Top-k Search Over Grid File

Martin Šumák and Peter Gurský

Institute of computer science, Faculty of Science, P. J. Šafárik University in Košice
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. Since we suppose locally accessible data only, the multidimensional indexes containing all attributes together seem to be more effective than a distribution of each attribute to a separate index. Therefore we introduce the top-k search algorithm over grid file – the multidimensional index not used for the top-k search yet. Grid file does not require computation with all attribute values together like R-tree, R*-tree (i.e. computation of area, perimeter) nor a metric like M-tree. Grid file can be used directly for indexing any type of attributes with natural ordering. Our experiments show that grid file, R-tree and R*-tree offer much better performance of the top-k search than separated B^+-trees and table scan.

1 Introduction

In our research we deal with the problem of searching top-k products in e-shops according to user preferences. Current e-shops typically provide fulltext search, menu of product domains, single attribute value specification and products sorted usually according to price or product name. We are not aware of any e-shop with more complex user preferences model e.g. a combination of selection techniques mentioned above.

Our model of user preferences [6] consists of preferences to values of several attributes in the form of fuzzy functions (see Figure 1) and a monotone combination function. Such complex user preference model approaches real life preferences and leads to more precise results than standard selection techniques. The preferences can be obtained implicitly by tracking user actions in the e-shop or explicitly by user specification. The top-k search with a query based on such preferences can be computed over different index structures – a set of B+-trees [5, 6], MDB-tree [12] and R-tree [13]. In this paper we introduce a top-k search algorithm over the next multidimensional index – Grid file.

In [12, 13] it was shown that the top-k search over multidimensional indexes (MDB-tree, R-tree, R*-tree) is faster over local data than over a combination of multiple indexes. The MDB-tree can hold all types of ordered attributes but the query cannot hold any subset of attributes. The R-tree structure allows a query to contain any subset of attributes but the metrics used in the R-tree requires having the numbered attributes only. The reason why we employed a grid file for the top-k search was the elimination of the limitations along with the preservation of the advantages of the mentioned indexes.

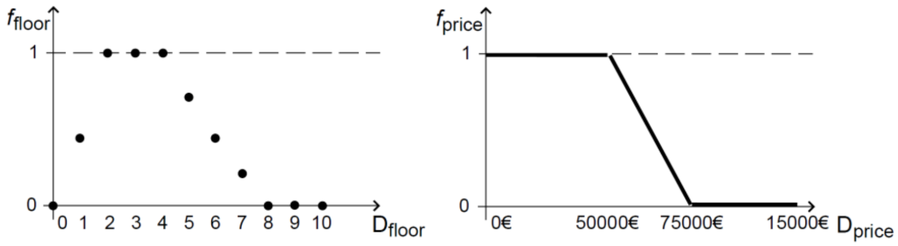

Fig.1. User's local preferences to a floor and a price of a flat

This paper is organized as follows. Section 2 presents the related work. Section 3 formalizes the problem of the top-*k* search over our user preference model. Section 4 presents the main contribution of this paper – the top-*k* search over grid file. Section 5 reports the experiments results comparing the top-*k* search performance over several index structures. Section 6 concludes this paper.

2 Related work

The top-*k* search was introduced by R. Fagin [4] as a problem of finding *k* best objects according to a monotone aggregation function over distributed ordered lists of attribute values. The original Threshold algorithm (TA) [4] has many improvements and modifications for the similar distributed environment, e. g. [1, 2, 4, 6]. We call them the TA-like algorithms.

The idea of the top-*k* search over data stored in several indexes was considered also for local data, especially inside a RDBMS, e. g. [10]. These approaches are concerned with augmenting the query optimizer to consider rank-joins (similar to TA) during a plan evaluation. Optimization can be effective especially in case of very selective attributes. The rank-join algorithm requires ordered data on input similarly to the middleware algorithms.

The idea of using R-tree for top-*k* search is already presented in [14] where the algorithm incremental nearest neighbour is exploited for that purpose. Nevertheless, this approach does not offer a query as complex as we offer in our query model.

Originally in the top-*k* query, the simple monotone aggregation function was considered only [4]. The query composed of local preferences and monotone combination function (resulting in non-monotone aggregation function) was introduced in [5]. In [6] it was shown that the simulations of sorted accesses using separated indexes for each attribute allow using TA-like algorithms.

The algorithm in [16] does a top-*k* search with an arbitrary non-monotone query analyzing the aggregation function with numerical methods. The algorithm supposes that the numerical methods can analyze any aggregation function over any domain sub-region (to find the maximum and possibly recognize monotonicity). In our opinion this analysis is rather difficult to do in a reasonable time. Note that this approach uses multiple indexes.

The grid file was introduced in [11]. In many papers the grid file is considered to be a dynamic index with a directory structure mapping grid windows to the disk pages

[7, 11, 15]. In our pilot grid file implementation we considered static data only, thus we made some simplifications (see Section 4). First, we made the numbering of grid windows that can substitute the presence of directory structure and dramatically decrease the number of accesses, thus making most objects accessible in 1 I/O. Second, unlike the original grid file we employed the overflow pages to avoid dense grid structure with many empty windows over possibly skew data. We analyzed several bulk loading techniques [3, 8, 9]. The STR algorithm [9] has the best results for our top-k search.

3 Top-k search problem definition

For a given set $\boldsymbol{S}$ of objects we have to find k most preferred objects for the user. Each object $O \in \boldsymbol{S}$ has the same m attributes with values $v_1(O), \dots, v_m(O)$ from attribute domains $\boldsymbol{A}_1, \dots, \boldsymbol{A}_m$ respectively (i.e. $v_i: \boldsymbol{S} \to \boldsymbol{A}_i$ for all $i \in \{1, \dots, m\}$). Query, i.e. input obtained from the user, consists of m fuzzy functions $f_1, \dots, f_m$ (or less if user does not consider all attributes) and a monotone combination function C. The overall value of object O is $C(f_1(v_1(O)), \dots, f_m(v_m(O)))$. For example, if C is a weighted sum, user is expected to specify only nonnegative weights – one for each considered attribute to specify a non-descending combination function. Then we have:

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

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

4 Grid file

The grid file [9] is an index structure for multidimensional points designed to store the data on disk pages. Grid file is based on slicing space in each dimension, i. e. an attribute domain, to get a multidimensional grid. For the formal description we introduce the following notation. Partition $P = (U_{1,1}, \dots, U_{1,n_1}) \times \dots \times (U_{m,1}, \dots, U_{m,n_m})$ is determined by a sequence of intervals in each dimension. Each (i-th) sequence consists of disjunctive intervals such that $U_{i,1} \cup \dots \cup U_{i,n_i} = \boldsymbol{A}_i$. Picking one interval in each dimension specifies a window $U_{1,j_1} \times \dots \times U_{m,j_m}$. Each window is mapped to one data page on disk (these pages are called primary pages). The grid file contains as many primary pages as windows. All data pages have the same fixed size (i.e. the same fixed capacity). Overfilled windows are handled by creating a linked chain of overflow pages.

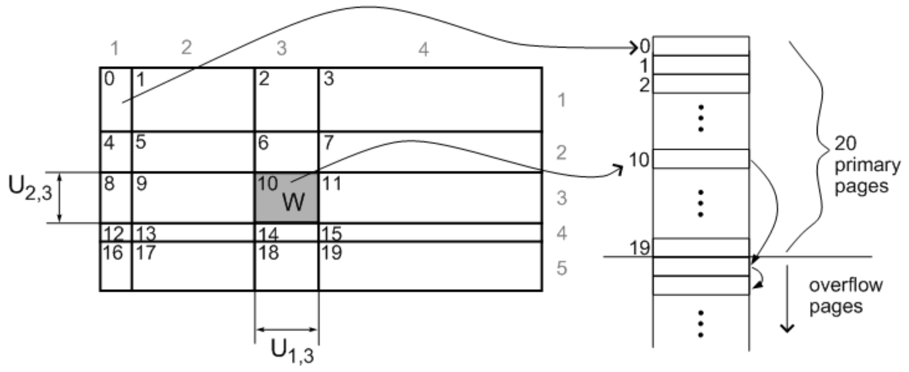

Fig. 2. An example of a two-dimensional grid with 20 windows and the overfilled window W. Window W is determined by the third interval in both dimensions.

On the other hand, the grid file may contain empty windows. Each empty window refers to an empty primary page. Reading empty pages (e. g. during query evaluation) is avoided by a set of numbers of non-empty windows held in memory. The windows are numbered in a fashion shown on Figure 2. Since we have a static grid, the mapping between a window and its number is easy to compute without the need of a directory structure. The extension of this idea to more dimensions is straightforward.

We create the grid file with bulk loading algorithm STR [9]. Page capacity and the number of objects in $\boldsymbol{S}$ determine the final number of windows (primary pages). In a m-dimensional space the m-th root of the number of windows determines the number of intervals in each dimension. Each dimension is partitioned into intervals with the same number of objects falling in them. Since real data is rarely distributed uniformly we reduce the number of overfilled windows by increasing the number of windows by the multiplication with appropriate filling factor (we use filling factor 1.3). After the bulk loading of input data we are not restricted from adding more objects – it simply leads to higher utilization of pages and possibly to some new overflow pages.

4.1 Top-*k* search over grid file

A contribution of this paper is a top-k search algorithm over grid file. As shown in the experiments, this approach is much more effective than B$^+$-trees based approach and it is comparable with the top-k search over R-tree [13].

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 $p\colon \boldsymbol{S} \to \boldsymbol{A}_1 \times \dots \times \boldsymbol{A}_m$ is the function mapping objects to m-dimensional points), the set $\boldsymbol{S}$ of objects can be stored in multidimensional index such as grid file [8, 9]. Grid file does not require attribute domains to be sets of numbers. It can handle different types of attributes at once. For example the first attribute can be price represented by decimal numbers while the second attribute can be a manufacturer represented by strings (with alphabetical ordering). Grid file treats attribute domains separately therefore they are not required to have any common property. Attribute domain just needs to be an ordered set.

For searching top-k objects over a grid file we developed algorithm similar to the breadth first search in graphs. For formal description of our algorithm some concepts need to be defined first.

Definition 1: Point Z is m-dimensional vector $Z = (Z_1, \dots, Z_m) \in \boldsymbol{A}_1 \times \dots \times \boldsymbol{A}_m$.

Having $O \in \boldsymbol{S}$ and point Z such that $Z = p(O)$ then $Z_i = v_i(O)$ for all $i \in \{1, \dots, m\}$.

Definition 2: A window is defined as an m-tuple of intervals $(U_1, \dots, U_m)$ where U_i is an interval within $\boldsymbol{A}_i$ for all $i \in \{1, \dots, m\}$.

Object O belongs to a window $W = (U_1, \dots, U_m)$ if $v_i(O) \in U_i$ for all $i \in \{1, \dots, m\}$.

Definition 3: We say that window $V = (U_{1,i_1}, \dots, U_{m,i_m})$ is a neighbour to window $W = (U_{1,j_1}, \dots, U_{m,j_m})$ iff there is a $q \in \{1, \dots, m\}$ such that $|i_q - j_q| = 1$ and for all $r \in \{1, \dots, m\} \backslash \{q\}$ holds $i_r = j_r$.

Note that window in a 2-dimensional grid has at most 4 neighbour windows – top, bottom, left and right. Window in the corner of the gird has two neighbour windows.

For the top-k search over a grid file we need to know how to evaluate objects and also grid windows. For this purpose we define aggregation function h giving the overall value for an object and the maximal possible overall value for any object in a grid window.

Definition 4: Function $h: \boldsymbol{S} \cup \boldsymbol{W} \to \boldsymbol{R}$ where $\boldsymbol{W}$ is a set of windows of grid file is defined as follows: $h(E) = C(y_1, \dots, y_m)$ where

$y_i = f_i(v_i(E))$ for all $i \in \{1, \dots, m\}$ if $E \in \boldsymbol{S}$ OR

$y_i = max\{f_i(x): x \in U_i\}$ if $E = (U_1, \dots, U_m) \in \boldsymbol{W}$.

Lemma 1: If $W = (U_1, \dots, U_m)$ is a window and O is an object such that $p(O) \in U_1 \times \dots \times U_m$ (i.e. O belongs to a window W) then $h(O) \leq h(W)$.

Proof: From definition 4 we have $h(O) = C(f_1(v_1(O)), \dots, f_m(v_m(O)))$ and $h(W) = C(max\{f_1(x): x \in U_1\}, \dots, max\{f_m(x): x \in U_m\})$. Moreover for all $i \in \{1, \dots, m\}$ holds: $f_i(v_i(O)) \leq max\{f_i(x): x \in U_i\}$ because $v_i(O) \in U_i$. Since combination function C is monotone (non-descending in each parameter) we get $h(O) \leq h(W)$.

Lemma 2: If $h(O) \geq h(W)$ (where O is an object and W is a window) then for any object Q within W holds $h(O) \geq h(Q)$.

Proof: directly from Lemma 1 and the transitivity of relation $\geq$.

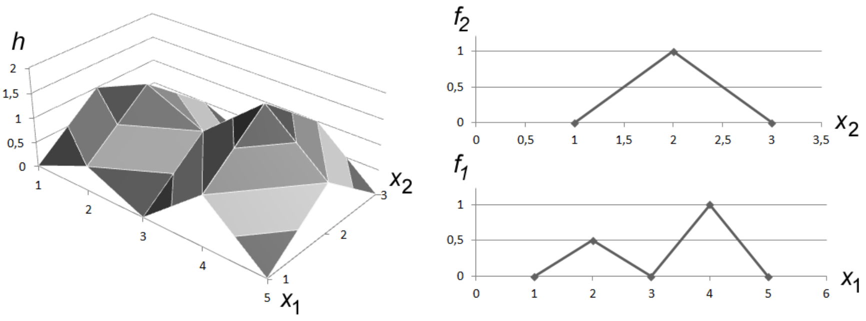

Fig. 3. Graphical representation of the aggregation function $h(E) = y_1 + y_2$ giving the overall value $f_1(x_1) + f_2(x_2)$ for a 2-dimensional data with user-defined fuzzy functions f_1 and f_2 on the right.

Preferential top-k search algorithm over grid file:

```
Input: grid file containing 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. queue = empty priority queue ordered by the value of
   the $h$ function of its elements in descending order
2. result = empty list of objects
3. for each local extreme of the function $h$ do
   a. choose arbitrary one window $W$ containing the
      extreme
   b. if queue does not contain $W$ then put $W$ into queue
      and label $W$ as visited window
4. while the result does not contain $k$ objects do
   a. let $E$ be the first element of the queue, remove $E$
      from the queue
   b. if $E$ is a window then
      i. add all objects within $E$ to the queue
      ii. add all not visited neighbour windows of $E$ to
          the queue and label them as visited windows
   c. if $E$ is an object then add it at the end of the
      result
5. return result
```

The estimation of time and space complexity can be reduced to an estimation of the number of visited windows, which is highly dependent on the grid partition, data distribution and query. The only estimation we can make are the lower and upper bounds, which is not very rewarding. In the best case we get the top-k objects after processing one window – the first one in the queue containing global extreme. In the worst case all the windows must be read – typically when a high k or low discriminating fuzzy functions or fuzzy functions containing many local extremes are obtained from user. Fuzzy functions with many local extremes are not common in a queries made by people.

Labeling visited windows can be realized by maintaining a set of numbers of visited windows starting with an empty set. Avoiding repetitive reading of visited windows is implemented the same way as avoiding reading of empty windows – by maintaining a set of window numbers in memory.

4.2 Correctness of the top-k search over grid file

The proof of correctness of presented algorithm can be reduced (without impact on generality) to the situation with just one local extreme of function h. If we prove that it works for the case of one local extreme then the generalization to more extremes can go as follows: we can prepare as many priority queues as windows with local extremes in step 3. Then in step 4.a we can pick the priority queue with the highest value of its first element and continue without any other changes. Using separated priority queues for each starting window picked in step 3 is not necessary. The same effect can be achieved by one priority queue managing windows of all local extremes because on the top there is always an element with the highest value from all top elements in imaginary separated priority queues.

Let us focus on one local extreme of the function h. First of all we will show that the value of the first element in priority queue is non-ascending. Using mathematic induction we will show that each time the first element is removed from priority queue, the new top element of the priority queue (i.e. in the next iteration of while cycle) has lower or equal value to the previous top element.

If the top element in the priority queue is an object then the condition holds trivially, since no new element is added into priority queue and the next top element was already in the priority queue in previous iteration.

Let's assume that there is a window at the top of the priority queue. We have to show that all new elements added into priority queue in steps 4.b.i and 4.b.ii have the overall value lower or equal to the window just removed from the top. For better imagination we will use the example on Figure 4.

The first induction step is as follows: at the beginning, the priority queue contains just one window W – the one containing a local extreme of function h. Window W is to be removed from the queue directly in the first iteration of while cycle (step 4). After that, the algorithm inserts the objects within window W and its neighbour windows A, B, C and D to the priority queue. In Lemma 1 we showed that objects belonging to window W have the overall value lower or equal to the overall value of window W. Trivially, the neighbour windows A, B, C, D do not have the overall value greater than window W because the algorithm started with window containing the only local extreme.

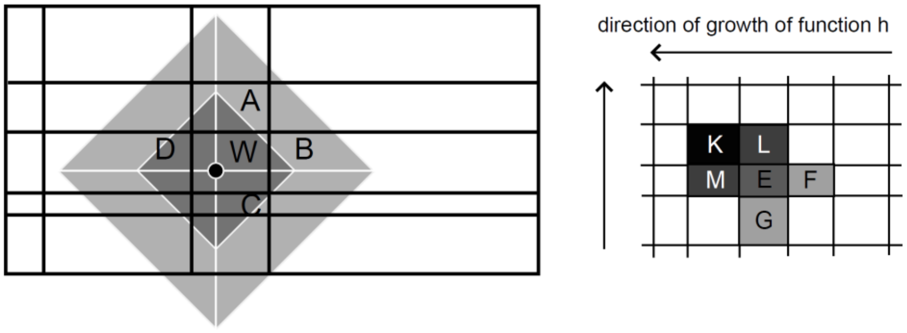

Fig. 4. On the left, there is a window W containing local extreme and its neighbour windows A, B, C and D. On the right, there is an example of the second inductive step with local extreme somewhere in left top corner. The darker shade means higher overall value.

For the second induction step we assume the following induction assumption: in each of previous iterations of while cycle, the overall value of the top element in the priority queue decreases or does not change. Let window E (Figure 4 on the right) be the first element in the queue. After removing the window E from the priority queue, the objects from E and not visited neighbour windows (L, M, F, G) are inserted into the priority queue. In each dimension there is one direction in which the respective fuzzy function is non-descending and the opposite direction oriented off the local extreme in which the fuzzy function is non-ascending. In the example on Figure 4 the local extreme is somewhere in the top left corner. Therefore we can trivially say that $h(K) \geq h(L)$, $h(K) \geq h(M)$, $h(L) \geq h(E)$, $h(M) \geq h(E)$, $h(E) \geq h(F)$ and $h(E) \geq h(G)$. We are left to show that windows L and M have been already visited and therefore they are not to be inserted to priority queue now. From the induction assumption we get that window E was added to priority queue as neighbour when either window L or M was removed from the top. Without impact on generality let us suppose that window L is the removed window. Since we know that window L has been already visited we are left to discuss window M. Since window K has higher value than window L and L is its neighbour, from the induction assumption we get that window K must have been visited before L. Moreover only top windows from the priority queue are processed. Therefore window K must have been processed before window L and window M must have been added into priority queue when window K was being processed. Hence windows L and M had been visited before window E was processed and are not inserted into the priority queue.

Although we described the second induction step on an example in a 2-dimensional space the generalization to more dimensions is straightforward. Even in 2D we can imagine a situation slightly different from the one drawn on Figure 4 on the right. Let us imagine the following change: $h(L) \leq h(E)$. In this situation window E has only one neighbour with higher overall value – window M. The discussion for this case is even simpler. From the induction assumption we know that window M must have been visited prior to window E.

Since value of the element at the top of the priority queue is non-ascending the first object that appears at the top is the best object of all. Each object which would appear in the priority queue later will be at most as good as any object in the result set. Since we look at all neighbour windows, processing the rest of the priority queue leads to acquiring all objects within grid file in order from the best to the worst.

Note that the presented grid file expansion strategy in the top-k search works correctly for our user preferences model, however it cannot be used for arbitrary aggregation function in general. For arbitrary aggregation function it requires a modification of the neighbour windows definition to two windows with a common point and it leads to exponentially more priority queue insertions according to number of dimensions than in the presented algorithm.

5 Experiments

Average time of top-k query evaluation is the basic measure we surveyed. We used a real data set containing approximately 27 000 flat or house advertisements in Slovakia having 6 attributes: price, area, floor, the highest floor of building, year of approbation and the number of rooms. Since the real data set was small we generated bigger pseudo real sets by generation of several similar objects for each one from the original set. This way we generated two sets, one with about 550 000 objects (the 20-multiple set) and second one with about 2 700 000 objects (the 100-multiple set).

We compared the following approaches for top-k search problem: grid file based approach, R-tree and R*-tree based approach [13], local TA on B^+-trees [6] and table scan (on heap file).

There are several algorithms based on ordered lists presented in [6]. All of them work with distributed data and sorted access. Moreover the original TA requires also the random access. Since we presuppose only locally accessible data (not distributed) we slightly adapted the TA in the following way: each B^+-tree which represents one ordered list (providing sorted access for one attribute) will contain all the data i.e. values of all attributes. Thus no random access is necessary because one sorted access to any ordered list provides complete information about all attribute values of one object. Such version of TA does not longer suffer from the handicap of distributed data. We made a small experiment which showed that algorithms NRA [6] and original TA are significantly less efficient than the local version of TA. Due to this handicap we have not involved algorithms NRA and original TA to the tests.

Each of the tests consists of the same set of 1100 random queries (i.e. about 200 random queries containing gradually 2, 3, 4, 5 and all 6 attributes). Not all n-attribute queries consist of the same attributes.

All of the compared approaches manage data file on disk differently. The page size is the only adjustable parameter common to all of them. Since we wanted to compare them objectively we had to find the most suitable page size for each one. For the brevity we do not present graphs showing the time dependency on page size. We found out that the best page sizes for top-k search over 20-multiple set are the following: 1 kB for R-tree, 2 kB for R*-tree, 8 kB for grid file, 4 MB for heap file (table scan) and 64 kB for B^+-trees (local TA). For the 100-multiple set we found out the following: 2 kB for R-tree, 1 kB for R*-tree, 4 kB for grid file, 2 MB for heap file (table scan) and 128 kB for B^+-trees (local TA). We strongly recommend to do such a survey for all application domains and not to consider these results to be universal. Different size of objects leads to a different capacity of page sizes and probably a different best page sizes. We compared average time of top-k query just for the best page sizes.

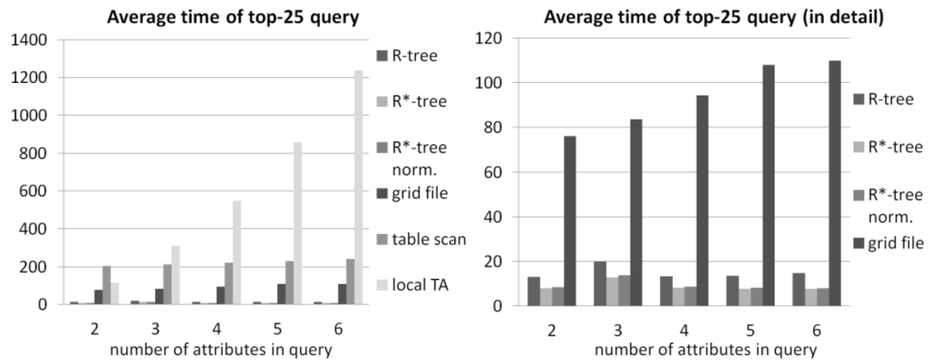

Fig. 5. The average time of top-25 query evaluation in milliseconds over 20-multiple data set (about 550 000 objects). On the right graph the table scan and the local TA are omitted for detailed comparison.

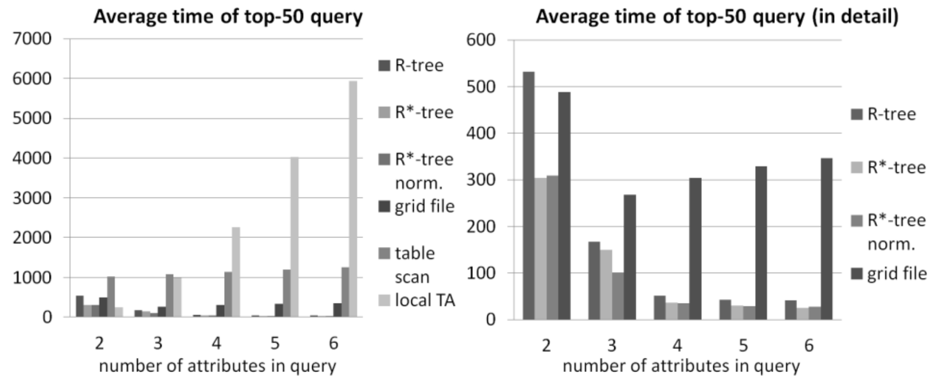

Fig. 6. The average time of top-50 query evaluation in milliseconds over 100-multiple data set (about 2 700 000 objects). On the right graph the table scan and the local TA are omitted for detailed comparison.

On the left graphs we see that number of attributes in query has a very low impact on time of table scan and very high impact on time of local TA. Moreover we can say that table scan is significantly less efficient than R-tree, R*-tree and grid file based approaches. The local TA is quite efficient for queries with only 2 attributes. Local TA loses its efficiency when 3 or more attributes are required.

R-tree, R*-tree and grid file based approaches seem to be faster therefore the graphs on the right bring the detailed look just on them. We can see that R*-tree offers a better efficiency than grid file in all cases. Moreover R*-tree with normalized data does not offer better search performance than R*-tree with original data. We did not use R-tree with normalized data because normalization of data has no effect when quadratic split algorithm is used [13].

6 Conclusion

In this paper we introduced the top-k search algorithm over grid file. Grid file is a multidimensional index structure in which we can store objects with arbitrary ordered attributes (numbers, strings, hierarchies) and which allows using a query with any subset of attributes.

Grid file organizes data by means of multidimensional intervals (windows) which are used also in R-tree as hyper-rectangles of nodes. In grid file there is no hierarchy or overlaps as it is in case of nodes of R-tree. Hence there was a question: can grid file offer better top-k search performance than R-tree or R*-tree? It would be premature to say no just because our introductory tests showed that the top-k search over R*-tree is faster. Our grid file implementation is quite simple. We have found many overflow pages and many empty windows because of the real data distribution. The results are quite promising and encourage us to look for more sophisticated ways of creating and organizing grid file.

Acknowledgement

This work was partially supported by VEGA 1/0832/12.

References

1. Akbarinia, R., Pacitti, E., Valduriez, P.: Best Position Algorithms for Top-k Queries. In VLDB, (2007)
2. Bast, H., Majumdar, D., Schenkel, R., Theobald, M., Weikum, G.: IOTop-k: Index-Access Optimized Top-k Query Processing. In VLDB, (2006)
3. Bercken, J., Seeger, B.: An Evaluation of Generic Bulk Loading Techniques. Proceedings of the 27th International Conference on Very Large Data Bases, ISBN:1-55860-804-4, pp.461-470, (2001)
4. Fagin, R., Lotem, A., Naor, M.: Optimal Aggregation Algorithms for Middleware. Proc. ACM PODS, 2001
5. Gurský, P.: Towards better semantics in the multifeature querying. Proceedings of Dateso 2006, ISBN 80-248-1025-5, pages 63-73 (2006)
6. Gurský, P., Pázman, R., Vojtáš, P.: 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.
7. Kumar, A.: G-tree: a new data structure for organizing multidimensional data. IEEE Transactions on knowledge and data engineering, ISSN: 1041-4347, pp. 341 - 347, vol. 6, issue 2, (1994)
8. Leutenegger, S. T., Nicol, D. M.: Efficient Bulk-Loading of Grid files. IEEE Transactions on knowledge and data engineering, ISSN: 1041-4347, vol. 9, no. 3, (1997)

9. Leutenegger, S.T.; Lopez, M.A.; Edgington, J.: STR: a simple and efficient algorithm for R-tree packing. Proceedings of the 13th International Conference on Data Engineering, ISBN: 0-8186-7807-0, pp. 497-506, (1997)
10. Li, C, Chang, K., Ilyas, I.F., Song, S.: RankSQL: Query Algebra and Optimization for Relational Top-k Queries. SIGMOD (2005)
11. Nievergelt, J., Hinterberger, H., Sevcik, K. C.: The Grid File: An Adaptable, Symmetric Multikey File Structure. ACM Transactions on Database Systems, pp. 33-71, vol. 9, issue 1, (1984)
12. Ondreička M., Pokorný J.: Efficient Top-K Problem Solvings for More Users in Tree-Oriented Data Structures. Proceedings of Signal-Image Technology & Internet-Based Systems, ISBN: 978-1-4244-5740-3, pp. 345-354 (2010)
13. Šumák, M., Gurský, P.: Top-k Search in Product Catalogues. Proceedings of Dateso 2011, ISBN 978-80-248-2391-1, pp. 1-12 (2011)
14. Tsaparas, P., Palpanas, T., Kotidis, Y., Koudas, N., Srivastava, D.: Ranked Join Indices. ICDE, pp.277-288 (2003)
15. Whang, K.-Y., Krishnamurthy, R.: The Multilevel Grid File - A Dynamic Hierarchical Multidimensional File Structure. Proceedings of Database Systems for Advanced Applications, ISBN 981-02-1055-8, pp. 449-459, (1992)
16. Xin, D., Han, J., Chang, K.: Progressive and Selective Merge: Computing Top-K with Ad-Hoc Ranking Functions. SIGMOD (2007)