Computing and Informatics, Vol. 28, 2009, 1001–1028, V 2009-May-13

ON SUPPORTING WIDE RANGE OF ATTRIBUTE TYPES FOR TOP-K SEARCH

Peter Gurský

Faculty of Science
University of Pavol Jozef Šafárik
Jesenná 5
040 01 Košice, Slovakia
e-mail: peter.gursky@upjs.sk

René Pázman

Softec, s. r. o.
Kutuzovova 23
831 03 Bratislava, Slovakia
e-mail: rene.pazman@softec.sk

Peter Vojtáš

Faculty of Mathematics and Physics
Charles University
Ke Karlovu 3
121 16 Praha 2, Czech republic
e-mail: peter.vojtas@mff.cuni.cz

Revised manuscript received 24 February 2009

Abstract. Searching top-k objects for many users face the problem of different user preferences. The family of Threshold algorithms computes top-k objects using sorted access to ordered lists. Each list is ordered w.r.t. user preference to one of objects' attributes. In this paper the index based methods to simulate the sorted access for different user preferences in parallel are presented. The simulation for different domain types – ordinal, nominal, metric and hierarchical – is presented.

Keywords: Top-k search, hierarchical attributes, metric attributes, user preferences

1 INTRODUCTION

Nowadays, in the era of huge databases, processing of ranked queries becomes a significant problem. Ranking queries (or top-k queries) are typical in multimedia databases, web databases and middleware systems. Such systems usually want to return only small number of objects suitable for a user without seeing all objects.

The main problem of the top-k querying is to find a good balance between query expressivity and computational complexity. Approaches that are optimal in computational complexity usually do not support high expressive queries. In the family of Threshold Algorithms (*TA*) the basic assumption is a monotone combination function over ordered sources. The expressivity of a monotone combination function over domain ordering is low. Many authors [22, 23] face the problem of more expressive queries by analyzing complicate ranking functions and offer a kind of multidimensional search.

Example 1. Imagine user u_1 looking for a low price flat with size about $60\,\mathrm{m}^2$. His overall ranking function F can be expressed as follows:

$$F_{u_1}(price, size) = \left(1 - \frac{price}{\max_{price}}\right) \cdot \max\left\{0, \left(1 - |1 - \frac{size}{60}|\right)\right\}. \tag{1}$$

It can be seen that F_{u_1} is not a monotone function and *TA* cannot be used ad-hoc.

Instead of difficult function analysis we prefer a different form of query compound of local preferences and a monotone combination function. As will be shown in Section 2, the expressivity of such a query is very high.

Local preference represents user's notion about the suitability of values from the attribute domain. Local preference can be expressed by fuzzy predicate that maps attribute domain to interval $[0, 1]$, where 1 means strong preferred domain value and 0 means the least preferable value.

The monotone combination function is used to compare objects incomparable in particular local preferences (one flat can be better in price, another in size). Typical monotone combination functions are weighted average, minimum, maximum or a set of ranking rules.

Fig. 1. Local preferences

Notice that the ranking function F_{u_1} of user u_1 can be written as a monotone combination (product) of partially linear scoring functions f_p and f_s depicted in Figure 1.

Formally the preferences of user u to a set of objects with m attributes $a_1, \ldots, a_m$ are described by m arbitrary scoring functions of one variable $f_{a_1}, \ldots, f_{a_m}$ (local preferences) and one monotone combination function F (global preferences), and for every object X with attribute values $x_1, \ldots, x_m$ the preference of user u to object X is equal to $F(f_{a_1}(x_1), \ldots, f_{a_m}(x_m))$.

In the naïve approach, the attribute vales according to local preferences can be ordered and any effective *TA*-like algorithm to find top-k flats can be used.

Unfortunately, the meaning of good values of attributes (specified by local preferences), as well as the combination function, can be different for each user.

Example 2. Consider a user u_2 with preferences to rather large flats but mainly those with price about \$ 50 k:

$$F_{u_2}(price, size) = 3 \cdot f_{p_2}(price) + f_{s_2}(size), \tag{2}$$

where

$$f_{p_2}(price) = \begin{cases} \frac{price}{10\,\mathrm{k}} - 4, & price \in \langle 40\,\mathrm{k}, 50\,\mathrm{k}\rangle \\ 6 - \frac{price}{10\,\mathrm{k}}, & price \in (50\,\mathrm{k}, 60\,\mathrm{k}) \\ 0, & \text{otherwise} \end{cases}$$

$$f_{s_2}(size) = \frac{size}{\max_{size}}$$

It can be seen that local preferences induce different ordering of the attributes than functions in Figure 1. Now the naïve approach fails – reordering in the time of the query is unacceptable.

Example 3. Assuming that the attributes price and size are indexed separately by a B+ tree the tree according to user local preferences can be traversed to simulate sorted access. In Example 2 the price tree is traversed from maximal value in descending direction using leave pointers. In the size tree two pointers can be created, both starting at 60 m^2. The first pointer traverses the tree in descending direction and the second one in ascending direction. Having simple functions (partially linear), the points of extremes to identify the pointers and directions can be found easily.

Similarly in (2), the price tree is searched using two pointers starting at 50 k and the size tree is traversed from max_{size}. The algorithm for traversing B+ tree is described in Section 4.1.

To enable top-k queries made by different users an intuitive user friendly interface has to be designed. As an alternative to complicated user input an inductive procedure can be used. Our system was integrated in the *UPRE* system described in [9]. In *UPRE*, a user defines his/her global preference by ordering or evaluating a sample of objects. Based on user global preferences the system creates input for top-k search in a two step learning process. First, local fuzzy preference functions

have to be induced for each attribute. Second, based on these functions the monotone combination function in the form of fuzzy rules is learned by an inductive logic programming method *IGAP* described in [11]. Alternatively the SVM based system in [21] can be used.

This paper faces the challenge to serve different users with various preferences over the same data simultaneously.

Instead of analyzing the complex ordering function of m variables like in [22, 23], we offer an extension of the family of Threshold Algorithms (*TA*s) [7] which allows us to cover queries generated by our preference model. In *TA*s the basic assumption is a monotone combination function over ordered sources. In terms of our model, the users in *TA*s must have identical local preferences and can differ only in global preferences.

Our extension of *TA*s is based on a simulation of sorted access without any real reordering of sources with a support of index structures. We describe the simulation over wide class of attribute types: ordinal (Section 4.1), nominal (Section 4.2), metric (Section 4.3) and hierarchical (Section 4.4).

Analysis of *TA*s followed us to a modification of *NRA* algorithm proposed by Fagin et al. [7] to improve its performance. In Section 3 two versions of our new *3P-NRA* algorithm is described. Instance optimal version of *3P-NRA* is proved to have maximal number of disk accesses equal to the number of accesses needed by standard *NRA* algorithm.

Performance study in Section 5 with 243 different user preferences over 3 synthetic datasets and 3 queries over real-world dataset shows significant decrease of the number of disk accesses and computational time against baseline approaches.

In summary, the contributions of this paper are:

- The sorted access by different indexes for 4 types of attribute domain – nominal, ordinal, metric and hierarchical – is described. Thus, the analysis of many valued functions is skipped and using of optimal *TA*-like algorithms is enabled (Section 4).
- A variant of *NRA* algorithm named *3P-NRA* is proposed (Section 3). In experiments the significant decrease of number of accesses and computational time against standard *NRA* algorithm is shown (Section 5).

2 RELATED WORK

Top-k query processing is studied in several scenarios. In the first category there are middleware based top-k algorithms, which combine several ordered sources from the individual subsystems [1, 2, 3, 7, 8, 10]. All of these algorithms assume monotone combination function and ordered sources or sources accessed only by random access. The *Best position algorithm* [1] works with the objects' positions in the ordered list in addition. All these algorithms compute the overall object value using a monotone combination function. We are not aware of solutions to different user problem in these papers. On the other hand, if the ordered sources are available, many of these

algorithms are effective and proven to be instance optimal for the top-k search of a single user (constant local preferences).

In the second category there are top-k algorithms embedded in RDBMS [13, 14, 15]. 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 the middleware algorithms. The way of ordering is not considered or implicitly the ordering of attribute domains is used.

The problem of more expressive queries was studied in different ways. The MPro [5] and Upper [4] systems access unordered sources by random access only.

In the PREFER system [12] the sorted access is provided by choosing one of several prepared ranked materialized views having ordering near to that made by ranked query. This approach requires non-trivial number of materialized views growing with the number of attributes.

Another possible view of recent research distinguishes the following: one branch of research generalizes types of data to uncertain (see e.g. [19, 18]) or to XML data (see e.g. [20]).

Zhang et al. in [23] present the OPT* algorithm combining discrete selection condition and continuous optimization over arbitrary ranking function to find the first best object. Xin et al. [22] analyse the ranking function of many variables similarly to [23] to navigate through the huge set of states over m B+ trees. If the ranking function over any domain subregion can be analyzed (to find the minimum and possibly recognize monotonicity) this approach is able to find top-k objects in an effective way.

Most of the above systems with high expressive queries use an analytic expression of a ranking function. For instance, in [22] the authors identify the function $F(A, B) = A \cdot e^{-A^2-B^2}$ to be not even "semi-monotone". For illustration, we show that this query can be represented in our model.

The function $F(A, B)$ is a product of two scoring functions of one variable $f_A(A) = A \cdot e^{-A^2}$ and $f_B(B) = e^{-B^2}$.

An analysis of the functions f_A and f_B can be used to find local maximums (needed by simulation procedure in Section 4.1). Such an analysis is much easier than the analysis of function of two or more variables and the composition of global and local preferences induces the same results. Unfortunately we do not have a decomposition procedure to convert any complex scoring function in analytical form to our form of the query.

Note that by monotone combination function also the Boolean restrictions on attribute values can be simulated. All we need is to set overall ranking value to zero if domain element does not fulfil selection condition (i.e. fuzzy predicate value is zero).

We believe that our user preference model is more intuitive for users than complex analytical function, because when they specify their preferred objects in natural language they usually express which values are more suitable than the others.

3 TOP-K ALGORITHMS

The optimal performances among all top-k algorithms have algorithms that aggregate ordered lists. In Section 4 the possibility of simulation of the sorted access over nominal, ordinal, metric and hierarchical attributes using tree like indexes is shown. This allows the use of *TA*-like algorithms.

In our implementation the modification of *NRA* algorithm [7] named *3P-NRA* (3-phased no random access) is primarily used. It will be shown in Section 5 that in practical implementation the orders of magnitude improvement over *NRA* are obtained.

In the rest of this paper the following notation is used. Value m represents the number of attributes of objects or the number of sources. An arbitrary object is denoted as $X = (x_1, \ldots, x_m)$, where $x_1, \ldots, x_m$ are real attribute values of X. $f_1, \ldots, f_m$ are arbitrary scoring functions of one variable that represent local preferences to attribute values. For each $i \in \{1, \ldots, m\}$, $f_i(x_i)$ represents a user preference to the real value of the i-th attribute of X. Let $V(X) = \{i_1, \ldots, i_n\} \in \{1, \ldots, m\}$ be a subset of known attributes $x_{i_1}, \ldots, x_{i_n}$ of X, we define $W_V(X)$ (or shortly $W(X)$ if V is known from context) to be minimal (worst) possible value of the combination function F for the object X. Monotonicity of combination function F is assumed. $W_V(X)$ is computed such that each missing attribute is substituted by the minimum of appropriate scoring function. For example if $V(X) = \{1, \ldots, g\}$ then $W_V(X) = F(f_1(x_1), \ldots, f_g(x_g), \min(f_{g+1}), \ldots, min(f_m))$.

Analogously we define maximal (best) possible value of the combination function F for object X as $B_V(X)$ (or shortly $B(X)$ if V is known from context). Since the sorted access returns values in descending order, the corresponding value from the vector $u = (u_1, \ldots, u_m)$ can be substituted for each missing value, where $u_1, \ldots, u_m$ are the last scoring values last seen by sorted access from each source (these values are scoring values for attributes of different objects usually). For example if $V(X) = 1, \ldots, g$ then $B_V(X) = F(f_1(x_1), \ldots, f_g(x_g), u_{g+1}, \ldots, u_m)$.

The real value of object X is $W(X) \leq F(f_1(x_1), \ldots, f_m(x_m)) \leq B(X)$. Note that unseen objects during the computation (no values are known) have $B(X) = F(u_1, \ldots, u_m)$. The value $\tau = F(u_1, \ldots, u_m)$ is well known as the threshold value.

In algorithms the top-k list T ordered by the worst value is used. The object in T with the smallest worst value is labeled T_k. In the (unordered) set C the candidates with the worst value smaller than or equal to $W(T_k)$ but with the best value larger than $W(T_k)$ are stored. These are the objects with a hope to get into T later. The objects in C are called candidates.

In *3P-NRA* the set C is implemented as a hash table with object identifiers as a key. To recognize the sources with all known values between objects in T and C the number of missing values for each source is maintained.

3P-NRA algorithm works as follows:

```
Input:  k, F, m sources
Output:  k ranked objects if exist
```

$T = \emptyset$, $C = \emptyset$
Phase 1:
Do the sorted access in parallel to all sources.
For every object X seen under sorted access compute $W(X)$ and do
    If $|T| < k$ then put X to T
    Else If $W(X) > W(T_k)$ then
            If $X \notin T$ move T_k to C, put X to T
        Else put X to C
If $W(T_k) \geq$ threshold τ goto Phase 2
repeat Phase 1
Phase 2:
Do the sorted access in parallel to the sources for which there are unknown values for objects in C and T.
For every object X seen under sorted access do
    If X is not in T or in C ignore it
    Else If $B(X) \leq W(T_k)$ remove X from C
        If $|C| = 0$ return T and exit
        If $W(X) > W(T_k)$ and $X \notin T$ move T_k to C, put X to T
If ($W(T_k)$ increased) OR (threshold τ decreased) then
   heuristic H can choose to go to Phase 3
repeat Phase 2
Phase 3:
For every object $X \in C$ compute $B(X)$; If X is no more relevant (i.e. $B(X) \leq W(T_k)$) remove X from C
If $|C| = 0$ return T and exit; otherwise goto Phase 2.

Phase 1 works similarly to the standard *NRA* algorithm with the exception of the threshold test. The heuristic H in *3P-NRA* algorithm can be used to skip an expensive computation of Phase 3. On the other hand, if H always chooses to do Phase 3 the *3P-NRA* algorithm is proved to be instance optimal. The instance optimality of *NRA* means that if *NRA* finds top k objects using y sorted accesses, then there are no algorithms reading the sources only by sorted access, that can find top k objects using less than $m \cdot y$ sorted accesses (see [7]).

Theorem 1. Let F be a monotone combination function, then algorithm *3P-NRA* finds top-k objects correctly.

Proof. Let $S(X) = F(f_1(x_1), \ldots, f_m(x_m))$. Assume that *3P-NRA* algorithm ended its computation at position $z = (z_1, \ldots, z_m)$ (i.e. algorithm did z_i sorted accesses to the i^{th} source) and returned objects $X_1, \ldots, X_k$. Let $X \notin \{X_1, \ldots, X_k\}$. We will show that $B(X) \leq W(T_k)$.

Note that the $W(T_k)$ cannot decrease during the computation and in the whole algorithm we removed from C only the objects with the best value smaller than or equal to actual $W(T_k)$. Thus it holds for each such object X that its best value is

smaller than or equal to the $W(T_k)$ at the end of the computation, more formally it holds for every removed object X from C holds that $S(X) \leq B(X) \leq W(T_k)$.

Next we need to consider the objects ignored in Phase 2. These are the objects not seen in Phase 1. Let X be such object with scores $f_1(x_1), \ldots, f_m(x_m)$ and let $v_1, \ldots, v_m$ be equal to values $u_1, \ldots, u_m$ at the end of Phase 1. It holds that $f_i(x_i) \leq v_i$ for each $i \in \{1, \ldots, m\}$. It can be seen from the monotonicity of combination function that $S(X) = F(f_1(x_1), \ldots, f_m(x_m)) \leq B(X) = F(v_1, \ldots, v_m)$. The last equality goes from the fact that X was not seen in Phase 1. It is known from the condition at the end of Phase 1 that $W(X_j) \geq F(v_1, \ldots, v_m)$ for each $j \in \{1, \ldots, k\}$ thus $B(X) \leq W(T_k)$.

Note that the fact that the sources where all values between objects in T and C are known can be skipped follows from the observations that all unseen objects can be ignored. From such sources the interesting objects cannot be retrieved any more. □

Theorem 2. Let F be a monotone combination function. If heuristic H always chooses to go to Phase 3, then algorithm *3P-NRA* makes at most the same number of sorted accesses as *NRA* algorithm, i.e. *3P-NRA* algorithm is instance optimal.

Proof. It can be seen that Phase 1 accesses the sources in the same way as *NRA* [7] algorithm as well as Phase 2 + Phase 3 together except the skipped sorted accesses to useless sources.

All we need to show is that algorithm *NRA* cannot stop earlier than Phase 1 does. Algorithm *NRA* ends when there are no more relevant objects out of the list T i.e. $B(X) \leq W(T_k)$ for all $X \notin T$. Phase 1 ends when $F(u_1, \ldots, u_m) \leq W(T_k)$. Observe that $F(u_1, \ldots, u_m)$ is the best value for all unseen objects. We need to wait until $F(u_1, \ldots, u_m)$ decreases to have $W(T_k) \geq F(u_1, \ldots, u_m)$ because we have to be sure that there is no object with the best value greater than $W(T_k)$. □

The improvements of the *3P-NRA* algorithm in contrast to *NRA* [7] are as follows:

- New objects are considered in phase 1 only. Other objects are ignored.
- Many computations of the best values are omitted.
- After acquisition of all unknown values of any attribute between objects in $T \cup C$ the algorithm stops the work with the corresponding source (no more sorted accesses to the source will be done). This feature decreases the number of disk accesses significantly.
- A good choice of heuristic H can yield a massive speedup of the algorithm, however it can slightly increase the number of disk accesses according to H.

In our tests in Section 5 the heuristic that goes to phase 3 only each $1\,000^{\text{th}}$ loop of phase 2 is used. In the tests the orders of magnitude speedup against algorithms without heuristic is shown.

4 INDEX BASED SORTED ACCESS

To accommodate to system which has to serve many different users and possibly various domains, we decided to keep data on disk. It is assumed that our system is query intensive and data insertions and deletions will be rare (e.g. updated once per day). Many functionalities of traditional transaction oriented RDBMS will not be used. Expected querying is structurally easy. This led to the estimation that the overhead of classical systems is too high and we decided to implement an own backend data organizing system based on indexes – one per each attribute. In this section the simulation of sorted access over ordinal, nominal, metric and hierarchical attributes is described. Such algorithms are needed to compute more complex queries typical for systems with many different users.

4.1 Ordinal Attributes: B+ Trees

The most common attributes are the ordinal ones. Typical structures for indexing the ordinal attributes are B-trees and B+ trees. In our implementation the B+ tree having all data in the tree and leaves stored on the disk has been slightly modified. Pages of leaf data are double linked and they can be easily traversed in either direction. The leaf traversal supported by B+ tree allows us to follow records (attribute values) from higher ranking to lower ranking. When a user defines a scoring function over the ordinal attribute, he/she actually defines the ordering of the domain. We assume that the found local maximums of the scoring function can be derived. Having partially linear scoring function, the search of local maximums is trivial. In the case of arbitrary scoring function in analytical form, some kind of additional analysis needs to be done. Local maximums are the starting points of traversing the B+ tree. After delivering the local maximums, the whole scoring function is used as a black box.

In the following algorithm the simulation of sorted access for one ordinal attribute and an arbitrary scoring function f is described. Let o be an object with value x_o in the given attribute, any triple $[p(o), f(x_o), direction_o]$ in T represents the following: $p(o)$ is the position of object o in a materialized leaf in the memory, $f(x_o)$ is the relevance of object o given by scoring function f and third value represents the direction of the next move after returning the pair $\langle o, f(x_o)\rangle$ to the master algorithm.

Function $\max(T)$ returns triple with highest score. If there are more triples with the same score, $\max(T)$ returns the one with the lowest object identifier.

Algorithm simulating the sorted access over B+ tree works as follows:

```
Input:   scoring function f (as a black box)
         Set M ={ x : f(x) is a representative of each local
          maximum of f }
Output:  The next best object with its score value in the sorted
          output stream
```

Function getNext():

```
If (first call) then
   For all x ∈ M do
       Traverse from the root of B+ tree to find neighbor records
        ⟨a, x_a⟩, ⟨b, x_b⟩ in a leaf (or leafs) such that x_a ≤ x ≤ x_b
       Add triples [p(a), f(x_a), left] and [p(b), f(x_b), right] to T, if there
        were no such records on the left or on the right do not add
        the corresponding triple.
   [p(o), f(x_o), direction_o] := max(T).
   Return ⟨o, f(x_o)⟩.
Else
     If T = ∅ return "no more objects";
     [p(o), f(x_o), direction_o] := max(T);  //last returned object
     Remove [p(o), f(x_o), direction_o] from T
     If direction_o from p(o) shows to the leaf on the disk, load it to
      the memory, if there is not such a leaf return getNext() ;
     Traverse to the record ⟨u, x_u⟩ defined by p(o) and direction_o.
     If [p(u), f(x_u), opposite(direction_o)] ∈ T //in local minimum
        Replace it by [p(u), f(x_u), null] in T
     Else add [p(u), f(x_u), direction_o] to the set T
     [p(v), f(x_v), direction_v] := max(T).
     If direction_v = null remove [p(v), f(x_v), direction_v] from T
     Return ⟨v, f(x_v)⟩.
```

Fig. 2. B+tree traversing

Example 4. To illustrate the simulation of sorted access consider the situation in Figure 2. Under the B+tree there is the fuzzy (score) function. It has one local maximum: $M = \{$ a random value in the interval $\langle 200, 250\rangle$, e.g. $225\}$. When top-k algorithm calls for the first object by sorted access, the random value in the interval (e.g. 225) by hierarchical traversal needs to be searched. Two neighbor objects around 225 are found, i.e. the triples $[p(o10), 1.0, \mathit{right}]$ and $[p(o1), 1.0, \mathit{left}]$ are added to T. Seeing that the both scores are the same the object with lower object id is returned, i.e. record $\langle o1, 1.0\rangle$. In further sorted access calls one of the gray arrows will be followed to get new objects to return. Next we traverse left for the object o3 with score 0.9 but returning better record $\langle o10, 1.0\rangle$. After the next sorted access call the direction of the last returned object is followed (i.e. to the right) and fuzzy value 1.0 of object o7 is computed and $\langle o7, 1.0\rangle$ is returned. After the next sorted access call and computing fuzzy value 0.6 of object o4 object o3 needs to be returned and the traverse to the left in the next call is done.

Traversing of the B+tree is much easier when the scoring function f is monotone. In this case only one direction is followed (the size of T is 1).

It can be easily shown that our **getNext()** function over B+tree accesses only the necessary leaves, thus it does the optimal number of disk accesses.

4.2 Nominal Attributes: Dis-Index

Usual types of nominal attributes are String and/or list of domain values (Enumeration). The unordered character of nominal attributes very often causes their absence in rank aware queries. Commonly, the nominal attributes are used to express the restrictive parts of queries only.

We believe that there are many situations, in which a user wants to express relevance to nominal attribute values in a scoring function. For instance consider the attribute *sellerType* in our flat scenario with possible values "mediator", "private person", "corporate entity" and "estate agency". Each user can specify his/her meaning about a suitability of each seller type respective to his/her experience. Similar attributes in our example domain can be e.g. house color, building material, floor type etc.

The suitability can be expressed by a fuzzyfication of nominal attributes, i.e. by assigning fuzzy values (scores) to the domain attribute values one by one. The assignment is expressed by a user and can be completely different for each user.

We created a very simple index structure Dis-index depicted in Figure 3. In Dis-index each domain value is stored in memory together with the corresponding pointer to the first page on the disk with objects' identifiers having appropriate domain value. Every inner page on the disk stores also the pointer to the next page holding identifiers with the same attribute value. Dis-index stores much more objects per page than B+tree, i.e. needs less accesses to the disk. We focus especially on small attribute domains where the fuzzyfication made by a user is easy to specify. The main problem of Dis-tree over bigger domains is many ties. If the cardinality of an active domain of the nominal attribute is big, the B+tree is better choice.

The simulation of the sorted access over Dis-index is straightforward. If there is a fuzzyfication f of the actual attribute domain, the domain values are sorted according to f in memory. First the identifiers of the value with the best fuzzyfication value are returned. If all objects under the best value are returned we follow the pages under the next best value etc.

Fig. 3. Dis-index

Example 5. Consider the situation in Figure 3 with fuzzyfication

$$\{\langle\text{"mediator"}, 0.6\rangle, \langle\text{"private person"}, 1.0\rangle, \langle\text{"corporate entity"}, 0.2\rangle,$$

$$\langle\text{"estateagency"}, 0.9\rangle\}$$

Together with the sorted access calls, first the values under "private person" are returned, i.e. records $\langle o1, 1.0\rangle, \langle o11, 1.0\rangle$, followed by "real estates" objects: $\langle o3, 0.9\rangle$, $\ldots, \langle o17, 0.9\rangle$, "mediator" objects and finally the objects under "corporate entity".

The correctness of the sorted access over Dis-index is evident. The big number of stored identifiers per page makes the Dis-index an effective variant to the B+ tree for small attribute domains, especially for nominal attributes.

4.3 Metric Attributes: M-Tree

Metric attributes are common in many areas. A typical metric attribute in multimedia databases is color. The color distance between two objects is typically expressed by cosine distance between two color histograms. In our real estates scenario, user usually counts with the locations of flats when evaluating the overall ranking of the flats. An advantage of the M-tree [6] (e.g. in contrast to R-tree) is that it allows arbitrary metrics. For example the metrics as the traveling time between any two objects in rush hours can be defined e.g. between flat and the city center.

A standard query over metric attribute is k-nearest neighbor (kNN) query [6]. We want to go further and simulate the sorted access stream over the fuzzy predicate $\mathit{suitableDistance}(\mathbf{O}, \mathbf{X})$, where $\mathbf{O}$ is user specified anchor point and $\mathbf{X}$ is an arbitrary point from the attribute domain. Using such fuzzy predicate, for example the preference to the flats from 5 to 20 km from the Bratislava city center can be expressed, which is typically the quiet zone far from the noisy center but not far to travel to.

The structure of M-tree is based on hierarchical organization of data objects according to a given metric d. The indexed data objects are recursively clustered in hyper-spherical metric regions associated with M-tree nodes. The inner nodes contain routing entries describing the metric regions, while the ground entries (stored in leaves) represent the indexed data objects. A routing entry, stored in an inner node, is denoted as:

$$\mathit{rout}(\mathbf{O}) = [\mathbf{O}, \mathit{ptr}(T(\mathbf{O})), r_{\mathbf{O}}, d(\mathbf{O}, \mathit{Par}(\mathbf{O}))]$$

where $\mathbf{O}$ is a routing object (a local pivot), $\mathit{ptr}(T(\mathbf{O}))$ is a pointer to the subtree $T(\mathbf{O})$ of $\mathit{rout}(\mathbf{O})$ (called covering subtree), $r_{\mathbf{O}}$ is a covering radius and $d(\mathbf{O}, \mathit{Par}(\mathbf{O}))$ is a precomputed distance, where $\mathit{Par}(\mathbf{O})$ is the parent routing object. Unlike the inner entities, the ground entries lack the pointer to a subtree as well as the covering radius. The ground entry over indexed object $\mathbf{O}$ will be denoted as $\mathit{grnd}(\mathbf{O})$. In situations where the type of node entry is not important, the routing entry $\mathit{rout}(\mathbf{O})$ or ground entry $\mathit{grnd}(\mathbf{O})$ will be denoted simply as $\mathit{entry}(\mathbf{O})$.

By defining the *SortedStreamMTree* algorithm for M-tree the sorted access over arbitrary anchor object and arbitrary scoring function can be simulated. Note that here is no need to search the local maxima.

SortedStreamMTree:

```
Input:   anchor object Q , scoring function f (as a black box)
         M-tree M using metric d
Output:  The next best object with its score value in the sorted
          output stream
```

Initialization:
$L = \emptyset$;
Load the root node of M
For each $entry(\mathbf{X})$ in root compute (for ground entries $r_{\mathbf{X}} = 0$):
$d_{min}(\mathbf{X}) = \max\{0, d(\mathbf{Q}, \mathbf{X}) - r_{\mathbf{X}}\}, d_{\max}(\mathbf{X}) = d(\mathbf{Q}, \mathbf{X}) + r_{\mathbf{X}}$,
$f_{\min}(\mathbf{X}) = \min f(z) : z\langle d_{min}(\mathbf{X}), d_{\max}(\mathbf{X})\rangle$
$f_{\max}(\mathbf{X}) = \max f(z) : z\langle d_{min}(\mathbf{X}), d_{\max}(\mathbf{X})\rangle$
Input $[entry(\mathbf{X}), f_{max}(\mathbf{X}), f_{min}(\mathbf{X})]$ into list L.
Triples in L are ordered by $f_{\max}$ in descending order. Triples with equal $f_{\max}$ value are secondary ordered by $f_{\min}$ also in descending order. Triples in L can hold routing entries as well as ground entries. Ground entries (points) are always ordered in front of the routed entries with the same $f_{\max}$ value.

Function getNext():
Do {
Remove the first triple $[entry(\mathbf{X}), f_{\max}(\mathbf{X}), f_{\min}(\mathbf{X})]$ from L.
If $\mathbf{X}$ is a ground entry, return $\langle \mathbf{X}, f_{\max}(\mathbf{X})\rangle$ and exit;
Load child node of $rout(\mathbf{X})$ using $ptr(T(\mathbf{X}))$.
For each $entry(\mathbf{Y})$ in node $T(\mathbf{X})$ do
Compute $f_{\min}(\mathbf{Y})$ and $f_{\max}(\mathbf{Y})$
Put $[entry(\mathbf{Y}), f_{\max}(\mathbf{Y}), f_{\min}(\mathbf{Y})]$ into the proper position in L
} While $L \neq \emptyset$;
Return ''no more objects'';

Fig. 4. Fuzzy function over metric attribute in M-tree

Example 6. To illustrate the simulation of sorted access stream over M-tree, consider the situation in Figure 4. In this example there are 11 objects (**A**–**K**) stored in an M-tree. The fuzzy function specifies the anchor object **Q** and the fuzzy function f with the maximum value in the gray zone and the zero value for the objects behind the dashed line. In the phase of initialization the values $d_{\min}(\mathbf{A})$, $d_{\max}(\mathbf{A})$ and $d_{\min}(\mathbf{B})$, $d_{\max}(\mathbf{B})$ are computed for root entries, i.e. the minimum and the maximum possible distance of the points in appropriate covering radius. By analysis of fuzzy values in intervals $\langle d_{\min}(\mathbf{A}), d_{\max}(\mathbf{A})\rangle$ and $\langle d_{\min}(\mathbf{B}), d_{\max}(\mathbf{B})\rangle$ the range of possible fuzzy values in both covering areas is obtained. Finally the sorted list $L = ([rout(\mathbf{A}), 1.0, 0.0], [rout(\mathbf{B}), 1.0, 0.0])$ is created. The first **getNext()** call works as follows. First, the triple $[rout(\mathbf{A}), 0.0, 1.0]$ is removed from L and the procedure goes to the left child of the root node. Processing the node the new list L with values like $([rout(\mathbf{B}), 1.0, 0.0], [rout_2(\mathbf{A}), 0.9, 0.0], [rout_2(\mathbf{C}), 0.0, 0.0])$ is obtained where $rout_2$ means the routing entry from the second level of the M-tree. After next cy-

cle the list L looks like ($[rout_2(\mathbf{B}), 1.0, 0.2]$, $[rout_2(\mathbf{E}), 1.0, 0.1]$, $[rout_2(\mathbf{A}), 0.9, 0.0]$, $[rout_2(\mathbf{D}), 0.8, 0.0]$, $[rout_2(\mathbf{C}), 0.0, 0.0]$). The last complete cycle in this **getNext()** call will produce the list L with triples like ($[grnd(\mathbf{B}), 1.0, 1.0]$, $[rout_2(\mathbf{E}), 1.0, 0.1]$, $[rout_2(\mathbf{A}), 0.9, 0.0]$, $[grnd(\mathbf{H}), 0.8, 0.8]$, $[rout_2(\mathbf{D}), 0.8, 0.0]$, $[grnd(\mathbf{I}), 0.7, 0.7]$, $[rout_2(\mathbf{C}), 0.0, 0.0]$). Finally $[grnd(\mathbf{B}), 1.0, 1.0]$ is removed from L and $\langle \mathbf{B}, 1.0 \rangle$ as a result of the first call of **getNext()** is returned.

The correctness of our sorted stream algorithm over M-tree is guaranteed by the character of the list L. A triple with the ground entry at the first position of the list can be found only if there are no routing entries with possibly better objects in the covering area. Counting with the $f_{\min}$ values higher preference to the routing entries can be set with higher probability to find good objects. This feature is interesting when a fuzzy function has intervals of the same score value. Ordering of the ground entries in front of the routed entries with the same $f_{\max}$ value inhibits the unnecessary node accesses when $f_{\max}$ value equals the $f_{\min}$ value.

Similar simulation algorithm can be generated to search over R-tree or other spatial structures.

4.4 Hierarchical Attributes: Hierar-Index

Hierarchical attributes are typical for ontologies, where objects have properties with values arranged into a hierarchy of classes or instances. It can be a classification of concepts, whole-part relationships, or any other tree structure of objects. Different ontology properties can be used to relate distinct nodes in a tree; typical are `rdfs:subClassOf` and `rdf:type`.

The main problem we found about the hierarchical attributes is that users are not able to rate every node in the hierarchy tree. Hierarchies can easily have hundreds of nodes. Users are covenient, and rating of 1-5 nodes in the hierarchy is a standard amount. Sorted access to objects with attribute values in a hierarchy corresponds to a linear ordering of the hierarchy tree. Some heuristics to make user rating with the linear ordering will be discussed.

Example 7. Imagine the attribute job position in the domain of job offers and a user who wants to find a job in IT most likely as a Linux administrator or a Java programmer. So he rates the Linux administrator and the Java programmer with high ratings. But what if there are other IT positions like C++ programmer or Solaris administrator with higher salary or closer to his home? We don't want to ignore the job positions in the neighborhood of the rated job positions. We prefer to send them with lower scores in a sorted access stream after the most preferred job positions.

Example 8. Another typical hierarchical attribute of many application domains is Location. Location represents a political hierarchy of regions (countries, states, districts, cities, neighbourhood units). A user looking for a new house in Slovakia

close to Bratislava can combine the metrical attribute with the specification of preference to houses near the anchor point in the center of Bratislava and the hierarchical attribute of Locations with high preference to Slovakia and negative preference to Hungary, Austria and Czech Republic which are very close to Bratislava. He can also specify a low preference to some districts he does not prefer, like Petržalka or villages near the Slovnaft refinery.

4.4.1 Distance Evaluation in a Hierarchy

In Section 4.4.4 our method of scoring the nodes not rated by a user will be discussed. In this method unsymmetrical distances between nodes of a hierarchy are used. To describe the distances only one node rated by a user with value 1.0 (strong preference) will be considered. The value of the rated node is called the searched value. The aim of the distances is to order nodes from the nearest to the farthest according to the similarity with the single searched value. The expected order is intimated in the following example.

Example 9. Imagine a situation where a user searches for a flat in Bratislava. Attributes for flat location refer to the hierarchy of regions where the root is Slovakia and leaves are villages, towns and city parts. The top-k search should offer flats associated with Bratislava node at first and then flats from Bratislava's districts, i.e. the subtree of Bratislava's node. Afterwards it should offer flats from region containing Bratislava (the direct parent of Bratislava's node), then from other towns from the same region (siblings of Bratislava) and then from districts of these towns. If this is not enough, it should find flats from Slovakia (the grand-parent of Bratislava) and afterwards from its other sub-regions etc.

The distance evaluation is taken from a method named Criteria Search [17], which uses hierarchical classifications of objects in order to find the best correspondence to user explicit preferences. The user specifies expected values of (mainly hierarchical) search criteria and the method evaluates each candidate object according to the user preferences.

The unsymmetrical distance from node v_s to node v_o is defined as the length of the shortest path between them:

$$dist(v_s, v_o) = \sum_{e \in spath(v_s, v_o)} \delta(e). \tag{3}$$

$spath(v_s, v_o)$ is the set of directed edges laying on the shortest path from v_s to v_o. $\delta(e)$ is the length of the directed edge e, which is described below.

To find the shortest path between two nodes, their lowest common ancestor needs to be found and the paths from the first node to the common ancestor and from the common ancestor to the second node need to be united:

$$spath(v_s, v_o) = spath_{up}(v_s, v_{ca}) \cup spath_{down}(v_{ca}, v_o) \tag{4}$$

where v_{ca} is the lowest common ancestor of v_s and v_o. $spath_{up}$ and $spath_{down}$ are sets of directed edges of the appropriate paths:

$$spath_{up}(v_s, v_{ca}) = \{(v_c, v_p) \mid v_c, v_p \text{ are values in the path from } v_s \text{ to } v_{ca} \text{ and } v_p \text{ is a direct parent of } v_c\} \quad (5)$$

$$spath_{down}(v_{ca}, v_o) = \{(v_p, v_c) | v_c, v_p \text{ are values in the path from } v_o \text{ to } v_{ca} \text{ and } v_p \text{ is a direct parent of } v_c\}. \quad (6)$$

We want to preserve the expected order (from Example 9) given by distance from the node having searched value. In this order wewant to process all descendants of the preferred node before its parent. Thus the distance from a node to its parent should be greater than distances to its descendants. Hierarchies have also a property of increasing similarity of concepts moving from the top of the hierarchy tree to its leaves. The children of the root are high level concepts whereas siblings in the lowest level of the hierarchy are specific concepts similar to each other. Thus the edge distances in the lower part of the tree should be smaller than edge distances in the higher parts. In the spirit of the above considerations, the edge distance is dependent on its direction and its depth in the tree. In [17] the distance of the directed edge e is defined as:

$$\delta(e) = \gamma(e)\varepsilon^{depth(e)} \text{ where} \quad (7)$$

$$\gamma(e) = \begin{cases} \gamma^+, & \text{if the edge } e \text{ is oriented from a child to a parent, } \gamma^+ > 0 \\ \gamma^-, & \text{if the edge } e \text{ is oriented from a parent to a child, } \gamma^- > 0. \end{cases} \quad (8)$$

$depth(e)$ represents the depth of the edge from the root of the hierarchy and ε denotes the discount of the edge distance from the top to the bottom of the hierarchy and should be slightly less than 1. γ^+ should be distinctly greater than γ^-.

Pázman in [17] choses the values as $\gamma^+ = 1.0$, $\gamma^+ = 0.2$ and $\varepsilon = 0.9$. It is a sufficient condition to make sure that the distances to the descendants of any node are smaller than to its ancestor.

Theorem 3. For an ancestor node of searched value (SV), say A, the distance from SV to any descendant of A (DA) is less than distance from SV to any node not placed in the subtree of A (NA) if

$$\gamma^+ > \sum_{i=0}^{depth_{\max}-1} (\gamma^- \varepsilon^i). \quad (9)$$

Proof. We want to prove that $dist(SV, DA) < dist(SV, NA)$.

From the definition of $dist$ it holds: $dist(SV, DA) = dist(SV, A) + dist(A, DA)$. Similarly, $dist(SV, NA) = dist(SV, A) + dist(A, NA)$.

So we need to prove: $dist(A, DA) < dist(A, NA)$.

Let A's depth in the tree be $depth_A$ and DA's depth be $depth_{DA}$. DA is a descendant of A, so $depth_{DA} > depth_A$.

The distance between these two nodes can be calculated as:

$$dist(A, DA) = \sum_{i=0..(depth_{DA}-depth_A)} \gamma^- * \varepsilon^{depth_A+i} = \varepsilon^{depth_A} * \sum_{i=0..(depth_{DA}-depth_A)} \gamma^- * \varepsilon^i.$$

Because $depth_{DA} > depth_A$ and depth of any edge is less than maximal depth of nodes ($depth_{\max}$), the following holds:

$$\sum_{i=0..(depth_{DA}-depth_A)} \gamma^- * \varepsilon^i <= \sum_{i=0..(depth_{\max}-1)} \gamma^- * \varepsilon^i$$

and thus:

$$dist(A, DA) <= \varepsilon^{depth_A} * \sum_{i=0..(depth_{\max}-1)} \gamma^- * \varepsilon^i.$$

Using statement (9), the following implies:

$$dist(A, DA) < \varepsilon^{depth_A} * \gamma^+. \tag{10}$$

On the other side, path to NA must lead through A's parent (AP). So $dist(A, NA)$ can be restricted as: $dist(A, NA) >= dist(A, AP)$ and thus:

$$dist(A, NA) >= \gamma^+ * \varepsilon^{depth_A-1}.$$

Because of $\varepsilon^{depth_A-1} > \varepsilon^{depth_A}$ and using statement (10), the following is true: $dist(A, DA) < \varepsilon^{depth_A-1} * \gamma^+$ and thus: $dist(A, DA) < dist(A, NA)$ which was to have been demonstrated. □

The situation is illustrated in Figure 5. The distance of the highlighted path from Pezinok to Petržalka is $0.9 + 0.18 + 0.162 = 1.242$. On the other hand the distance from Pezinok to Dubový vŕšok is only 0.162.

Fig. 5. The distance calculation for two hierarchy values when γ^+ is 1.0, γ^- is 0.2 and ε is 0.9

In the expected order there are the objects from the closest to the farthest. The given order can be used to simulate the sorted access. From hierarchy in Figure 5 the first objects from Pezinok will be sent, then the objects of the children nodes of Pezinok and all their descendants. After processing the whole subtree of Pezinok the objects in Bratislava region are processed, then the siblings of Pezinok and their children after that. The next processed node will be Slovakia followed by the rest of the hierarchy.

The scores of the nodes can be computed as $1 - (dist(v_s, v_o)/\max_{dist})$, where $\max_{dist}$ represents the distance from the deepest leaf in the hierarchy to its farthest node and vs is the searched value.

4.4.2 Hierar-Index

Now the index for the hierarchical attributes will be explained. This index is similar to the Dis-index (see Section 4.2) – attribute values are organized in a memory structure and object identifiers are stored on disk. The Hierar-index holds the information about nodes and domain values in memory and object identifiers on disk pages. Each node in the hierar-index is represented as

$$x = [label(x), Par(x), \{Ch_1(x), \ldots, Ch_c(x)\}, offset_x] \tag{11}$$

where $label(x)$ represents a unique domain value, $Par(x)$ is a pointer to the parent node, $\{Ch_1(x), , Ch_c(x)\}$ are pointers to all children nodes of the node x and finally $offset_x$ represents the position of the first page on disk, where identifiers of the objects having attribute value $label(x)$ are stored. The node in Figure 6 has the label "Programmer".

Fig. 6. Hierar-index node

Hierar-index is updated off-line and no node changes in the time of a query are done.

4.4.3 Sorted Access Simulation

To support the sorted access over a hierarchical attribute the nodes of the hierarchy need to be processed according to their scores in descending order. The simulation presented in this chapter is independent of the method of scoring the nodes that are not evaluated by a user.

The sorted access simulation algorithm is similar to the algorithm used in the Dis-index. The simulation procedure returns the rated records according to the rating of the attribute values as usual.

The sorted access simulation procedure is proposed to be more general to use various scoring techniques. The top-k algorithm stops reading of the rated records after it has the top-k objects. Rating of some nodes can be useless when the whole subtree will be scored with zero or when descendants can be processed after the processing of their ancestor i.e. they have smaller or equal ratings to their ancestor.

The simulation function **getNext()** works with the priority queue Q containing some nodes of the Hierar-index. Priority queue Q is organized according to scores of the nodes having the node with the highest score on the top of the queue. Priority queue can be updated with useful objects in line 7 of the **getNext()** during the sorted access simulation.

Initialization and update of Q depends on the method of scoring the nodes not rated by a user. One of the possible methods is proposed in Section 4.4.4.

The update is given by scoring nodes method too. If all nodes have their scores before the first call of the **getNext()** function, this line can be skipped.

Function getNext():
```
If (currentNode = null) and (Q = ∅) return ''No more objects'';
If currentNode = null then
   Remove node x with the highest score(x) from Q
   currentNode := x;
X := getNextObjectFromNode(currentNode);
If X = null then
   updateQueue(Q, currentNode);
   currentNode := null;
   return getNext();
Else return ⟨X, score(currentNode)⟩;
```

Function **getNextObjectFromNode** loads disk pages relevant to the given node if necessary and returns object identifiers one per call. If a node contains no objects or all the objects appertaining to the node were returned, the function returns `null`.

The procedure **updateQueue** is correct if it does not add a node that was already processed or the node with higher score than any of already processed nodes has. Then the **getNext()** function processes the nodes in correct order from the nodes with the highest scores to the nodes with the lowest scores, and the whole simulation of the sorted access is correct.

4.4.4 Proposal of the Method of Scoring the Nodes Not Rated by User

It is probably impossible to generate a universal system for scoring the not rated nodes because each hierarchical attribute has its own semantics. Imagine the hierarchy of localities and a user that rated Bratislava as the preferred node. If the attribute represents the area where he can use the services of a delivery company then the company delivering in Nitra should be rated with zero. On the other hand if the attribute is a job location then Nitra can be interesting and needs to be rated higher.

The object associated with the inner node can usually have two different semantics.

Firstly it can be an uncertain information, i.e. the real (unknown) value of the object attribute is somewhere in the subtree. For example it could be known that a hotel is placed in High Tatras but the exact place is not known.

Secondly the object can fulfil the properties of the whole subtree, e.g. the company delivering the packages in Bratislava can deliver a package to any part of Bratislava.

We focus on the first semantic.

In Section 4.4.1 the unsymmetrical distance is used in a scoring method allowing the rating in interval $[0, 1]$ of more than one node.

The scoring of the nodes has two phases. First an initial priority queue is created

and then Q is updated when the sorted access simulation algorithm **getNext()** calls the procedure **updateQueue**. The input for these scoring methods is a nonempty user fuzzyfication f of some nodes.

Initialization:

```
For each node x set:
    score(x) := null;
    count_x := 0;
    sum_x := 0;
For each fuzzy pair ⟨x, f(x)⟩ set score(x) := f(x);
For each node x in domain of f do
    If all ancestors y of x has f(y) = null then
       For all ancestors y of x
          count_y++;
          sum_y += f(x) · (1 − dist(x,y)/max_dist);
For each node y having count_y > 0 do
    score(y) := sum_y / count_y
Add all nodes having score(x)! = null to Q
```

updateQueue(Q, x)

```
For all children y of x having score(y) = null do
    score(y) := f(x) · (1 − dist(x,y)/max_dist);
    Add y to priority queue Q
```

These two methods can be used in simulation procedure **getNext()**. The intuition behind this procedure is to propagate fuzzyfied values through the hierarchy in accordance with distances. A fuzzyfied node influences a node without explicit preference according to its proximity (inverse to distance) to the other node. Any node is scored using its direct fuzzyfied ancestor if it exists, otherwise it is evaluated according to its fuzzyfied descendants.

Fig. 7. User preferences calculation with three nodes rated by user (numbers in squares). Numbers on the sides display distance of directed edges in the tree in the tree's various levels. Inferior index represents the round of **updateQueue** in which the value was computed.

Example 10. Figure 7 shows the more complicated situation where a user ranked three nodes with different values. The maximal distance value equals to 3.252 and is computed as a distance from nodes labeled "Centrum" or "Vajnory" to nodes labeled "Košice" or "Prešov". Note that this number can be computed in the time of indexing.

In the initialization only the score of the root node is computed. The value obtained from the node labeled "Košice reg." is $1.0 * (1 - (1/3.252)) = 0.69$, and

the value from node labeled "Bratislava reg." is $0,7 * (1 - (1/3.252)) = 0.49$. The score of the root is computed as $(0.69 + 0.49)/2 = 0.59$. The priority queue after initialization is (in this example a simplified version is used, real priority queue contains whole nodes):

$$Q_0 = \langle(\text{Košice reg.}, 1.0), (\text{Bratislava reg.}, 0.7), (\text{Slovakia}, 0.59), (\text{Centrum}, 0)\rangle.$$

The first call of the **getNext()** function removes the node "Košice reg." from the priority queue and starts to process the objects having attribute value "Košice reg." from disk and returns the first object from disk in rated record having rated value 1.0. When all objects from the disk related to "Košice reg." are processed, the simulation calls the procedure **updateQueue**. Here the node labeled "Košice" is scored by value $1.0 * (1 - (0.18/3.252)) = 0.94$ and added to priority queue:

$$Q_1 = \langle(\text{Košice}, 0.94), (\text{Bratislava reg.}, 0,7), (\text{Slovakia}, 0.59), (\text{Centrum}, 0)\rangle.$$

Now the sorted access simulation removes "Košice" from the priority queue and processes its objects. Next call of **updateQueue** does not add any node to priority queue and the "Bratislava reg." is processed. With the next call of **updateQueue** two nodes are added to priority queue:

$$Q_2 = \langle(\text{Pezinok}, 0.66), (\text{Bratislava}, 0.66), (\text{Slovakia}, 0.59), (\text{Centrum}, 0)\rangle.$$

The other updates of the priority queue are intimated by inferior indexes of the scores in Figure 7.

This generalized version of scoring nodes induces the same ordering as the approach of [17] when user ranks only one node. The computed scores slightly differ because Pázman's approach computes the score based strictly on the distance from the searched value and our generalized approach computes the scores of nodes that don't have a fuzzyfied ancestor based on the score of its parent and the distance from the parent. This comparison was verified in a small experiment in the NAZOU project [16].

There can be several other methods to score the nodes not rated by a user. The rating depends on the semantic of the hierarchical attribute. One of the easiest methods is to score all nodes in the subtree of fuzzyfied node with the same value as the fuzzyfied node and set score to zero for all nodes without a fuzzyfied ancestor.

5 EXPERIMENTS

This section reports the experiments on our system supporting multiple users over both real-world and artificial data.

The experiments are conducted on a PC having Intel Pentium M 2 GHz CPU and 512 MB RAM running on Windows XP. Lists of B+ trees 4 kB have per page which yields 338 objects per node. Internal nodes of B+ trees are stored in memory.

Nodes of Dis-index and Hierar-index have 4 kB size holding 1 021 objects per node. Nodes of M-trees have size 1kB with 35 entries per node.

5.1 Tests of Sorted Access Simulation Algorithms

In this experiment all the above-mentioned sorted access simulation algorithms are compared with table scan, i.e. with processing of preordered sorted list. Four different indexes were generated: B+tree, Dis-index, M-tree and Hierar-index, containing 100 000 randomly distributed objects.

Domain of attribute in B+tree is interval $[0, 1]$. 5 fuzzy functions in sorted access simulation over B+tree were tested. Fuzzy functions are partially linear functions between points $\langle 0, 0\rangle$, local maximum and $\langle 1, 0\rangle$ with local maxima at 0, 0.25, 0.5, 0.75 and 1.

Dis-index was tested using 10 randomly generated fuzzyfications.

M-tree contains points of Euclidean two-dimensional space randomly generated in square $[0, 2\,000] \times [0, 2\,000]$ with anchor point $[1\,000, 1\,000]$. Fuzzy functions similar to those used in simulation over B+tree were tested, but they were transformed to interval $[0, 1\,415]$.

In Hierar-index, a full 3-ary tree of height 5 with 121 nodes was created. Objects were randomly distributed over all nodes. In the simulation 10 different fuzzyfications were used. In each fuzzyfication 5 random nodes with random fuzzy score were chosen. The input constants used by distance function: γ^+ is 1.0, γ^- is 0.2 and ε is 0.9.

Fig. 8. The time needed to retrieve different number of scored objects from different storage types

The time needed to retrieve different number of objects with their score in sorted stream was measured. The average times are presented in Figure 8. Simulations over M-tree and Hierar-index are faster than table scan because there are only object identifiers on disk, thus standard tables scan needs more I/Os. B+tree is slower mainly because of partially filled pages on disk. The slowest is the simulation algorithm over M-tree. The main reasons are more space needed to represent the object and smaller page size (we have used smaller page size because the circles with less points have smaller intersections).

5.2 Tests of Top-k Search Algorithms

In all experiments 4 different algorithms were compared: Threshold algorithm (*TA*) and *NRA* algorithms from [7], instance optimal version of *3P-NRA* and finally the algorithm *3P-NRA*, labeled as *3P-NRA2*, with the heuristic H choosing to go to Phase3 of the algorithm every $1\,000^{\text{th}}$ loop of Phase2.

5.2.1 Real-World Data

Our real-world dataset contains 18 817 flats and houses obtained by wrapping the advertisement estate system `www.byt.sk` in Slovakia. We have identified 8 attributes – *price*, *rooms*, *floor*, *size*, *status*, *age*, *seller* and *place*, where *status* and *seller* are considered to be nominal attributes and *place* is a hierarchical attribute. All other attributes are continuous. To work also with a metric attribute, over 150 different real GPS coordinates were manually assigned to all places and new attribute named *coordinates* was added. As a metric the standard arc distance on the sphere was used. In all following queries only the "ORDER BY" part is presented.

Query Q1: $3*f_p(price)+2*f_r(rooms)+f_f(floor)+2*f_{si}(size)+2*f_{st}(status)+f_a(age)+f_{se}(sellerType)+3*f_{pl}(place)+2*f_c(lat,long)$; f_p and f_a are linearly decreasing fuzzy predicates, f_r and f_{si} are linearly ascending fuzzy predicates, f_f is a partially linear function between points $\langle 0,0\rangle$, $\langle 1,0.5\rangle$, $\langle 2,1\rangle$, $\langle 6,0.4\rangle$, $\langle 11,0.4\rangle$ and $\langle 12,0\rangle$ (12 is the highest floor in dataset), f_{st} is a fuzzyfication $\{\langle$"initial status"$,0\rangle$, $\langle$"partial reconstruction"$,0.3\rangle$, $\langle$"complete reconstruction"$,0.6\rangle$, $\langle$"new"$,1\rangle\}$, f_{se} is a fuzzyfication $\{\langle$"mediator"$,0.1\rangle$, $\langle$"private person"$,0.5\rangle$, $\langle$"corporate entity"$,0.7\rangle$, $\langle$"estate agency"$,1.0\rangle\}$, f_{pl} is a fuzzyfication $\{\langle$"Bratislava"$,1\rangle$, $\langle$"Petržalka"$,0\rangle\}$ (Petržalka is a part of Bratislava in which our user does not want to buy a flat) and finally f_c is a linearly decreasing fuzzy function up to 100 km from anchor point at $48°9'N$ and $17°8'E$ situated in Bratislava city center (flats behind 100 km are inadmissible). Note that using the combination of hierarchical and metric attribute the algorithm can be forced to search the flats and houses 100 km from Bratislava placed only in Slovakia using appropriate fuzzyfication of hierarchical attribute (100 km from Bratislava there are also Hungary, Austria and Czech Republic). Data in our dataset contain flats and houses in Slovakia only and therefore flats in Petržalka were supressed instead.

Query Q2: $f_s(size)*\max_{size}*\frac{1}{\max_{price}*(1-f_p(price))}*f_r(rooms)$, where:

$$f_s(size)=\frac{size}{\max_{size}},$$

$$f_p(price)=1-\frac{price}{\max_{price}}$$

and

$$f_r(rooms)=\begin{cases}1 & \text{if } rooms\in\{3,4\}\\ 0 & \text{otherwise.}\end{cases}$$

This query is a simulation of the not monotonic scoring function $\frac{price}{size}$: $P=3$ or $rooms=4$ used in [23].

Query Q3:
IF $(fp(price)\geq 0.95$ and $f_{si}(size)\geq 1.0$ and $f_r(rooms)\geq 0.9$ and $f_f(floor)\geq 1.0)$
OR $(f_p(price)\geq 0.95$ and $f_{si}(size)\geq 0.9$ and $f_r(rooms)\geq 0.95$ and $f_f(floor)\geq 1.0)$
THAN $grade\geq 1$;
IF $(fp(price)\geq 0.9$ and $f_{si}(size)\geq 0.8$ and $f_f(floor)\geq 1.0)$ **THAN** $grade\geq 0.6$;

IF ($fp(price) \geq 1.0$ and $f_f(floor) \geq 1.0$) **THAN** $grade \geq 0.4$;
The fuzzy functions in Query 3 are identical to the fuzzy functions in Query 1. Query 3 simulates IGAP learned form of monotone combination function [11].

5.2.2 Artificial Data

Our artificial data consist of is 3 different datasets. Each dataset consists of 5 attributes $(x_1, \ldots, x_5)$. Each attribute has 10 000 values in the range of $[0, 1]$. Dataset 1 and Dataset 2 were randomly generated with exponential and Gaussian distributions, respectively. Exponential distribution has mean 0 and standard deviation 0.1. Gaussian distribution has mean 0.5 and standard deviation 0.1. Dataset 3 has attribute x_1 from Dataset 1, x_2 from Dataset 2. Attribute x_3 is a full 3-ary hierarchy tree of height 5 (i.e. has 121 nodes) with randomly assigned objects to the tree nodes. Attribute x_4 represents the nominal attribute and was randomly generated with uniform distribution to 5 crisp values. Attribute x_5 is a representative of metric attribute containing 10 000 points of Euclidean two-dimensional space randomly generated in square $[0, 2\,000] \times [0, 2\,000]$.

Queries: In our experiments we focus on testing of many different users with various preferences. In each ordinal and metric attribute different users preferring either value near 0 or 0.5 or 1 represented by fuzzy functions f_0, $f_{0.5}$ and f_1, respectively were simulated:

$$f_0(x) = \begin{cases} -x+1, & x < 0.8 \\ 0, & x \geq 0.8, \end{cases}$$

$$f_{0.5}(x) = \begin{cases} |x-0.5|, & x \in (0.1, 0.9) \\ 0, & \text{otherwise}, \end{cases}$$

$$f_1(x) = \begin{cases} x, & x > 0.2 \\ 0, & x \leq 0.2. \end{cases}$$

Values 0 represent strict restrictions. Note that for metric attribute these fuzzy functions from domain of $[0, 1]$ to $[0, 1\,415]$ were transformed with the anchor point $[1\,000, 1\,000]$ to cover the distance to all data points. Nominal and hierarchical attributes have 3 different fuzzyfications. In each fuzzyfication one random domain value has the score 1.0, two random domain values have the score 0.5 and two random domain values have the value 0.0. Thus, in nominal attribute all domain values are covered and in the hierarchical attribute the score of other domain values needs to be computed by distance driven traverse in the hierarchy tree.

Using all possible combinations over 5 attributes there are $3^5 = 243$ settings of local preferences for different users. For each setting a single scoring function is used: $f_a(x_1)+2*f_b(x_2)+4*f_c(x_3)+8*f_d(x_4)+12*f_e(x_5)$ if $\min\{f_a(x_1), f_b(x_2), f_c(x_3), f_d(x_4),$

$f_e(x_5)\} > 0$; where $a, b, c, d, e \in \{0, 0.5, 1\}$, otherwise the value of scoring function equals 0.

On each dataset 243 users with 1, 5, 10 and 20 retrieved objects were simulated to yield the total of 729 queries over 3 synthetic datasets.

Fig. 9. Average time of top-k search over artificial data

Fig. 10. Improvement of average number of page accesses in contrast to *TA* (more is better)

Fig. 11. Time of top-k search with queries over real world data

5.2.3 Results

Figure 9 shows the time needed to find top-k results for queries Q1, Q2 and Q3 over our real world data. It is shown that the combination of local preferences expressed by fuzzy functions is an effective way to search top-k objects for heterogeneous queries. Observe that for all queries, our algorithms work near best search times. Especially in the most complex query Q1 the algorithm *3P-NRA2* outperforms other algorithms by more than two orders of magnitude. *TA* outperforms other algorithms for Q2 because of several fake offers like flats for free or flats having size of a small country that are identified by *TA* in few steps.

In Figure 10 the average time of searching over our 3 datasets is measured. Again, it can be seen that using the *3P-NRA2* algorithm the time needed to retrieve the final top-k results can decrease drastically. Tests show more than 5 orders of magnitude speedup. It is caused by both dropping useless sources in Phase2 and mainly the Phase3 skipping caused heuristic H (it saves a lot of CPU time).

Figure 11 shows the results of tests with the number of disk accesses. Graphs show the improvement of disk accesses in contrast to Threshold algorithm. Note that OPT* algorithm [23] was formally added as a representative of multidimensional approach. The comparison from [23] showing the improvement of OPT* algorithm against the TA (the implementation of OPT* algorithm or data used in [23] is not accessible) is used. The authors observe about 3 times less disk accesses than TA when searching the best object. Another approach in [22] is also hard to replicate because of unclear analysis of ranking function. Moreover, the authors didn't compare their approach to *TA*s.

Observe that all *NRA*, *3P-NRA* and *3P-NRA2* requires up to 110 times less disk accesses than *TA*. Finally, it can be seen that the instance optimality of *3P-NRA* according to Theorem 2 guarantees the best number of disk accesses.

6 CONCLUSIONS

In this paper a method supporting top-k answers for multiple users is presented. It proposes a new combination of back end data maintenance system and a middleware top-k optimization heuristics. Back end system consists of various indices for ordinal, nominal, metric and hierarchical attributes. The efficient variants of the NRA algorithm were proposed and querying with optimal cost was enabled. The combination of fuzzy predicates and a monotone combination function can be used to express many types of not monotone functions. Efficiency of sorted accesses for different fuzzy predicates is assured by effective index traversal.

Instead of analyzing the ranking function of m variables, an extension of *TA*s [7] was offered to be able to cover not monotone queries generated by our preference model.

Efficiency and high expressivity of queries using both real life and artificial data different users and datasets were demonstrated. The experimental results show up to 5 orders of magnitude speedup and significant disk access saving.

The our implementation is integrated also in bigger project *NAZOU – Tools for acquisition, organization and maintenance of knowledge in an environment of heterogeneous information resources* (see `http://nazou.fiit.stuba.sk`). In *NAZOU* there is a scale of tools concerning to find relevant job offers for the user and besides searching of suitable job offers to users also crawling, wrapping, annotating and data mining are considered.

Acknowledgements

This work was partially supported by Czech projects 1ET100300517, 1ET100300419 and MSM-0021620838 and Slovak projects NAZOU and VEGA 1/3129/06.

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.: IO-Top-k: Index-Access Optimized Top-k Query Processing. In VLDB, 2006.

[3] Balke, W.—Güntzer, U.: Multi-Objective Query Processing for Database Systems. In VLDB, 2004.

[4] Bruno, N.—Gravano, L.—Marian, A.: Evaluating Top-k Queries over Web-Accessible Databases. In ICDE, 2002.

[5] Chang, K. C. C.—Hwang S. W.: Minimal Probing: Supporting Expensive Predicates for Top-k Queries. In SIGMOD, 2002.

[6] Ciaccia, P.—Patella, M.—Zezula, P.: M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces. In VLDB, 1997.

[7] Fagin, R.—Lotem, A.—Naor, M.: Optimal Aggregation Algorithms for Middleware. In ACM PODS, 2001.

[8] Gurský, P.—Lencses, R.—Vojtáš, P.: Algorithms for User Dependent Integration of Ranked Distributed Information. In TCGOV, 2005.

[9] Gurský, P.—Horváth, T.—Novotný, R.—Vaneková, V.—Vojtáš, P.: UPRE: User Preference Based Search System. In IEEE/WIC/ACM Web Inteligence, 2006.

[10] Güntzer, U.—Balke, W.—Kiessling, W.: Towards Efficient Multi-Feature Queries in Heterogeneous Enviroments. In ITCC, 2001.

[11] Horváth, T.—Vojtáš, P.: Ordinal Classification with Monotonicity Constraints. In Proc. 6th Industrial Conference on Data Mining ICDM, 2006.

[12] Hristidis, V.—Papakonstantinou, Y.: Algorithms and Applications for Answering Ranked Queries using Ranked Views. VLDB Journal, Vol. 13, 2004, No. 1.

[13] Ilyas, I.—Aref, W.—Elmagarmid, A.: Supporting Top-k Join Queries in Relational Database. In VLDB, 2003.

[14] Ilyas, I.—Shah, R.—Aref, W. G.—Vitter, J. S.—Elmagarmid, A. K.: Rank-Aware Query Optimization. In SIGMOD, 2004.

[15] Li, C.—Chang, K.—Ilyas, I.—Song, S.: RankSQL: Query Algebra and Optimization for Relational Top-k Queries. In SIGMOD, 2005.

[16] Návrat, P.—Bieliková, M.—Rozinajová, V.: Acquiring, Organizing and Presenting Information and Knowledge from the Web. In: Rachev, B.—Smrikarov, A. (Eds.): Proc. of CompSysTech '06, 2006.

[17] Pázman, R.: Ontology Search with User Preferences. In Návrat, P. et al.: Tools for Acquisition, Organisation and Presenting of Information and Knowledge, 2006.

[18] Soliman, M. A.—Ilyas, I. F.—Chang, K. C. C.: Top-k Query Processing in Uncertain Databases. In Proc. ICDE, 2007.

[19] Re, Ch.—Dalvi, N. N.—Suciu, D.: Efficient Top-k Query Evaluation on Probabilistic Data. In Proc. ICDE, 2007.

[20] Theobald, M.—Schenkel, R.—Weikum, G.: An Efficient and Versatile Query Engine for TopX Search. In VLDB, 2005.

[21] Yu, H.—Hwang, S.—Chang, K.: Enabling Soft Queries for Data Retrieval. Information Systems, Elsevier, 2007.

[22] Xin, D.—Han, J—Chang, K.: Progressive and Selective Merge: Computing Top-k with Ad-Hoc Ranking Functions. In SIGMOD, 2007.

[23] Zhang, Z.—Hwang, S.—Chang, K.—Wang, M.—Lang, C.—Chang, Y.: Boolean + Ranking: Querying a Database by k-Constrained Optimization. In SIGMOD, 2006.

Peter Gurský has finished the master study at University of Pavol Jozef Šafárik in Košice, Slovakia in 2003. Nowadays he is finishing his PhD study at the same university. He works in the area of top-k search algorithms, data indexing and semantic web. He participated in the NAZOU project as a developer of a Top-k aggregator tool.

René Pázman works in Softec, s. r. o. in Bratislava, Slovakia as an analyst and project manager since 1994. He was involved mainly in creation of information systems for large financial institutions, but within Softec's endeavor of support of research in informatics, he has also participated in the research the NAZOU research project from the area of semantic web. He is also a Ph. D. student in the area of theory of multiagent systems at Slovak Academy of Sciences.

Peter Vojtáš is a Professor of computer science at the Department of Software Engineering, School of Computer Science at Charles University in Prague, Czech Republic. He graduated in 1974 in theoretical cybernetics at Charles University. After research in applications of mathematical logic in Slovak Academy of Science in the 90's he moved to computer science research at P. J. Šafárik University Košice, especially to logic programming. His recent research interest is in flexible querying; uncertainty and vagueness (imperfection) of information; logic programming – deduction, induction, abduction; semantic web, user preferences. He is the leader of the Košice group in the NAZOU project.