AS (visit its child node) only if it is not dominated by more than $K\,(=2)$ points currently in AS. Continuing the example, since t_6 is not dominated by any point in AS, it is active, while t_{10} is inactive as it is dominated by t_{11}, t_{12}. For the remaining entries in H, t_5 and t_4 are inserted to AS, while E_3 and E_1 are pruned (they are dominated by t_{11}, t_{12}), meaning that they cannot contain any active

object. The final active set includes $AS = \{t_{11}, t_{12}, t_6, t_5, t_4\}$.

Extending the above method to preference functions with other dimension "monotone direction" combinations is straightforward. For example, in Fig. 6a, to find the active set for functions increasingly monotone on A_1 but decreasingly on A_2, we only need to replace the representative function to $f(t) = (-t.A_1) \cdot t.A_2$ (or any function conforming to this monotone-direction combination). In general, the final active records include those for all the combinations. We index the active set with an R-tree, which replaces the original R-tree (on the whole dataset) in answering top-k queries ($k \leqslant K$). It is worth mentioning that, this technique of reducing the space overhead is best suited for static datasets. If data insertions/deletions are frequent, the active set needs to be maintained accordingly, thus compromising the update overhead.

5. Alternative types of ranked queries

In the last section, we have shown that conventional top-k search (of any monotone preference function) can be efficiently supported using a single multi-dimensional access method. In this section, we demonstrate that this indexing scheme also permits the development of effective algorithms for other variations of the problem, which are difficult to solve using the existing methods. Specifically, Section 5.1 discusses ranked retrieval on the data satisfying certain range conditions, and Section 5.2 focuses on the "group-by ranked query", which finds multiple top-k lists in a subset of the dimensions simultaneously. Finally, Section 5.3 concerns top-k queries for non-monotone preference functions.

5.1. Constrained top-k queries

So far our discussion of ranked search considers all the data records as candidate results, while in practice a user may focus on objects satisfying some constraints. Typically, each constraint is specified as a range condition on a dimension, and the conjunction of all constraints forms a (hyper-) rectangular *constraint region*. Formally, given a set of constraints and a preference function f, a *constrained top-k* query finds the k objects with the highest scores, among those satisfying *all* the constraints. Consider, for example, Fig. 8, where each point captures the current price and turnover of a stock. An investor would be interested in only the stocks whose price (turnover) is in the range (0, 0.5] ((0.6, 0.8]). In Fig. 8, the qualifying stocks are those (only t_5, t_6) in the dashed constraint region. The corresponding constrained top-1 query with $f(t) = t.price + t.turnover$ returns t_6, since its score is higher than t_5. In this example, the constraint region is a rectangle, but, in general, a more complex region may also be issued by a user (e.g., the user may be interested only in stocks satisfying $t.price + t.turnover < 1.5$).

With some minor modifications, BRS can efficiently process any constrained top-k query. Specifically, the only difference from the original BRS is that an intermediate entry is inserted into the heap H, only if its MBR intersects the constraint region CR. For the query in Fig. 8, after retrieving the root, the algorithm only inserts E_5 into H, but not E_6 (as no point in its subtree can satisfy all the constraints). Further, the *maxscore* of E_5 is calculated as that of the intersection between its MBR and CR. In particular, the rationale of taking the intersection is to exclude points in E_5 falling outside CR from the *maxscore* computation. In this example, the *maxscore* of E_5 equals 0.13 (instead of 0.14, as for the top-1 query with the same $f(t)$ on the whole dataset). Next the algorithm visits the child N_5 of E_5, and inserts only entry E_2 (E_1 is pruned since it does not intersect CR). Finally, the algorithm accesses N_2, returns t_6, and finishes. Following a derivation similar to Section 4.2, it is easy to show that the modified BRS also achieves the optimal I/O performance for all queries. Fig. 8 demonstrates the ISC passing the final result t_6. The optimal cost here corresponds to the number of nodes (N_5, N_2) whose MBRs intersect the shaded region, which is bounded by the ISC and CR.

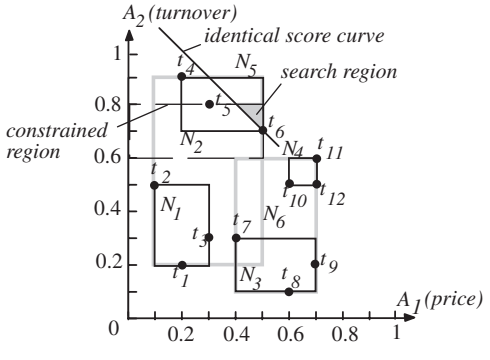

Fig. 8. A constrained top-1 query.

The existing methods cannot support constrained top-k queries efficiently. As discussed in [1], *Onion* needs to create a separate set of convex hulls for *each* constraint region, assuming that possible constraints are *known in advance. Prefer*, on the other hand, treats a constrained query simply as a normal one (i.e., ignoring all the range conditions). Specifically, it examines objects in descending order of their scores, and stops when k constraint-qualifying objects have been found. In contrast, our algorithm concentrates the search on only the qualifying objects, and thus avoids the overhead of inspecting unnecessary data.

5.2. Group-by ranked search

A *group-by top-k* query first groups the records by a (discrete) attribute A_G, and then retrieves the top-k tuples for each group, using a preference function that evaluates the score of an object according to the *remaining* attributes (excluding A_G). Consider a database that stores the following information of hotels: the price (one night's rate), distance (from the town center), and class (five-star, four-star, etc.). A group-by top-k instance would find the best k hotels in *each class* (i.e., $A_G =$ class), i.e., the result consists of five lists, the first containing the best 1-star hotels, the second the best 2-star hotels and so on. A naïve solution is to build a 2D R-tree (on price and distance) for the hotels of each class, so that the top-k list of a class can be found using the corresponding R-tree. However, these R-trees cannot be used to process general ranked queries (e.g., those involving all three attributes). Recall that our goal is to use a *single* "global" index on all attributes (in this case, a 3D R-tree) to handle *all* query types. To achieve this, we provide a method that answers any group-by query using directly the global index.

An important observation is that the top-k objects in each group can be retrieved using constrained top-k search. To find the top-k i-star hotels ($1 \leqslant i \leqslant 5$), for instance, we can apply the modified BRS in Section 5.1, with the constraint region being a 2D plane that is perpendicular to the "class" axis, and crosses this axis at i. Issuing a separate constrained query for each group, however, may access the same node multiple times (e.g., the root, obviously, must be loaded in each query), thus compromising the processing time. In particular, the query cost increases linearly with the number g of groups, and may be prohibitive if g is large. To remedy this, we propose the Group-by BRS (GBRS) algorithm of Fig. 9, which finds the results of all groups by traversing each node at most once. The key idea is to process all the constrained queries in a simultaneous manner, by maintaining g top-k lists $L_1, L_2, \ldots, L_g$, storing the top-k objects

Algorithm get_*g-maxscore* (E=(l_1,h_1, l_2,h_2,…, l_d,h_d), f, g, L_i)
/* E is a MBR with extent [l_i, h_i] along the i-th dimension ($1 \leq i \leq d$), f a preference function on the non-grouped attributes, g is the number of groups and L_i the result list of the i-th constrained query ($1 \leq i \leq g$) */
1. if the lists L_i of all queries q_i spanned by E contain k objects, return $-\infty$
2. let E' be the projection of E onto the non-grouped attributes
3. return *get_maxscore*(E', f) //invoke the algorithm in Figure 4

end get_*maxscore*

Algorithm GBRS (*RTree*, f, k, g}
/* *RTree* is the R-tree on the dataset, f is the preference function, k denotes how many points to return, and g is the number of groups */
1. initiate the candidate heap H /* H takes entries in the form (*REntry*, *key*) and manages them in descending order of *key* (*REntry* is an entry in *RTree*) */
2. initiate g lists L_1, L_2, …, L_g with size k
3. load the root of *RTree*, and for each entry e in the root
4. e.*maxscore*=**get_*g-maxscore***(e.MBR, f, g, L_i)
5. insert (e, e.*maxscore*) into H
6. while (some L_i ($1 \leq i \leq k$) contains less than k objects)
7. *he*=de-heap(H)
8. if (**get_*g_maxscore***(*he*.MBR, f, g, L_i)=$-\infty$) continue; //to de-heap the next entry
9. if *he* is a leaf entry then add *he* to the list L_i of the query corresponding to its group
10. else for every entry e in *he.childnode*
11. e.*maxscore*=**get_*g-maxscore***(e.MBR, f)
12. if e.*maxscore*$\neq -\infty$ then insert (e, e.*maxscore*) into H
13. return L_1, L_2, …, L_g

end GBRS

Fig. 9. The group-by top-k algorithm.

for each query. In particular, the ith object ($1 \leqslant i \leqslant k$) inserted to list L_j ($1 \leqslant j \leqslant g$) is guaranteed to be the one with the ith highest score for the jth (constrained) query. Therefore, GBRS terminates as soon as all the lists contain k objects.

Given an MBR E, a constrained query q is said to be *spanned* by E, if the group represented by q is covered by the projection of E on A_G. We define the *g-maxscore* of E as follows: (i) it equals $-\infty$, if all the queries spanned by E have already found k objects; (ii) otherwise, it is the *maxscore* of E (i.e., according to $f(t)$). Unlike *maxscore*, the *g-maxscore* of E may change during the execution of GBRS. In particular, at the beginning, *g-maxscore* is set to *maxscore*, but becomes $-\infty$ as soon as all queries E spans have retrieved their top-k, and stays at this value afterwards. GBRS, similar to BRS, uses a heap H to manage all the entries that have been encountered so far, but differs from BRS in two ways. First, the entries in H are sorted by their *g-maxscore*. Second, every time an entry E is de-heaped, its *g-maxscore* is re-computed. If its current *g-maxscore* is not $-\infty$, we visit its subtree (for an intermediate E) or add it to the appropriate result list L_i (for a leaf E); otherwise, E is simply discarded.

In the above discussion, we assume that the g lists $L_1, L_2, \ldots, L_g$ can be stored in memory. If this is not true, our algorithm can be easily modified to process as many groups as possible at a time, subject to the available amount of memory. Neither *Onion* nor *Prefer* is applicable to group-by top-k retrieval. Specifically, in *Onion* (*Prefer*) the convex hulls (*views*) computed in the original data space (involving all the dimensions) are useless for ranked search in individual groups. As a result, to support group-by search, both methods require dedicated pre-computation (for all possible groups), thus significantly increasing the space consumption (especially if multiple axes can be the grouping dimensions).

5.3. Non-monotone preference functions

The existing methods on ranked search assume monotone preference functions. One major difficulty supporting non-monotonicity is the lack of "dominance" relationship between a pair of points (recall that, "dominance" is coupled with the increasing/decreasing monotonicity on individual axes, which is undefined for non-monotone preferences). For *Onion*, the absence of "dominance" invalidates the underlying assumption that the result of a top-1 query lies on the convex hull (as mentioned in Section 2.1, this assumption holds *only* for linear preferences). *Prefer*, on the other hand, relies on the pre-sorted object scores according to some selected functions. For non-monotone functions, however, the ordering of object scores according to a query preference can deviate significantly from all the pre-computed orderings, thus impairing the pruning ability of this technique.

BRS, however, can be adapted to support a large number of non-monotone preference functions f. In particular, the algorithm in Fig. 5 is applicable as long as the *maxscore* of an MBR E can be correctly evaluated, with respect to the given f. Note that, the original *maxscore* computation algorithm (Fig. 4) is not applicable since, in the non-monotone case, the point in E achieving the highest score is not necessarily a corner of E (again, due to the invalidation of "dominance"). Instead, we can compute the *maxscore* following a different approach, assuming that f has partial derivative everywhere on all dimensions.

Recall that the goal of computing *maxscore* is to maximize $f(t.A_1, t.A_2, \ldots, t.A_d)$, given that $l_i \leqslant t.A_i \leqslant h_i$ for ($1 \leqslant i \leqslant d$). The standard mathematical way to solve this problem is as follows. Take the derivative of f with respect to each variable $t.A_i$, and set the resulting formula to 0. Thus, we obtain i equations $\partial f/\partial t.A_i = 0$. Each solution of this equation set gives the coordinates of an *extreme point* that, if falling in E, *may* achieve the *maxscore* for E. Let S_E be a set of all such points. Then, the *maxscore* in E equals the maximum score of the points in S_E, and the points on the boundary of E. Note that equation set $\partial f/\partial t.A_i = 0$ ($1 \leqslant i \leqslant d$) only needs to be solved once, and the solution can be used in the *maxscore* computation of all MBRs.

As an example, consider $f(t) = -(t.A_1 - 0.5)^2 \cdot (t.A_2 - 0.5)^2$, and an MBR E whose A_1-projection is an interval [0.6, 0.8], and its A_2-projection is [0.4, 0.6]. We take $\partial f/\partial t.A_1 = -2(t.A_2 - 0.5)^2 \cdot (t.A_1 - 0.5)$, and $\partial f/\partial t.A^2 = -2(t.A_1 - 0.5)^2 \cdot (t.A_2 - 0.5)$. Setting both equations to 0, we obtain the only solution $t.A_1 = 0.5$, $t.A_2 = 0.5$ (i.e., S_E has only one element). Hence, the *maxscore* of E may equal the score of point (0.5, 0.5) (if this point is covered by E), or the score of some point on the boundary of E. Here, (0.5, 0.5) falls out of E; the *maxscore* must be achieved by a point on the boundary of E. Hence, we consider each edge of E in turn, which essentially performs the above

process recursively in a lower dimensional space. We illustrate this by inspecting the left edge of E, i.e., $t.A_1 = 0.6$. With this equality condition, $f(t)$ becomes $-0.01 \cdot (t.A_2 - 0.5)^2$, which takes its maximum value 0 at $t.A_2 = 0.5$. Since (0.6, 0.5) is a point in E, this is the point having the largest score among all the points on the left edge. In fact, carrying out the above idea to the other edges, it is easy to verify that 0 is indeed the *maxscore* of E.

BRS offers *exact results* to all non-monotone functions for which the equation set $\partial f/\partial t.A_i = 0$ $(1 \leqslant i \leqslant d)$ can be accurately solved. For some functions f, however, $\partial f/\partial t.A_i$ may become excessively complex so that the equation set (and hence, *maxscore*) can only be solved using numerical approaches (see [25]). In this case, BRS provides approximate answers, i.e., the scores of the top-k objects returned may be slightly lower than those of the real ones. The precision of these results depends solely on the accuracy of the numerical method used. Finally, the algorithms discussed in Sections 5.1 and 5.2 can also be extended to support non-monotone functions in the same way.

6. Experiments

In this section, we experimentally study the efficiency of the proposed methods, deploying synthetic datasets similar to those used to evaluate the *Prefer* system [4] (i.e., currently the best method for ranked search). The data space consists of d (varied from 2 to 5) dimensions normalized to [0, 1]. Each dataset contains N (ranging from 10k to 500k) points following the *Zipf* or *correlated* distribution. Specifically, to generate a *Zipf* dataset, we decide the coordinate of a point on each axis independently, according to the *Zipf* [26] distribution (the generated value is skewed towards 0). The creation of a *correlated* dataset follows the approach in [4]. Particularly, the attribute values of a tuple t on the first $\lfloor d/4 \rfloor + 1$dimensions are obtained randomly in [0, 1]. Then, on each remaining axis i $(\lfloor d/4 \rfloor + 2 \leqslant i \leqslant d)$, the attribute value $t.A_i$ is set to $(\sum_{j=1}^{i-1} c_j \cdot t \cdot A_j) - \lfloor \sum_{j=1}^{i-1} c_j \cdot t \cdot A_j \rfloor$, where c_j is a random constant in [0.25, 4]. Datasets created this way have practical correlated coefficients[2] (around 0.5) [4].

[2]The correlated coefficient *COR* measures the degree of correlation between different attributes of a relation. Given two attributes A_1 and A_2, *COR* is $cov(A_1,A_2)/(s_1 \cdot s_2)$, where $cov(A_1,A_2) = E(A_1 \cdot A_2) - E(A_1) \cdot E(A_2)$, and $s_i = \sqrt{E\{[A_i - E(A_i)]^2\}}$.

We compare BRS (and its variants) to *Onion* and *Prefer*. Performance is measured as the average number of disk accesses in executing a "workload" consisting of 200 queries retrieving the same number k of objects. Unless otherwise stated, the preference function $f(t)$ of each query is a linear function $f(t) = \sum_{i=1-d}(w_i \cdot t.A_i)$, where the weights w_i are randomly generated in $[-1, 1]$. Further, the weights are such that no two queries have the same function. Linear functions are selected as the representative preference because they are popular in practice, and constitute the optimization goal in most previous work [1–3].

The disk page size is set to 4K bytes. An R*-tree [6] is created on each dataset, with node capacities of 200, 144, 111, 90 entries in 2, 3, 4 and 5 dimensions, respectively. We disable the system cache in order to focus on the I/O characteristics of each method, except in scenarios that specifically aim at studying the buffered performance. All the experiments are performed on a Pentium IV system with 1GB memory. Section 5.1 first illustrates the results for processing conventional top-k queries, and then Section 5.2 evaluates the techniques for other variations.

6.1. Evaluation of conventional ranked search

We first demonstrate the superiority of BRS over the existing methods, and the efficiency of techniques estimating its query costs. Then, we evaluate the effect of space reduction, as well as the performance of BRS for non-linear monotone functions.

6.1.1. Query cost comparison

Since the performance of *Prefer* depends on the number of materialized views (each equal in size to the entire database), we first decide how many views should be used by *Prefer* for the subsequent experiments. Towards this, we use 3D (*Zipf* and *correlated*) datasets with cardinalities $N = 100$k, and compare the cost of BRS and *Prefer* in processing a workload of top-250 queries.

Fig. 10 shows the speedup of BRS (i.e., calculated as the cost of *Prefer* divided by that of BRS), as a function of the number of views in *Prefer*. When both methods consume the same amount of space (i.e., only 1 view for *Prefer*), BRS is more than 20 times faster. This is expected because, in this case, *Prefer* relies on the tuple ordering according to a single function, and therefore, incurs prohibitive

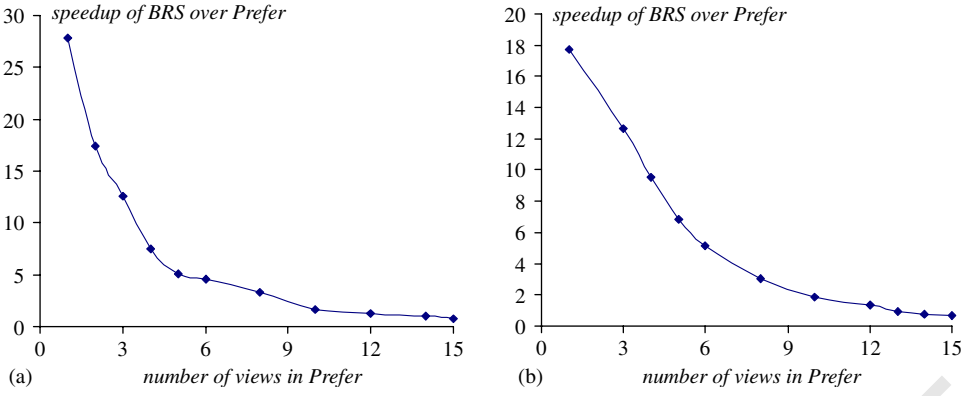

Fig. 10. Speedup of R-trees over *Prefer* ($d = 3$, $N = 100$k, $k = 250$). (a) *Zipf*, (b) *correlated*.

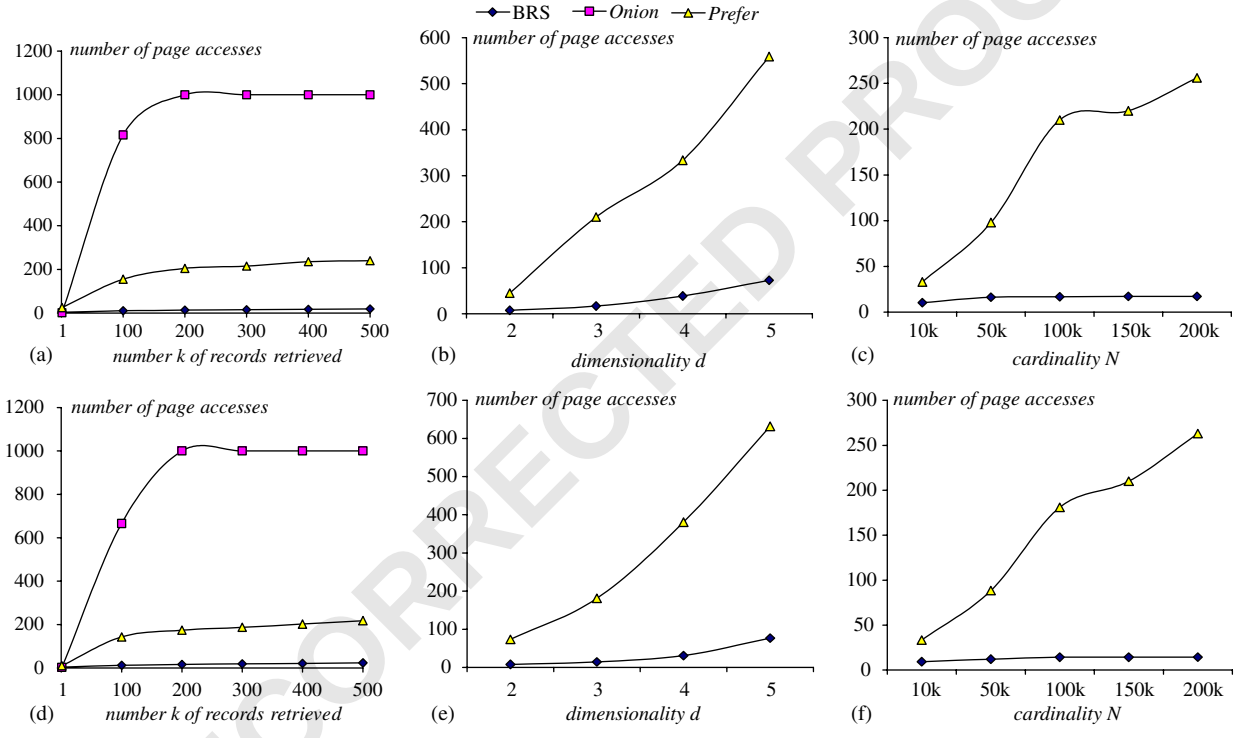

Fig. 11. Comparison of alternative methods. (a) Query cost vs. k ($d = 3$, $N = 100$k), (b) query cost vs. d ($k = 250$, $N = 100$k), (c) query cost vs. N ($k = 250$, $d = 3$), (d) query cost vs. k ($d = 3$, $N = 100$k), (e) query cost vs. d ($k = 250$, $N = 100$k), (f) query cost vs. N ($k = 250$, $d = 3$).

overhead for queries whose preference functions are significantly different. In particular, *Prefer* starts outperforming BRS after its number of views reaches 15 and 13 for *Zipf* and *correlated* datasets, respectively. In the sequel, we report the performance of *Prefer* with three views (i.e., allowing *Prefer* to consume an amount of space three times larger than BRS).

Fig. 11 compares the performance of alternative algorithms for *Zipf* (first row) and *correlated* data (second row). Specifically, Figs. 11a and d plot the number of page accesses (per query) in retrieving various numbers k of objects, using datasets with $d = 3$ and $N = 100$k. It is clear that BRS outperforms *Onion* and *Prefer* significantly, and the difference increases with k. *Onion*, on the other

hand, is by far the most expensive method, i.e., its cost is up to five times higher than *Prefer*, and almost 100 times than BRS. Since *Onion* is considerably slower than the other solutions in all the experiments, we omit it from further discussion.

To study the effect of dimensionality, in Figs. 11b and e, we fix $k = 250$, $N = 100$k, and measure the query overhead as d varies. Both BRS and *Prefer* deteriorate as d increases due to, however, different reasons. The deterioration of BRS is mainly caused by the well-known structural degradation of R-trees in higher dimensional spaces [19]. For *Prefer*, the number of views required to maintain the same query cost grows exponentially with d [4]. Thus, given the same space limit (i.e., three views), its performance drops very fast as d grows. For all the dimensionalities tested, BRS is consistently faster than *Prefer* by an order of magnitude.

Figs. 11c and f compare the two methods for datasets of different cardinalities N ($k = 250$ and $d = 3$). Interestingly, the performance of BRS is hardly affected by N. To explain this, note that as N increases, both the node MBRs and search regions actually decrease (recall that, the search region, defined in Section 4.2, is the part of the data space divided by the ISC $f(t) = s_k$, where s_k is the kth highest object score). As a result, the number of nodes whose MBRs intersect the search region (i.e., the cost of BRS) is not affected seriously. On the other hand, for larger datasets, *Prefer* needs to inspect a higher number of records before reaching the watermark (see Section 2.1), which explains its (serious) performance degradation.

The next experiment studies the effect of an LRU buffer on the query overhead. Specifically, we increase the buffer up to 10% of the database size, and measure the number of page faults of BRS. Fig. 12 shows the results of both data distributions with $k = 3$, $d = 3$, $N = 100$k, respectively. BRS achieves significant improvement even with a small buffer. Particularly, *the average cost is less than* 1 *page access when the buffer size is* 6% *of the dataset*! This is not surprising since, as mentioned in Section 4.2, the search regions of all queries are around the corners of the data space, meaning that the sets of nodes visited by successive queries may have large overlap. As a result, there is a high probability that a node requested by a query already resides in the buffer (i.e., it was accessed by previous queries), thus reducing the I/O cost. Also observe that the cost does not decrease further when the buffer is

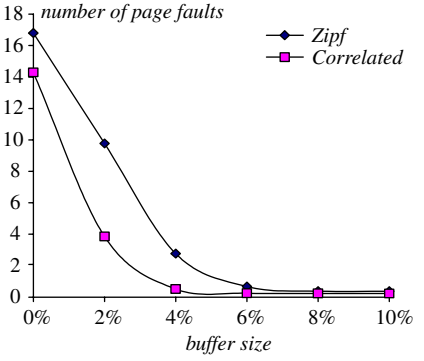

Fig. 12. Performance of BRS vs. buffer size ($k = 250$, $d = 3$, $N = 100$k).

larger than 6%, indicating that the optimal I/O performance has been achieved.

Prefer, on the other hand, receives much less improvement and is slower than BRS by two orders of magnitude (therefore, it is omitted from Fig. 12). This is expected because, given a node capacity of 144 entries, the dataset (with 100k tuples) occupies 700 pages on the disk, so a 6% cache size contains 40 pages. As shown in Fig. 11, *Prefer* accesses on the average 200 pages for each query, which indicates that a query can avoid at most 20% ($= 40/200$) node accesses. Furthermore, the materialization of more views leads to even less improvement since they share the same cache space.

6.1.2. *Quality of cost prediction*

Having established the query efficiency of BRS, we proceed to evaluate the method (proposed in Section 4.3) that predicts its cost using histograms. We adopt the equi-width histogram, which as mentioned in Section 4.3, partitions the data space into c^d equal-size buckets, where c is the histogram resolution, and d the dimensionality. Two histograms (with the same resolution) are maintained to store the distributions of the data and leaf MBRs, respectively. The resolution decreases with the dimensionality in order to keep the memory consumption approximately the same. In particular, we use $c = 50, 30, 14, 6$, for $d = 2, 3, 4, 5$, respectively. To apply the Monte-Carlo method described in Section 4.3, we set $\alpha = 500$ in all cases. The precision is measured as the average estimation error for all the queries in a workload.[3] Formally, let act_i and est_i denote the actual and estimated

[3]In our experiments, the variance of the error for individual queries is not significant.

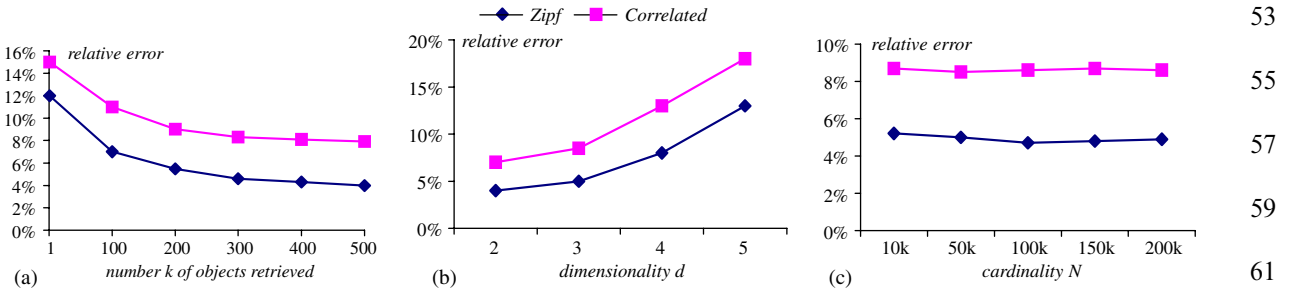

Fig. 13. Accuracy of cost estimation for BRS. (a) Error vs. k ($d = 3$, $N = 100$k), (b) error vs. d ($k = 250$, $N = 100$k), (c) error vs. N ($k = 250$, $d = 3$).

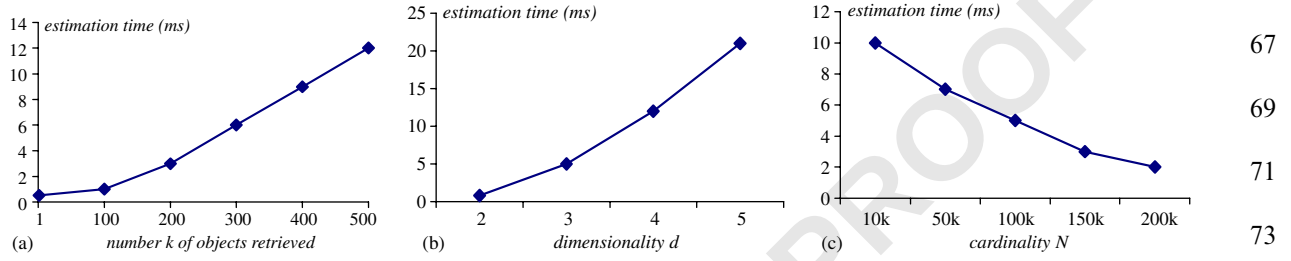

Fig. 14. Estimating time (*Zipf* data). (a) Time vs. k ($d = 3$, $N = 100$k), (b) time vs. d ($k = 250$, $N = 100$k), (c) time vs. N ($k = 250$, $d = 3$).

numbers of node accesses for the ith query ($1 \leqslant i \leqslant 200$); then the error is calculated as $(1/200) \cdot \sum_{i=1-200}(|act_i - est_i|/act_i)$.

Fig. 13 demonstrates the error rate for the same experiments as in Fig. 11. It is clear that our prediction is highly accurate, yielding maximum error 18%. Further, the error decreases with the number k of objects retrieved (Figs. 13a), increases with dimensionality d (Fig. 13b), and remains roughly the same for different dimensionalities N (Fig. 13c). Specifically, the improved accuracy for larger k happens because probabilistic prediction approaches generally perform better as the output cardinality increases. Further, due to the space constraint (500k bytes) on histograms, the histogram resolution decreases as d grows, leading to a coarser approximation of the data distribution, and hence, lower precision. The steady accuracy with respect to N can be explained by the stable performance of BRS illustrated in Figs. 11c and f, again confirming the close connection between estimation accuracy and output cardinality.

Fig. 14 shows the average time to produce the estimation for a single query on the *Zipf* datasets (the results for *correlated* are similar, and hence, omitted). As expected, the estimation overhead increases with k and d since in both cases a larger number of buckets need to be considered (in deciding the search region size and query cost, respectively). The overhead decreases for higher N, because larger cardinality results in a smaller search region, which in turn diminishes the number of buckets that intersect the region (and hence need to be examined). The longest time required is around 20 ms, indicating that the proposed method can be efficiently integrated in practical query optimization. Note, however, that, as shown in Fig. 12, the cost of BRS in the presence of a buffer larger than 4% is negligible, rendering query optimization for this case trivial.

6.1.3. *Effect of space reduction*

The next set of experiments aims at studying the benefits of the algorithm in Section 4.4 for reducing the R-tree size, in case that k is no larger than a constant K. Fig. 15a shows the space saving (measured as a percentage over the database size) for various values of K, using 3D datasets with 100k points. Note that, for $K = 1$, our method eliminates 99% (95%) of the *Zipf* (*correlated*) dataset (in other words, only 1% (5%) of the records may ever be retrieved by a top-1 query). As expected, the saving diminishes for larger K, but nevertheless, the reduced *Zipf* (*correlated*) R-tree is only around 10% (20%) of the original size, even for answering top-500 queries. The space saving is generally

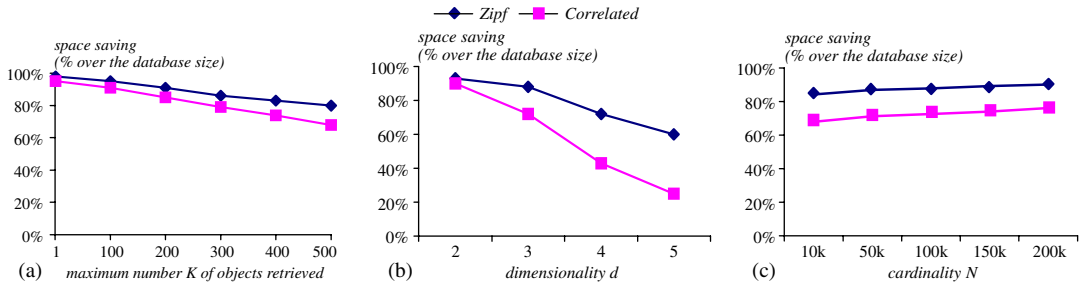

Fig. 15. Percentage of reduced space over the dataset. (a) Saving vs. K ($d = 3$, $N = 100$k), (b) saving vs. d ($K = 250$, $N = 100$k), (c) saving vs. N ($K = 250$, $d = 3$).

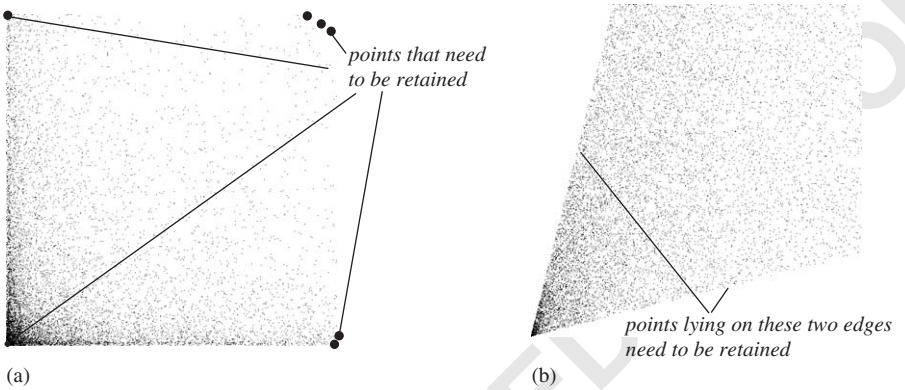

Fig. 16. Points retained after the space reduction (a) *Zipf*, (b) correlated.

smaller for correlated data. To explain this, consider the application of the algorithm to the 2D datasets in Fig. 16 for top-1 retrieval. The set of points that needs to be stored includes all points in the four skylines, viewed from each corner of the data space, respectively. For *Zipf* (where all the dimensions are independent), the skylines consist of points close to the corners (illustrated using enlarged dots in Fig. 16a). For *correlated*, in addition to points around the corners, the skylines also contain the points on the two indicated edges; hence, more data points must be retained than for *Zipf* data.

Fig. 15b demonstrates the space saving as a function of dimensionality d, setting the other parameters to their median values. Less space can be saved for larger d (up to 20%) because (i) the chance that a point dominates another, decreases in higher dimensionality [15], and (ii) the number of possible monotone-direction combinations increases exponentially with d. Fig. 15c plots the saving when N varies, and indicates a gradual increase of saving. This is most obvious for the *Zipf* dataset, where, no matter how large the dataset is, the number of points (close to the data space corners) that need to be retained is always limited. Similar observations hold for *correlated*.

6.1.4. Performance for non-linear monotone functions

The last experiments in this section evaluate the efficiency of BRS for non-linear functions. For this purpose we select three types of popular monotonic functions: (i) simple quadratic: $f(t) = \sum_{i=1-d}(w_i \cdot t.A_i^2)$, (ii) exponential: $f(t) = \sum_{i=1-d}(w_i \cdot e^{t.A_i})$, and (iii) logarithmic: $f(t) = \sum_{i=1-d}(w_i \cdot \ln(t.A_i))$. For each query, w_i is randomly generated in $[-1, 1]$. Fig. 17 shows the query cost as a function of k for datasets with $d = 3$, $N = 100$k. BRS is very efficient (less than 30 page accesses) for all types of functions. *Prefer* and the results by varying the other parameters are omitted since the diagrams are similar to those reported in Fig. 11.

6.2. Evaluation of complex ranked search

In the sequel, we study the performance of BRS for constrained top-k queries, group-by ranked search, and non-monotone preference functions.

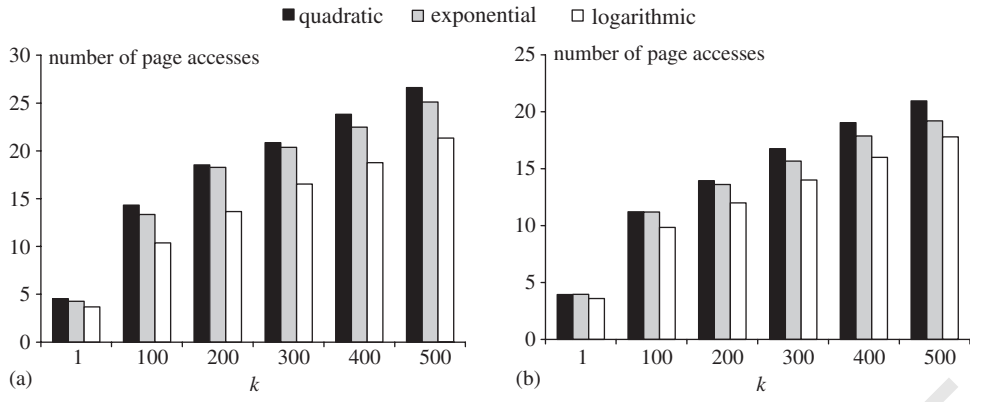

Fig. 17. Performance of BRS for non-linear functions ($d = 3$, $N = 100$k). (a) Skewed data, (b) correlated data.

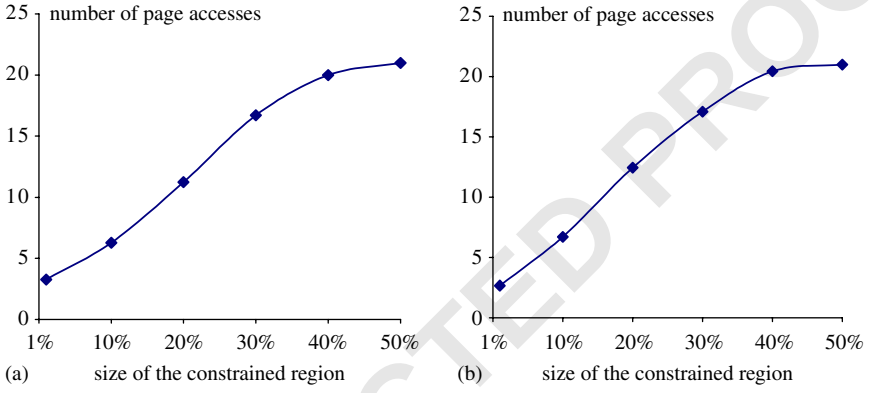

Fig. 18. Performance of BRS for constrained ranked queries ($k = 250$, $d = 3$, $N = 100$k). (a) *Zipf*, (b) correlated.

Onion and *Prefer* are not considered because they either result in exactly the same costs as for conventional ranked retrieval (shown in Section 6.1), or are not applicable at all (i.e., to non-monotone functions). We use 3D (*Zipf* and *correlated*) datasets with 100k records, and linear functions for queries with monotone preferences.

6.2.1. Cost of constrained ranked processing

We generate workloads of constrained queries in the same way as conventional top-k, except that each query is associated with an equal-sized constraint region, which is a d-dimensional box with identical extents along all dimensions. The position of a region follows the underlying data distribution (e.g., the region distribution for a *Zipf* dataset is also *Zipf*). Regions of different queries have distinct positions. Fig. 18 illustrates the performance of BRS (with the modifications described in Section 5.1) for retrieving 250 objects, as a function of the constrained size, represented using the length of a constraint region (e.g., for $d = 3$, a region with size 0.1 has a volume 0.1% of the data space). Interestingly, the query overhead initially increases (from very low values) when size is small, but stabilizes when the region is sufficiently large. This is because, for small windows, the performance of BRS depends mainly on the region size, while the cost converges to that of a normal ranked query (without any constraint) for sufficiently large windows.

6.2.2. Efficiency of group-by top-k search

To examine the efficiency of GBRS in Section 5.2, we create 3D datasets as follows. First, a 2D dataset with 100k points is generated following the *Zipf* or *correlated* distribution. Then, each point is associated with a "group id", an integer selected randomly in [1, *gnum*], where *gnum* is the total number of groups. We compare GBRS with the alternative approach that executes multiple (separate) constrained queries (one for each group). The

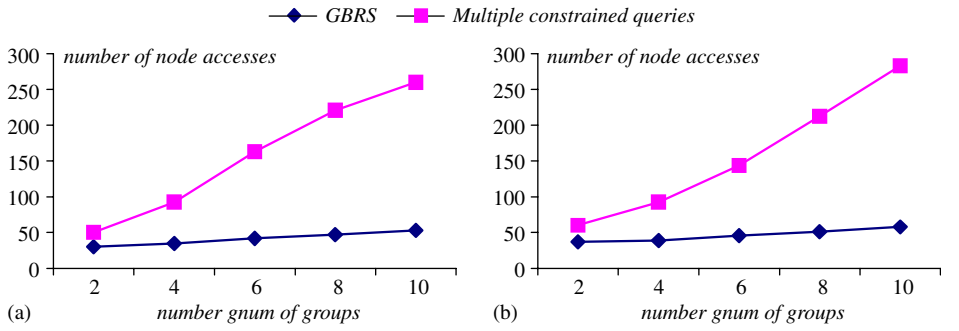

Fig. 19. Performance of GBRS ($k = 250$, $d = 3$, $N = 100$k). (a) *Zipf*, (b) correlated.

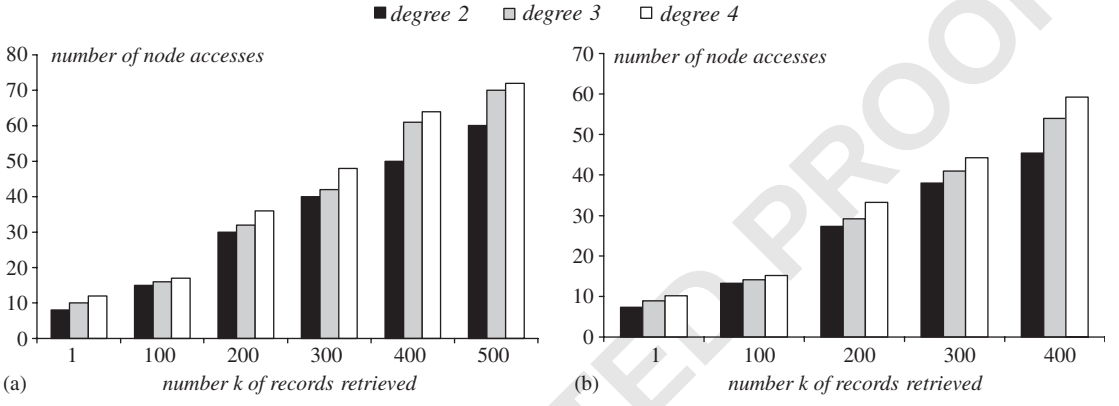

Fig. 20. Performance of BRS for non-monotone functions ($d = 3$, $N = 100$k). (a) *Zipf*, (b) correlated.

results are shown in Fig. 19, for both distributions, where it is clear that GBRS is significantly faster, and the difference increases with *gnum*.

6.2.3. Performance for non-monotone functions

Finally, we demonstrate the ability of BRS to support non-monotone functions. For this purpose, we experiment with polynomial preference functions in the form: $f(t) = \sum_{i=1}^{d} \sum_{j=1}^{deg} (c_{ij} \cdot t.A_i^j)$, where *deg* is the degree of the function varied from 2 to 4, and c_{ij} a constant randomly generated in $[-1, 1]$. It is worth mentioning that, for $deg = 2$, the resulting quadratic functions differ from the "simple quadratic" tested in Fig. 17 (which is always monotone). Exact solutions are returned for all queries. Using 3D datasets with cardinalities 100k, Fig. 20 plots the query cost as a function of k, for polynomials of degrees 2, 3, 4. BRS answers all queries with no more than 70 I/O accesses, or less than 10% of the total tree size (the trees for these datasets contain around 750 nodes).

To summarize, although consuming only a fraction of the space required by other methods, BRS processes queries significantly faster (usually by orders of magnitude). Further, it supports complex variations of ranked retrieval beyond the scope of the existing approaches, at no additional space overhead.

7. Conclusion

In spite of the importance of ranked queries in numerous applications, the existing solutions are not efficient because they either incur high processing overhead or consume excessive space. In this paper, we propose BRS, a novel approach that solves the problem using branch-and-bound algorithms. Specifically, BRS requires only a single off-the-shelf R-tree built on the ranking attributes of a given relation, and efficiently answers all top-k queries, regardless of (i) the number k of objects retrieved, (ii) the preference function used, and (iii) the additional search requirements (e.g., constrained

or not). As confirmed with extensive experiments, BRS significantly outperforms the existing alternatives on all aspects including query time, space overhead, and applicability.

Although our discussion focused on R-trees, BRS can be used with other multi-dimension access methods (e.g., SR-trees [27], X-trees [28], A-trees [29]), especially in high-dimensional spaces where the performance of R-trees degrades. Another direction for future work is to address the "approximate ranked query" that retrieves any k tuples whose scores are within a specified range from the best ones. Such tuples may be equally good in practice with the actual results (provided that the approximate range is small), but much faster to compute. Furthermore, it would be interesting to process top-k queries in data streaming environments where the data are not known in advance. Instead, the goal is to compute and continuously maintain the best results as new tuples arrive and the old ones are deleted or expire.

Acknowledgments

Yufei Tao was supported by SRG grant, number 7001843, from the City University of Hong Kong. Dimitris Papadias was supported by grant HKUST 6180/03E, from Hong Kong RGC.

References

[1] Y. Chang, L. Bergman, V. Castelli, C. Li, M. Lo, J. Smith. The onion technique: indexing for linear optimization queries, ACM SIGMOD, 2000.

[2] V. Hristidis, N. Koudas, Y. Papakonstantinou. PREFER: a system for the efficient execution of multi-parametric ranked queries, ACM SIGMOD, 2001.

[3] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, D. Srivastava. Ranked join indices, ICDE, 2003.

[4] V. Hristidis, Y. Papakonstantinou, Algorithms and applications for answering ranked queries using ranked views, VLDB J. 13 (1) (2004) 49–70.

[5] A. Guttman, R-trees: a dynamic index structure for spatial searching, ACM SIGMOD, 1984.

[6] N. Beckmann, H. Kriegel, R. Schneider, B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles, ACM SIGMOD, 1990.

[7] N. Bruno, S. Chaudhuri, L. Gravano, Top-k selection queries over relational databases: mapping strategies and performance evaluation, TODS 27 (2) (2002) 153–187.

[8] N. Bruno, L. Gravano, A. Marian, Evaluating top-k queries over web-accessible databases, ICDE, 2002.

[9] I. Ilyas, W. Aref, A. Elmagarmid, Joining ranked inputs in practice, VLDB, 2002.

[10] K. Chang, S.-W. Hwang, Minimal probing: supporting expensive predicates for top-k queries, SIGMOD, 2002.

[11] N. Roussopoulos, S. Kelly, F. Vincent, Nearest neighbor queries, ACM SIGMOD, 1995.

[12] G. Hjaltason, H. Samet, Distance browsing in spatial databases, ACM TODS 24 (2) (1999) 265–318.

[13] C. Böhm, H. Kriegel, Determining the convex hull in large multidimensional databases, DAWAK, 2001.

[14] D. Kossmann, F. Ramsak, S. Rost, Shooting stars in the sky: an online algorithm for skyline queries, VLDB, 2002.

[15] D. Papadias, Y. Tao, G. Fu, B. Seeger, An optimal and progressive algorithm for skyline queries, ACM SIGMOD, 2003.

[16] R. Benetis, C. Jensen, G. Karciauskas, S. Saltenis, Nearest neighbor and reverse nearest neighbor queries for moving objects, IDEAS, 2002.

[17] S. Berchtold, C. Böhm, D. Keim, H. Kriegel, A cost model for nearest neighbor search in high-dimensional data spaces, ACM PODS, 1997.

[18] A. Papadopoulos, Y. Manolopoulos, Performance of nearest neighbor queries in R-trees, ICDT, 1997.

[19] C. Boehm, A cost model for query processing in high dimensional data spaces, ACM TODS 25 (2) (2000) 129–178.

[20] Y. Theodoridis, T. Sellis, A model for the prediction of R-tree performance, ACM PODS, 1996.

[21] M. Muralikrishna, D. DeWitt, Equi-depth histograms for estimating selectivity factors for multi-dimensional queries, ACM SIGMOD, 1988.

[22] S. Acharya, V. Poosala, S. Ramaswamy, Selectivity estimation in spatial databases, ACM SIGMOD, 1999.

[23] N. Bruno, L. Gravano, S. Chaudhuri, STHoles: a workload aware multidimensional histogram, ACM SIGMOD, 2001.

[24] D. Gunopulos, G. Kollios, V. Tsotras, C. Domeniconi, Approximate multi-dimensional aggregate range queries over real attributes, ACM SIGMOD, 2000.

[25] W. Press, B. Flannery, S. Teukolsky, W. Vetterling, Numerical Recipes in C, Cambridge University Press, Cambridge, ISBN 0-521-75033-4, 2002.

[26] G. Zipf, Human Behaviour and the Principle of Least Effort: An Introduction to Human Ecology, Addison-Wesley, Reading, MA, 1949.

[27] N. Katayama, S. Satoh, The SR-tree: an index structure for high-dimensional nearest neighbor queries, ACM SIGMOD, 1997.

[28] S. Berchtold, D. Keim, H.P. Kriegel, The X-tree: an index structure for high-dimensional data, VLDB, 1996.

[29] Y. Sakurai, M. Yoshikawa, S. Uemura, H. Kojima, The A-tree: an index structure for high-dimensional spaces using relative approximation, VLDB, 2000.