Top-*k* Search in Product Catalogues

Martin Šumák and Peter Gurský

Faculty of science, University of Pavol Jozef Šafárik
Jesenná 5, 040 01 Košice, Slovakia `martin.sumak@student.upjs.sk`, `peter.gursky@upjs.sk`

Abstract. In the era of huge datasets, the top-*k* search becomes an effective way to decrease the search time of top-*k* objects. The original top-*k* search requires a monotone combination function and lists of objects ordered by attribute values. Our approach of the top-*k* search is motivated by complex user preferences over product catalogues. Such user preferences are composed of the local user preferences of the attributes' values (user defined arbitrary fuzzy functions, one for each attribute) and a user defined monotone combination function. This paper compares two different approaches of the top-*k* search for this type of non-monotone query. The first approach uses several B+trees, one for each attribute, and it is based on ordered lists. The second approach is new for this type of query and requires an R-tree index.

1 Introduction

The top-*k* search becomes more popular with increasing datasets sizes. Users are usually interested in a few best objects rather than a large list of objects as a result. Top-*k* algorithms usually follow two main goals. First, they minimize the number of source data to be processed. Second, the algorithms try to minimize the number of accesses to the sources and the computation time as well.

Our research in the area of the top-*k* search is motivated by product catalogue search. Users of the common e-catalogues have usually limited options of a preference specification. Typically, the only way to simplify product selection is to order the products by a single attribute. Sometimes, a user can restrict the object set by an interval of attribute values.

Our goal is to enable usage of more complex user preferences. Complex user preferences allow user to specify his top-*k* objects more accurately. On the other hand, preferences should be easy to specify and understandable for common user. Our model of user preferences consists of local preferences to attribute values and a global monotone combination function.

User's local preference to the values in domain D_A of attribute A is represented by a fuzzy function f_A: $D_A \rightarrow [0; 1]$. The fuzzy function gives the value 1 for the attribute values the most preferred by user and the value 0 for the attribute values that the user does not accept. Values between 0 and 1 represent the rates of the user acceptance of the attribute value.

Example 1: Imagine a user searching for a cheap laptop with medium screen size. Such preferences can be expressed by fuzzy functions similar to the ones shown on figure 1. We can see that user accepts laptops cheaper than approximately 700 € with a screen size between 11" and 15,5". Moreover, we know that preferred screen sizes are between 12" and 14" and that the cheaper the laptop is the better preference it has.

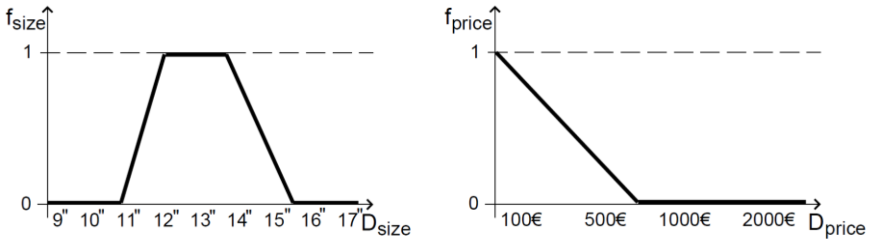

Fig.1. User's local preferences to the screen sizes and the laptops prices

The fuzzy functions can be specified explicitly by user using sliders [4]. Alternatively they can be learned from objects' ranking or by a click analysis on the catalogue website [12, 13].

The second part of user preferences is a global monotone combination function that combines the fuzzy values of the local preferences to the overall object value. If user considers m attributes in his preferences, the combination function C: $[0; 1]^m \rightarrow \mathbf{R}$ expresses the overall value of the user preferences to the object. The higher the overall value, the more preferred the object is.

Example 2: Let us consider the previous example. Our user can specify that the price is twice as important as the screen size. We can express this importance as weights w_{size}= 1 and w_{price}= 2. Then the combination function can be expressed as weighted sum of the fuzzy values:

$$C(f_{size}(size), f_{price}(price)) = 1*f_{size}(size) + 2*f_{price}(price)$$

The monotone combination function is quite difficult to express by a common user. Therefore the preferred approach uses a predefined combination function (sum, minimum and product) or it can be learned as well as the local preferences [12, 13]. A learned combination function has usually a form of a weighted sum or a set of monotone fuzzy rules [13].

Top-*k* search algorithms like *Threshold algorithm* (TA), algorithm *No Random Access* (NRA) [1] or *3 phased-No Random Access* (3P-NRA) can be used with user preferences mentioned above [4, 12].

The contribution of this paper is:

- new efficient algorithm for the top-*k* search based on R-tree (see section 4)
- experimental efficiency comparison of R-tree based algorithm to algorithms TA and 3P-NRA (see section 5)

In [2], algorithms like TA, NRA (or 3P-NRA) are used for searching over several types of attributes i.e. ordinal, metric, hierarchical, etc. Our new solution based on R-tree supports only ordinal attributes so far. Ordinal attributes in TA, NRA (or 3P-NRA) algorithms are handled by B+trees. In this paper we call these algorithms B+trees based approach.

2 Problem definition

For a given set **S** of objects we have to find k most preferred objects for the user. Each object $O \in \mathbf{S}$ has the same m attributes with real values $v_1(O),\ldots, v_m(O)$, where function v_i: $\mathbf{S} \rightarrow \mathbf{R}$ for all $i \in \{1, \ldots, m\}$. Input obtained from the user consists of m fuzzy functions $f_1,\ldots,f_m$ (or less if user does not consider all attributes) and a monotone combination function C. The overall value of object O is $C(f_1(v_1(O)),\ldots, f_m(v_m(O)))$. For example, if C is a weighted sum, user is expected to specify only nonnegative weights – one for each considered attribute. Then we have:

$$C(f_1(v_1(O)),\ldots,f_m(v_m(O))) = w_1*f_1(v_1(O)) + \ldots + w_m*f_m(v_m(O))$$

where $w_1,\ldots,w_m$ are the weights. The bigger the overall value, the more preferred the object O is to user. Output is a list of k objects from **S** ordered from the most preferred objects to the less preferred ones.

3 B+trees based approach

The original TA [1] and its derivates (NRA, 3P-NRA) require:

- m lists of pairs having form <object identifier, attribute value> previously ordered by attribute values
- a monotone combination function

In the TA, NRA, 3P-NRA algorithms, the ordered lists are read sequentially and the values from the lists are used as an input for a monotone combination function. Typically a top-k algorithm returns top-k objects after processing only a part of the lists.

Our approach requires the monotonicity of a combination function C, having fuzzy values of the local preferences as an input. In the naïve approach, the lists could be sorted according to the local preferences. Then, having the lists ordered by fuzzy values of the local preferences and a monotone combination function, we can use any TA-like algorithm to find top-k objects. Unfortunately, every user usually has different local preferences as well as different combination function. The sorting prior to every top-k search is highly ineffective – it is cheaper to do a table scan and compute the overall values for all objects in **S**.

Instead of ordering the lists, a B+tree can be prepared over each attribute of objects in **S** [2]. The main idea of this approach is that it does not need an ordered list to offer the ordered sorted access.

Example 3. Let us assume that the price and screen size attributes are indexed separately each in one B+tree. According to the fuzzy functions shown on figure 1, i.e. the user local preferences, the B+trees can be traversed to simulate the sorted accesses. The price tree is traversed from a minimal attribute value in ascending direction using pointers between leafs of the B+tree. In the case of the screen size two pointers are created, both starting at the some attribute value with the highest fuzzy value, i.e. 13" with fuzzy value 1. The first pointer traverses leafs of the B+tree in descending direction and the second one in ascending direction (see Fig.2).

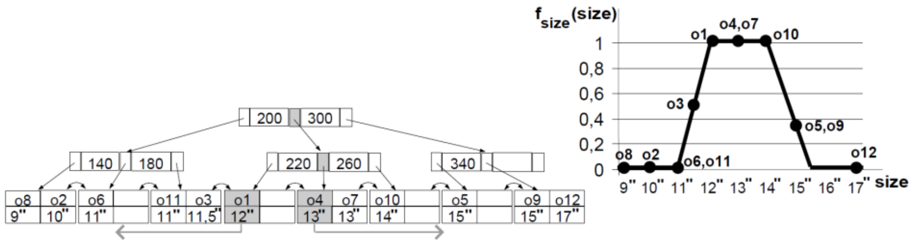

Fig.2. Traversing B+tree organized by attribute screen size during the sorted access simulation for TA-like algorithms

Having simple functions (partially linear), the points to identify the pointers and directions, i.e. the extremes of the fuzzy function, can be found very easily. The entry of a B+tree leaf with the highest fuzzy value can be identified by simple traversing from the root to the appropriate leaf.

At the beginning of the sorted access simulation [2] the entries with the local maximums of a fuzzy function are identified, based on which a set of pointers to leaf entries is obtained. Each pointer points to a leaf entry that should be returned as the next entry in the sorted access simulation. For each pointed entry a fuzzy value is computed. The entry with the highest fuzzy value is returned and its pointer is shifted to the neighbor entry of the corresponding direction.

The sorted access simulation allows us to use any TA-like algorithm. Instead of the original NRA, we prefer to use an improvement of the NRA algorithm called 3P-NRA with significantly better search time as shown in [2] especially in combination with the proposed heuristics. Hence we use 3P-NRA in our experiments in section 5.

4 Searching over R-tree

A contribution of this paper is a top-*k* search algorithm based on R-tree. As shown in the experiments, this approach is much more effective than B+trees based approach.

Since each object O can be represented by the point $p(O) = (v_1(O),\dots, v_m(O))$ in m-dimensional space (note that p: $\mathbf{S} \rightarrow \mathbf{R}^m$) the set $\mathbf{S}$ of objects can be stored in multidimensional R-tree index [8, 9]. The figure 3 shows an example of R-tree index used for point data. For searching top-*k* objects we adapted algorithm *Incremental Nearest Neighbor* (INN) [10] (also known as *sorted access*) commonly used for searching *k*-nearest neighbors. For formal description of our algorithm we first have to define some concepts.

Definition 1: *Point* P in m-dimensional space is vector $\mathrm{P} = (\mathrm{P}_1,\dots, \mathrm{P}_m) \in \mathbf{R}^m$.

Remark 1: Having $O \in \mathbf{S}$ and point P such that $\mathrm{P} = p(O)$ then $\mathrm{P}_i = v_i(O)$ for all $i \in \{1,\dots, m\}$.

Definition 2: *Rectangle* R (parallel with axes) is a pair of points $R = (\mathrm{L}, \mathrm{H})$ where $\mathrm{L}_i \leq \mathrm{H}_i$ for all $i \in \{1,\dots, m\}$.

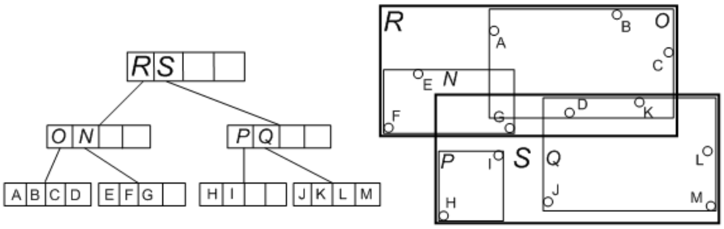

Fig.3. Example of R-tree for two-dimensional point data and capacity of nodes equal to 4

For the implementation and the tests the more effective variant R*-tree [9] was used. R*-tree has the same structure and properties as the R-tree, it differs only in the way of construction. However, the algorithm for k-nearest neighbors and for top-k search is the same for both of them.

Real values of objects' attributes come from different ranges (screen sizes of laptops are from 8" to 17" while the prices are from 400 € to 3000 €) and they are in different units. Therefore we will not build an R-tree index over real values of attributes but we will build it over linear normalized values where all values are within interval [0; 1]. The user defined local preferences (fuzzy functions) will be adjusted according to such normalized data. The normalization is necessary for effective utilization of R*-tree because of R*-tree aims e.g. to have as quadratic nodes as possible.

INN algorithm offers the objects within R-tree in the incremental way and in order from the nearest ones from a query point Q. In top-k search the input is not a single point but it consists of the user's preferences. Objects on the output should be ordered from best for the user. This can be easily achieved by ordering the priority queue within INN algorithm by some other value – not by minimal distance from Q but by maximal overall value (using combination function). For a node **N** we have to compute the maximal possible overall value for any object in a sub-tree rooted by the node **N**. For this purpose we define function h.

Definition 3: Function h: $\mathbf{S} \cup \mathbf{V} \rightarrow \mathbf{R}$, where **V** is a set of nodes of R-tree, is defined as follows: $h(\mathrm{E}) = C(y_1,\ldots, y_m)$ where for all $i \in \{1, \ldots, m\}$ $y_i = f_i(v_i(\mathrm{E}))$ if $\mathrm{E} \in \mathbf{S}$ or $y_i = max\{f_i(x)\colon \mathrm{L}_i \leq x \leq \mathrm{H}_i\}$ if $\mathrm{E} \in \mathbf{V}$ and (L, H) is minimal bounding rectangle of E.

Even though user's preferences can be specified by arbitrary fuzzy functions, nevertheless it must be possible to compute the maximum of the fuzzy function on an arbitrary interval and also to compute a value in any permissible x. We prefer to work with the partially linear fuzzy functions. They are easy to define by users using sliders in a graphic interface and they are also simple in computations.

The figure 4 shows a graphical representation of an example of function h. The value $h(O)$ of object O is greater than the value $h(\mathbf{N})$ of node **N**. Object O is therefore better for user than any possible object in a sub-tree of node **N** (any point in rectangle (L, H)).

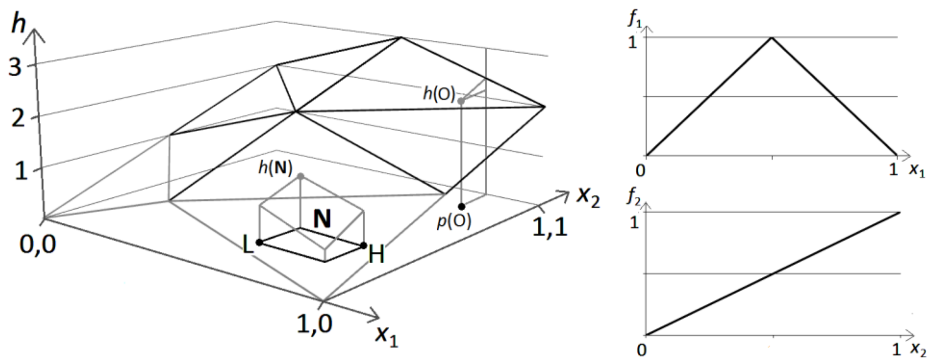

Fig.4. Graphical representation of the function $h(E) = C(y_1, y_2) = y_1 + 2 * y_2$ over normalized data having two attributes of objects with user-defined fuzzy functions f_1 and f_2 on the right

Algorithm preferential top-*k* search over R-tree:

```
Input: R-tree index of objects from S, fuzzy functions
       $f_1, …, f_m$, combination function $C$ and number $k$
Output: ordered list of $k$ objects with the highest value
        of the $h$ function
1. pqueue = empty priority queue ordered by the value of
   the $h$ function of its elements in descending order
2. result = empty list of objects
3. add the root node of the R-tree in the empty pqueue
4. while the result does not contain $k$ objects do
       a. let E be the first element of the pqueue, remove
          E from the pqueue
       b. if E is an inner node then add all its child
          nodes to the pqueue
       c. if E is a leaf node then add all its objects to
          the pqueue
       d. if E is an object then add object E to the end
          of the result
5. return result
```

Algorithm starts with the root node as the only element in the priority queue. At the beginning all the objects are taken into account (root contains all the objects in its sub-tree). The idea of the algorithm is to remove the node at the top of the priority queue, insert all its child nodes instead (or objects if the node was a leaf) with respect to ordering. This is repeated until some object appears on the top of the priority queue. Then the object is put to the end of the result list (the next best object to user). Priority queue ensures that on the top there is an element (i.e. a node or an object) with the maximal value of the function *h* among all elements in the priority queue. Moreover the priority queue can be organized to prefer object to a node with the equal value of function *h*. If the element on the top of the priority queue is a node, then its sub-tree can contain an object with higher value of function h than any other object present in the priority queue. On the other side, if there is an object on the top of the priority queue, its value of function h is higher than or equal to the value of function h of any other object in the priority queue and also of any object in sub-trees of nodes in the priority queue. Hence the first object appeared on the top of the priority queue is the best of all objects. The second object appeared on the top of the priority queue is the best of all remaining objects (i.e. the second best of all objects) and so on.

Inserting all child nodes to the priority queue requires the data (rectangles of child nodes) from the node itself only. Therefore it is possible to store only pointers to nodes in the priority queue for better efficiency and load real nodes when the information about their child nodes or objects is needed.

5 Tests and results

In the tests the time and number of IOs of the following three algorithms are compared: TA, 3P-NRA and the *preferential top-k search over R-tree*. Since R*-tree is the most efficient variant of R-tree, we used it in all the following tests. The utilization of R*-tree nodes was within the range from 30 to 90. In the tests we used three different distributions of artificial random data: Gaussian, exponential and uniform. We used sets of 100 000 and 1000 000 objects. The combination function was weighted sum where weights were chosen randomly from interval [1; 5]. Fuzzy functions for the evaluation of objects were chosen also randomly from 4 types (familiarly called: ascending, descending, hill, valley) used in [21]. All the tests were done on computer with 2GB of RAM, 2core Intel Centrino processor and SSD hard disk. The purpose of the tests was to reveal the influence of the data distribution, user preferences or the number of considered attributes on time and number of IOs. Presented values are the average values of 5 identical tests.

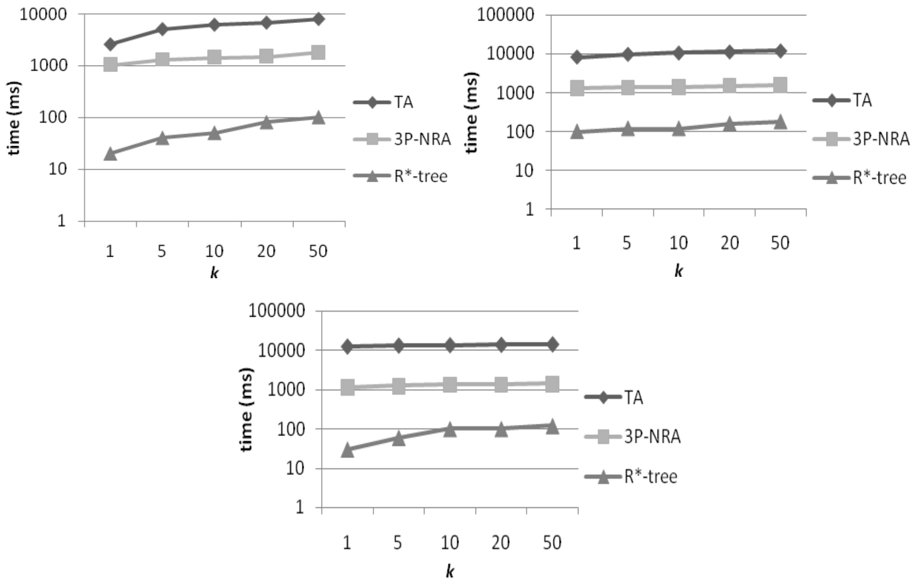

Fig.5. Search time of the top-1, 5, 10, 20, 50 objects in ms over 100 000 random objects with 10 attributes of exponential (top-left), Gaussian (top-right) and uniform (bottom) distribution with all 10 attributes in a query

First set of charts (figure 5) describes the search time of top-*k* objects. We can observe significant speedup of searching when data are indexed in R*-tree and the query uses all 10 attributes of objects. The dependence of number of IOs copies the time dependence for all three algorithms.

The question is whether the R*-tree based algorithm will be so fast if the query would contain only some of all indexed attributes. TA and 3P-NRA algorithms can be considered faster trivially (because they read fewer ordered lists). The R*-tree based algorithm always searches over the R*-tree structure containing all the attributes independently of the user query.

Next set of charts (figures 6, 7) describes the search time of the top-10 and the top-50 objects of 10 attributes with less than 10 attributes used in a query.

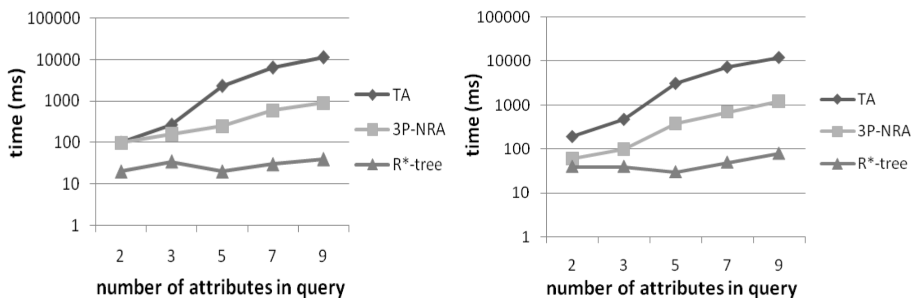

Fig.6. Search time of the top-10 (left) and the top-50 (right) objects in ms over 100 000 random objects of 10 attributes with uniform distribution with 2, 3, 5, 7, 9 attributes in a query

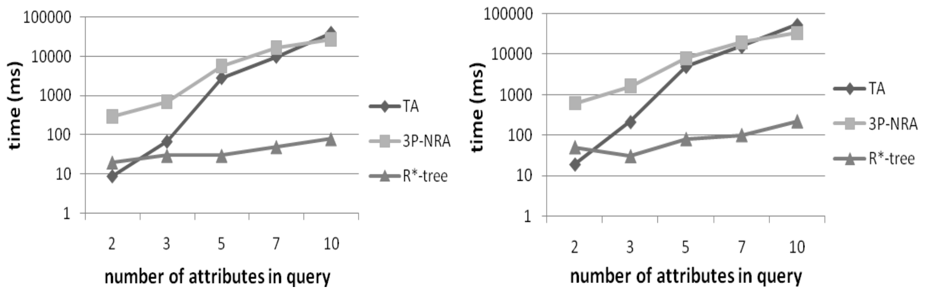

Fig.7. Search time of the top-10 (left) and the top-50 (right) objects in ms over 1 000 000 random objects of 10 attributes with exponential distribution with 2, 3, 5, 7, 10 attributes in a query

As we can see on the charts above the R*-tree based algorithm is fast even for a small number of attributes in the query. In farther tests we observe that this property of R*-tree based algorithm is preserved also for a bigger set of objects (1 000 000), more attributes (20) and also for other distributions (exponential, Gaussian).

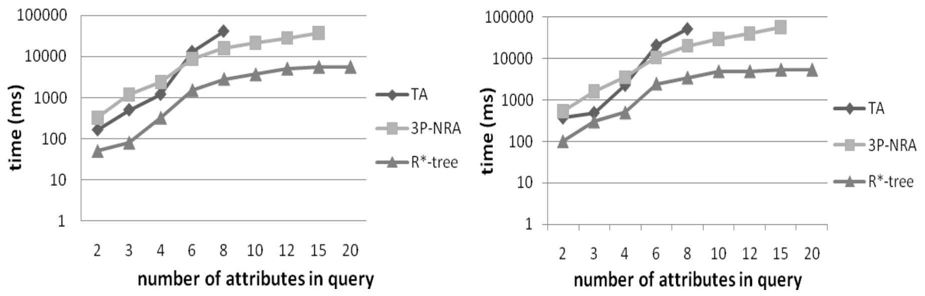

Fig.8. Search time of the top-10 (left) and the top-50 (right) objects in ms over 1 000 000 random objects of 20 attributes with Gaussian distribution with 2, 3, 4, 6, 8, 10, 12, 15, 20 attributes in a query

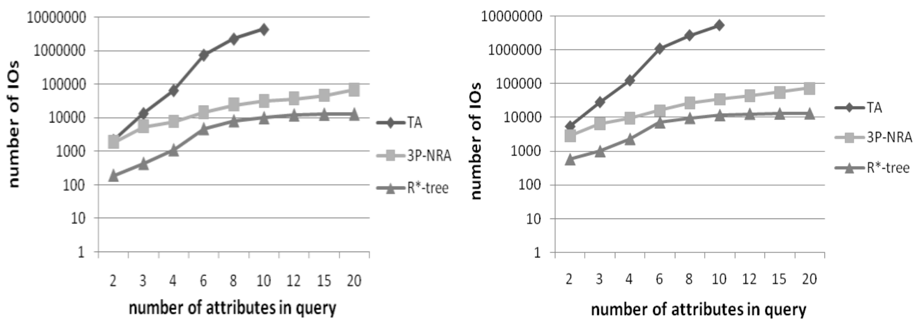

Fig.9. Number of IOs in the search of the top-10 (left) and the top-20 (right) objects over 1 000 000 random objects of 20 attributes with Gaussian distribution with 2, 3, 4, 6, 8, 10, 12, 15, 20 attributes in a query

The tests show that R*-tree based approach is much faster than the B+trees based approach. The remarkable speedup is reached by decreasing the number of IOs – from several B+trees to one R*-tree.

6 Related work

This paper studies the top-*k* search for complex user preferences composed of arbitrary local preferences and a monotone combination function. Local preferences and combination function together generate a non-monotone function giving the overall value.

The area of the top-*k* search was extensively researched in the last 7-11 years. The main stream of the top-*k* search algorithms [1, 14, 15, 16, 17], we call them TA-like algorithms, considers having a monotone combination function and possibly distributed ordered lists for each attribute.

The query composed of local preferences and monotone combination function was introduced in [2]. As shown in section 3 the simulation of sorted access to the lists allows us to use TA-like algorithms.

A special branch of top-*k* algorithms is considered to be embedded in RDBMS [18, 19, 20]. These approaches are concerned with augmenting the query optimizer to consider rank-joins during plan evaluation. Optimization can be effective especially in the case of very selective attributes. The rank-join algorithms require ordered data on input similarly to TA-like algorithms. The way of ordering is not considered or the ordering of attribute domains is used implicitly.

The approaches in [5, 6] do the top-*k* search with an arbitrary (also non-monotone) query analyzing the aggregation function with mathematical methods. If a ranking function analysis over any domain sub-region is possible (to find the maximum and possibly recognize monotonicity), according to authors this approach is able to find the top-*k* objects in an effective way. In our opinion this analysis is rather difficult to be done for an arbitrary function. However, we could not proceed in further analysis of this approach because the source codes or a deeper description of the algorithms are not available.

R-tree is designed especially for multidimensional range queries and similarity search (nearest neighbor queries). Moreover R-tree can be effectively used for top-*k* queries. The idea of using R-tree for the top-*k* search is briefly presented in [7] where authors compare the top-*k* search over R-tree and over their new index called "Ranked Join Index". They consider only a simple weighted sum as a top-*k* query. Note that our top-*k* query, consisting of m fuzzy functions $f_1,\dots,f_m$ and a monotone combination function C, is more complex. The composition of fuzzy functions and a combination function is not monotone in general, thus our approach has a higher expressive power. Since the "Ranked Join Index" requires a monotone top-*k* query, it cannot be used with our query.

7 Conclusions and future work

Unlike separate ordered lists in the original TA-like algorithms our approach uses single R-tree or R*-tree. Since R-tree is a multidimensional index, each attribute can be represented as one dimension. A disadvantage in comparison to the original TA-like algorithms is that we must know the values of all objects' attributes and all the data must be stored locally in one structure. We must have the whole R-tree prepared prior to the query (we emphasize that we have to prepare the R*-tree only once, not prior to each query). We assume that a condition of locally accessible data prepared in advance is performable for almost all attributes. Nevertheless in the case of dynamic, remote or not ordinal attributes, the TA-like algorithms and the R-tree based approach can be combined. Since algorithm for the top-*k* search over R-tree can be used to produce an ordered list, we can carry out the top-*k* search using a TA-like algorithm. One ordered list for a TA-like algorithm can be simulated by the top-*k* search algorithm over R-tree and several (not so many) ordered lists can be obtained in other way, i.e. simulation over remote B+tree. In this case we can dramatically decrease the number of ordered lists read by a TA-like algorithm and consequently the overall time of the top-k search as well. This is the idea of our future work.

Acknowledgement

This work was partially supported by VEGA 1/0131/09 and by the Agency of the Slovak Ministry of Education for the Structural Funds of the EU, under project ITMS: 26220120007.

References

1. R. Fagin, A. Lotem, M. Naor, "Optimal Aggregation Algorithms for Middleware", in Proc. ACM PODS, 2001
2. P. Gurský, R. Pázman, P. Vojtáš, "On supporting wide range of attribute types for top-k search", *Computing and Informatics*, Vol. 28, no. 4, 2009, ISSN 1335-9150, p. 483-513..

3. T. Horváth, "A Model of User Preference Learning for Content-Based Recommender Systems", *Computing and informatics* Vol. 28 (2009), No. 4: SAV, Slovakia, 2009, ISSN 1335-9150, p: 453-481.
4. V. Vaneková, P. Vojtáš, "Fuzziness as a Model of User Preference in Semantic Web Search", in Proc. IFSA-EUSFLAT 2009, pp. 998-1003
5. Z. Zhang, S. Hwang, K. Chang, M. Wang, C. Lang, Y. Chang, "Boolean + Ranking: Querying a Database by K-Constrained Optimization," In SIGMOD 2006.
6. D. Xin, J. Han, K. Chang, "Progressive and Selective Merge: Computing Top-K with Ad-Hoc Ranking Functions," in SIGMOD 2007.
7. P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, D. Srivastava, "Ranked Join Indices", ICDE, pp.277, 2003
8. A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", SIGMOD Conference 1984
9. N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles", SIGMOD Conference 1990: 322-331
10. G. R. Hjaltason , H. Samet, "Distance browsing in spatial databases", ACM Transactions on Database Systems (TODS), v.24 n.2, p.265-318, June 1999
11. G. R. Hjaltason, H. Samet, "Ranking in Spatial Databases", Proceedings of the 4th International Symposium on Advances in Spatial Databases, p.83-95, August 06-09, 1995
12. P. Gurský, T. Horváth, R. Novotný, V. Vaneková, P. Vojtáš, "UPRE: User preference based search system", In IEEE/WIC/ACM Web Inteligence (2006)
13. T. Horváth, P. Vojtáš, "Ordinal Classification with Monotonicity Constraints", In Proc. 6th Industrial Conference on Data Mining ICDM (2006)
14. R. Akbarinia, E. Pacitti, P. Valduriez, "Best Position Algorithms for Top-k Queries", in VLDB, 2007.
15. H. Bast, D.Majumdar, R. Schenkel, M. Theobald, G. Weikum, "IOTop-k: Index-Access Optimized Top-k Query Processing". In VLDB, 2006.
16. W. Balke, U. Güntzer, "Multi-Objective Query Processing for Database Systems", in VLDB, 2004.
17. U. Güntzer, W. Balke, W. Kiessling, "Towards Efficient Multi-Feature Queries in Heterogeneous Enviroments", in ITCC, 2001.
18. F. Ilyas, W. Aref, A. Elmagarmid, "Supporting Top-k Join Queries in Relational Database", in VLDB, 2003.
19. F. Ilyas, R. Shah, W.G. Aref, J. S. Vitter, A.K. Elmagarmid, "Rank-Aware Query Optimization", in SIGMOD, 2004.
20. C. Li, K. Chang, I. F. Ilyas, S. Song, "RankSQL: Query Algebra and Optimization for Relational Top-k Queries", in SIGMOD, 2005.
21. P. Gurský, "Towards better semantics in the multifeature querying", proceedings of Dateso 2006, ISBN 80-248-1025-5, pages 63-73, 2006