Answering Metric Skyline Queries by PM-tree

Tomáš Skopal and Jakub Lokoč

Department of Software Engineering, FMP
Charles University in Prague, Czech Republic
{skopal, lokoc}@ksi.mff.cuni.cz

Abstract. The task of similarity search in multimedia databases is usually accomplished by range or k nearest neighbor queries. However, the expressing power of these "single-example" queries fails when the user's delicate query intent is not available as a single example. Recently, the well-known skyline operator was reused in metric similarity search as a "multi-example" query type. When applied on a multi-dimensional database (i.e., on a multi-attribute table), the traditional skyline operator selects all database objects that are not dominated by other objects. The metric skyline query adopts the skyline operator such that the multiple attributes are represented by distances (similarities) to multiple query examples. The metric skyline is supposed to constitute a set of representative database objects which are as similar to all the examples as possible and, simultaneously, are semantically distinct. In this paper we propose a technique of processing the metric skyline query by use of PM-tree, while we show that our technique significantly outperforms the original M-tree based implementation in both time and space costs.

1 Introduction

As the volumes of complex unstructured data collections grow almost exponentially in time, the attention to content-based similarity search steadily increases. The concept of numeric similarity between two data entities is one of the approaches used for querying unstructured data, where a similarity function serves as a multi-valued relevance of data objects to a query (example) object. The content-based similarity search paradigm has been successfully employed in areas like multimedia databases, time series retrieval, bioinformatic and medical databases, data mining, and others. At the same time, the "similarity-centric" view on such data demands specific alternative techniques for modeling, indexing and retrieval, which dramatically differ from the traditional approaches to management of structured data (e.g., B-trees in relational databases).

In the rest of the section we introduce into the fundamentals of similarity search and briefly summarize the paper contributions.

1.1 Similarity search

Given a collection $\mathcal{C}$ of unstructured data entities (e.g., multimedia objects, like images), to query the collection we need to establish a model consisting of the

J. Pokorný, V. Snášel, K. Richta (Eds.): Dateso 2010, pp. 22–37, ISBN 978-80-7378-116-3.

object *universe* $\mathbb{U}$, a *transformation* function (a feature extraction method, resp.) $t : \mathcal{C} \rightarrow \mathbb{U}$, and a *similarity* function $\delta : \mathbb{U} \times \mathbb{U} \rightarrow \mathcal{R}$. The transformation t turns the collection $\mathcal{C}$ of original data entities into a *database of descriptors* $\mathbb{S} \subset \mathbb{U}$. In most cases the similarity function δ is expected to be a *metric* distance, because metric properties can be effectively used to index the database $\mathbb{S}$ for efficient (fast) query processing, as discussed later in Section 1.2.

Single-example queries. The portfolio of available similarity query types consists of mostly single-example queries. The *range* query and the *k nearest neighbor* (kNN) query represent the two most popular similarity query types. Using a range query (Q, r_Q) we ask for all objects $O_i \in \mathbb{S}$ the distances of which to a single query object Q are at most r_Q. On the other hand, a kNN query (Q, k) selects the k database objects closest to Q.

Besides range and kNN queries, there exist some less frequently used query types, like *reverse (k)NN queries* [15], returning those database objects having the query object Q within their (k) nearest neighbor(s).

Multi-example queries. Although the single-example queries are frequently used nowadays, their expressive power may become unsatisfactory in the future due to increasing complexity and quantity of available data. The acquirement of a query (example) object is the user's "ad-hoc" responsibility. However, when just a single example should represent the user's delicate intent on the subject of retrieval, finding such an example could be a hard task. Hence, instead of querying by a single example, an easier way for the user could be a specification of several query examples which jointly describe the query intent. Such a multi-example approach allows the user to set the number of query examples and to weigh the contribution of individual examples. Moreover, obtaining multiple examples, where each example corresponds to a partial query intent, is much easier task than finding a single "holy-grail" example.

In this paper we deal with metric skyline query (detailed in the Section 2), which represents a "native" multi-example query type.

1.2 Metric access methods

When the similarity function δ is a distance metric, the *metric access methods* (MAMs) can be used for efficient (fast) similarity query processing [16, 11, 2]. The principle behind all MAMs is the utilization of metric postulates (positiveness, symmetry and triangle inequality), which allow to partition the data space into equivalence classes of close (similar) data objects. The classes are embedded within a data structure which is stored in an *index file*, while the index is later used to quickly answer range, kNN, or other similarity queries. In particular, when issued a similarity query, the MAMs exclude many non-relevant equivalence classes from the search (based on metric properties of δ), so only several candidate classes of objects have to be exhaustively (sequentially) searched. In consequence, searching a small number of candidate classes turns out in reduced cost of the query. The number of *distance computations* $\delta(\cdot, \cdot)$ is considered as the major component of the overall costs when indexing or querying a database. Some other cost components (like *I/O costs*, internal *CPU costs*) could be taken into consideration when the computational complexity of δ is low.

In the following we briefly describe the M-tree and the PM-tree, two MAMs used further in the paper for implementation of metric skyline queries.

M-tree. The *M-tree* [5] is a dynamic metric access method that provides good performance in database environments. The M-tree index is a hierarchical structure, where some of the data objects are selected as centers (references or local *pivots*) of ball-shaped regions, and the remaining objects are partitioned among the regions in order to build up a balanced and compact hierarchy, see Figure 1.

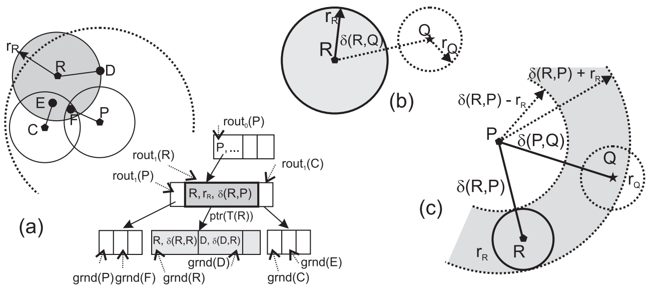

Fig. 1. (a) M-tree (b) Basic filtering (c) Parent filtering.

Each region (subtree) is indexed recursively in a B-tree-like (bottom-up) way of construction. The inner nodes of M-tree store *routing entries*

$$rout_M(R) = [R, r_R, \delta(R, \mathrm{Par}(R)), ptr(T(R))]$$

where R is a data object representing the center of the respective ball region, r_R is a *covering radius* of the ball, $\delta(R, Par(R))$ is so-called *to-parent* distance (the distance from R to the object P of the parent routing entry), and finally $ptr(T(R))$ is a pointer to the entry's subtree $T(R)$. In order to correctly bound the data in $T(R)$'s leaves, the routing entry must satisfy the *nesting condition*: $\forall O_i \in T(R), r_R \geq \delta(R, O_i)$. The data is stored in the leaves of M-tree. Each leaf contains *ground entries*

$$grnd_M(D) = [D, id(D), \delta(D, \mathrm{Par}(D))]$$

where D is the data object itself (externally identified by $id(D)$), and $\delta(D, Par(D))$ is, again, the to-parent distance. See an example of entries in Figure 1a.

The queries are implemented by traversing the tree, starting from the root. Those nodes are accessed, the parent regions of which are overlapped by the query region, e.g., by a range query ball (Q, r_Q). The check for region-and-query overlap requires an explicit distance computation $\delta(R, Q)$ (called *basic filtering*). In particular, if $\delta(R, Q) \leq r_Q + r_R$, the data ball (R, r_R) overlaps the query

(Q, r_Q), thus the child node has to be accessed, see Figure 1b. If not, the respective subtree is filtered from further processing. Moreover, each node in the tree contains the distances from the routing/ground entries to the center of its parent routing entry (the to-parent distances). Hence, some of the M-tree branches can be filtered without the need of a distance computation, thus avoiding the "more expensive" basic overlap check. In particular, if $|\delta(P, Q) - \delta(P, R)| > r_Q + r_R$, the data ball R cannot overlap the query ball (called *parent filtering*), thus the child node has not to be re-checked by basic filtering, see Figure 1c. Note $\delta(P, Q)$ was already computed at the unsuccessful parent's basic filtering.

PM-tree. The idea of PM-tree [12, 14] is to enhance the hierarchy of M-tree by an information related to a static set of p global pivots $P_i \in \mathcal{P} \subset \mathbb{U}$. In a PM-tree's routing entry, the original M-tree-inherited ball region is further cut off by a set of *rings* (centered in the global pivots), so the region volume becomes more compact – see Figure 2a. Similarly, the PM-tree ground entries are enhanced by distances to the pivots, which are interpreted as rings as well. Each ring stored in a routing/ground entry represents a distance range (bounding the underlying data) with respect to a particular pivot. A routing entry in PM-tree inner node is defined as:

$$rout_{PM}(R) = [R, r_R, \delta(R, \text{Par}(R)), ptr(T(R)), \text{HR}],$$

where the new HR attribute is an array of p_{hr} intervals ($p_{hr} \leq p$), where the t-th interval HR_{P_t} is the smallest interval covering distances between the pivot P_t and each of the objects stored in leaves of $T(R)$, i.e. $\text{HR}_{P_t} = \langle \text{HR}_{P_t}^{min}, \text{HR}_{P_t}^{max} \rangle$, $\text{HR}_{P_t}^{min} = min\{\delta(O_j, P_t)\}$, $\text{HR}_{P_t}^{max} = max\{\delta(O_j, P_t)\}$, $\forall O_j \in T(R)$. The interval HR_{P_t} together with pivot P_t define a ring region (P_t,HR_{P_t}); a ball region (P_t,$\text{HR}_{P_t}^{max}$) reduced by a "hole" (P_t,$\text{HR}_{P_t}^{min}$). A ground entry in PM-tree leaf is defined as:

$$grnd_{PM}(D) = [D, id(D), \delta(D, \text{Par}(D)), \text{PD}],$$

where the new PD attribute stands for an array of p_{pd} pivot distances ($p_{pd} \leq p$) where the t-th distance $\text{PD}_{P_t} = \delta(R, P_t)$.

The combination of all the p entry's ranges produces a p-dimensional minimum bounding rectangle (MBR), hence, the global pivots actually map the metric regions/data into a "pivot space" of dimensionality p (see Figure 2b).

When issuing a range or kNN query, the query object is mapped into the pivot space – this requires p extra distance computations $\delta(Q, P_i), \forall P_i \in \mathcal{P}$. The mapped query ball (Q, r_Q) forms a hyper-cube $\langle \delta(Q, P_1) - r_Q, \delta(Q, P_1) + r_Q \rangle \times \cdots \times \langle \delta(Q, P_p) - r_Q, \delta(Q, P_p) + r_Q \rangle$ in the pivot space that is repeatedly utilized to check for an overlap with routing/ground entry's MBRs (see Figures 2a,b). If they do not overlap, the entry is filtered out without any distance computation, otherwise, the M-tree's filtering steps (parent & basic filtering) are applied.

Note the MBRs overlap check does not require an explicit distance computation, so the PM-tree usually achieves significantly lower query costs when compared with M-tree – up to an order of magnitude (see [12–14]).

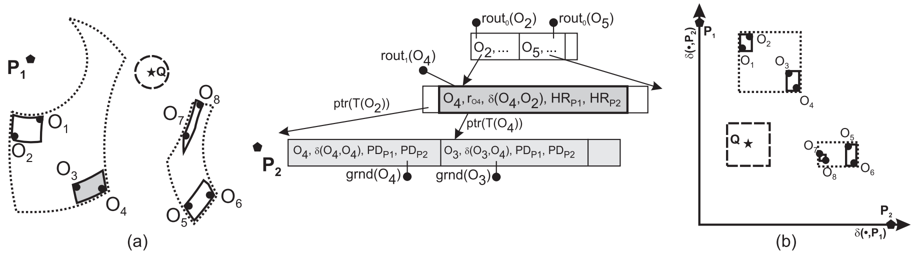

Fig. 2. (a) PM-tree employing 2 pivots (P_1, P_2). (b) Projection of PM-tree into the "pivot space".

1.3 Paper contributions

In this paper we introduce metric skyline processing by use of PM-tree. We follow the pioneer work [3] where the concept of metric skyline query was introduced, and its implementation utilizing M-tree was proposed. In Section 2 the metric skyline query and its original implementation is discussed, while in Section 3 we propose our original PM-tree implementation of metric skyline processing. In experimental results (Section 4) we show that PM-tree based metric skyline processing outperforms the original M-tree implementation not only in terms of distance computation costs, but also in terms of I/O costs, internal CPU costs and internal space costs.

2 Metric skyline queries

In relational databases, the multi-criterial retrieval is popular in situations where a query exactly specifying the desired attribute ranges cannot be effectively issued. Instead, there is a need for a simplified query concept which selects the desired database objects by some aggregation technique. Besides the top-k queries [6], a popular multi-criterial technique is the *skyline operator* [1].

2.1 The Skyline Operator

The traditional skyline operator is an advanced retrieval technique that selects objects from a multidimensional database that are "the best" from the global point of view. The only assumption on the database is that the attribute domains (dimensions) are linearly ordered, such that the lower (or higher) value of an attribute is, the better the object is (in that attribute). In the rest of the paper we use the convention that a lower value in an attribute is better.

The skyline operator selects all objects from the database (the *skyline set*), that are not *dominated* by any other object. An object O_1 dominates another object O_2 if at least one of O_1's attribute values is lower than the same attribute in O_2, and the other attribute values in O_1 are lower or equal to the corresponding attribute values in O_2. Hence, O_1 is the *dominating* object, while O_2 is the *dominated* object. In Figure 3a see an example of skyline set consisting of 5 objects, dominating the remaining 6 objects.

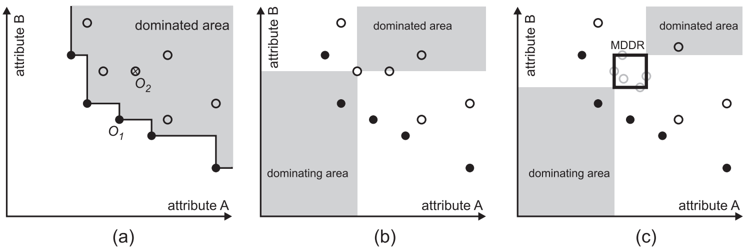

Fig. 3. (a) Skyline set and the dominated objects. A dominating-dominated (a) object and (b) rectangle (MDDR).

Skyline Processing. There exist many approaches to the efficient implementation of the skyline operator, while we outline two of them – the *Sort-First Skyline* algorithm [1] and the *branch-and-bound* algorithm which will be useful further in the paper.

In the *Sort-First Skyline* algorithm, the database objects O_i are just ordered ascendentally based on the L_1 norm on attributes (coordinates) of O_i, i.e., $||O_i||_{L_1} = O_i^1 + O_i^2 + \cdots + O_i^n$. Then, following the L_1 order, the sorted database is passed such that each visited object O_i is checked whether it is dominated by the already determined skyline objects. If O_i is not dominated, it is added to the skyline set (empty at the beginning), otherwise, O_i is ignored. After the one-pass database traversal is finished, the skyline set is complete. The algorithm is correct because of the L_1-norm ordering. Suppose an object O_i is being processed (see Figure 3b). Because every object possibly dominating O_i lies in the dominating area, its L_1 norm must be lower than that of O_i. However, such an object has already been visited (and possibly added to the skyline set) because of the ordered database traversal. Thus, O_i can be either safely added to the skyline set or filtered out.

The *branch-and-bound* approach employs a *spatial access method* (SAM), e.g. the R-tree [8]. The database is indexed by the SAM, while for the skyline processing a memory-resident *priority heap* is additionally utilized. The heap priority is defined, again, as the L_1 norm, however, besides the database objects themselves, the heap may contain also minimum bounding rectangles (MBRs, natively maintained by, e.g., R-tree). For future use outside the scope of SAM, we call MBRs as *minimum dominating-dominated rectangles* (MDDRs). The MDDRs serve as spatial rectangular approximations of the underlying database objects (or nested MDDRs), while they can be effectively used for filtering. The order of an MDDR within the heap is defined by the L_1 norm of its minimal corner (the point of MDDR with minimal values in all dimensions), which is the maximal lower bound to L_1 norm of any object inside the MDDR.

2.2 Metric Skyline Queries

The spatial skyline queries were generalized recently to support an arbitrary metric distance δ (i.e., not just Euclidean), constituting thus the *metric skyline queries* (MSQ) [3, 4].

Generally speaking, the metric skyline model just adds an abstract transformation step before the usual skyline processing. The step consists of transformation of a database in a metric space into database in m-dimensional vector space through a set $\mathcal{Q}$ of $m = |\mathcal{Q}|$ query examples. In the second step, the traditional skyline operator is performed on the transformed database. In particular, a database object O_i in the metric space is transformed into a vector V_i, where its j-th coordinate is defined as the distance from j-th query to O_i, i.e., $V_i = \langle\delta(Q_1, O_i), \delta(Q_2, O_i), \ldots, \delta(Q_m, O_i)\rangle, Q_j \in \mathcal{Q}$.

Motivation. The motivation for MSQ can be seen in the insufficient expressive power of range and kNN queries, as mentioned in Section 1.1. Besides the possibility of employing multiple query examples, the metric skyline query has also another unique property, the absence of query extent, i.e., the query is defined just by the set $\mathcal{Q}$. This property could be seen as both advantage and disadvantage. The advantage is that metric skyline query returns all distinct objects from the database that are as similar to the query examples as possible. Hence, we obtain all such objects; we are freed from tuning the precision and recall proportion. Unfortunately, the disadvantage of MSQ is the skyline set (answer set) size. If $m = |\mathcal{Q}| = 1$ we obtain a regular 1-NN query. However, with increasing m the skyline size usually grows substantially, while a skyline set size exceeding several percent of the database is usually useless for an end-user. Thus, to be discriminative enough, the metric skyline query should employ only a few query examples (say, 2–5).

M-tree Based Implementation. The above described straightforward two-step abstraction is not suitable for implementation of MSQ. An explicit transformation of the original database $\mathbb{S}$ into a metric space would require expensive static preprocessing of the database, consisting of $|\mathcal{Q}| \cdot |\mathbb{S}|$ distance computations, extra storage costs, etc. Remember, the main cost component in similarity search by MAMs is the number of distance computations, so any MSQ algorithm should be designed to avoid computing as many distances as possible.

The authors of metric skyline queries proposed a native MSQ processing by M-tree [3, 4], where the transformation step was applied only on a part of the database that cannot be skipped during the processing. Basically, the M-tree based metric skyline algorithm was inspired by the traditional skyline processing by R-tree and the priority heap $\mathcal{H}$ under L_1 norm (as described in Section 2.1).

In the following we have re-formulated the original description in [3, 4] to the more abstract MDDR formalism, due to its easier extensibility to our original contribution in Section 3. The modification of R-tree based skyline processing to the metric case resides in an "on-the-fly" derivation of MDDRs, which cover the transformed data objects. Instead of "native" R-tree MDDRs (MBRs, resp.), we distinguish two types of derived MDDRs in M-tree, as follows:

(1) The *Par-MDDR* (parent MDDR) of a routing/ground entry $entry(R, r_R, \cdots)$, constructed by use of the parent routing entry $rout(P, \cdots)$ as $\text{MDDR}_{Par} = \langle LB^{Q_1}_{Par}, UB^{Q_1}_{Par}\rangle \times \cdots \times \langle LB^{Q_m}_{Par}, UB^{Q_m}_{Par}\rangle$, where $LB^{Q_i}_{Par}$ is a lower-bound distance from Q_i to the region (R, r_R) (through its parent P), while $UB^{Q_i}_{Par}$ is an upper-bound distance from Q_i to (R, r_R). Thus, $LB^{Q_i}_{Par} = max(\delta(Q_i, P) - (\delta(P, R) + r_R), (\delta(P, R) - r_R) - \delta(Q_i, P), 0)$, and $UB^{Q_i}_{Par} = \delta(Q_i, P) + \delta(P, R) + r_R$.

(2) The *B-MDDR* (basic MDDR), constructed directly from a routing/ground entry as $\text{MDDR}_B = \langle \delta(Q_1, R) - r_R,\ \delta(Q_1, R) + r_R\rangle \times \cdots \times \langle \delta(Q_m, R) - r_R, \delta(Q_m, R) + r_R\rangle$. In consequence, B-MDDR of ground entry is a single point.

Obviously, we have chosen the terms “Par-MDDR” and “B-MDDR” due to the analogy with parent- and basic filtering used when processing a range or kNN query in M-tree. The Par-MDDR of a routing/ground entry can be derived without an explicit distance computation; the $\delta(Q_i, P)$ distances were already computed during the top-down M-tree traversal. The derivation of B-MDDR is more expensive, it requires m computations of $\delta(R, Q_i), \forall Q_i \in \mathcal{Q}$.

An MDDR M_1 dominates all objects inside an MDDR M_2 if the L_1 norm of M_1’s *maximal corner* is lower than the L_1 norm of M_2’s *minimal corner*, where a max/min corner is the point with max/min values in all dimensions of an MDDR. For an example of Par-MDDR and B-MDDR, see Figure 4.

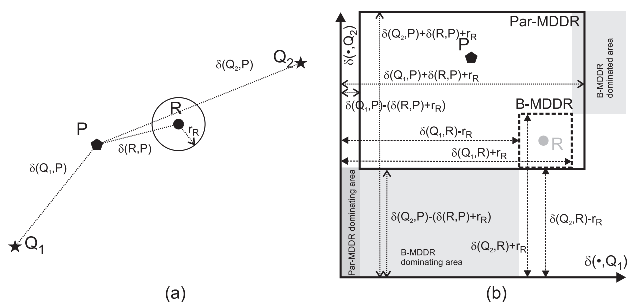

Fig. 4. (a) Metric space with M-tree regions (b) Transformed vector space with MDDRs

The MSQ algorithm starts by inserting routing entries from the M-tree root into the heap $\mathcal{H}$. The heap keeps order given by L_1 norm applied on the entries’ B-MDDRs’ minimal corners. Then a loop follows until the heap gets empty:

(1) An entry $entry(R, \dots)$ with the lowest L_1 value of its B-MDDR is popped from the heap.
(2) If the entry is a ground entry, it is added to the set of skyline objects. All entries on the heap which are dominated by this new skyline object are removed. Jump to Step 1.
(3) If the entry is a routing entry, the entry’s child node is fetched. The Par-MDDRs of the child node’s entries are checked for dominance by the set of already determined skyline objects, while the dominated ones (and the respective subtrees, in case of routing entries) are filtered from further processing.

(4) The B-MDDRs of the non-filtered child entries are derived. Those entries not dominated by the already retrieved skyline set are inserted into the heap. Jump to Step 1.

Discussion. Unfortunately, in the original contribution [3, 4] the cost analysis and also the experiments were focused solely on measuring the number of dominance checks, i.e., how many times B-MDDRs and Par-MDDRs were checked for dominance by a skyline object. The authors completely ignored the number of distance computations (the crucial performance factor for any MAM), but also the heap size and the number of operations on heap, spent by running the metric skyline algorithm on M-tree.

As we present later in experimental evaluation, the M-tree based algorithm, as proposed in [3, 4], is extremely inefficient in terms of the heap size and the number of operations on the heap. In fact, the maximal heap size could reach the size of the database, making such an implementation inapplicable in database environments. In the following section we introduce our PM-tree based method, which not only decreases the number of distance computations spent for metric skyline processing, but also drastically decreases the maximal heap size and the number of operations on the heap.

3 PM-tree based metric skyline

The M-tree based approach to metric skyline processing can be extended to a PM-tree based implementation. In the following we introduce an algorithm that makes use of the PM-tree's extensions over the M-tree – the pivot set $\mathcal{P}$ and the respective ring regions maintained by routing/ground entries in PM-tree nodes (for PM-tree details see Section 1.2).

First of all, when a metric skyline query is started, a *query-to-pivot* matrix of pair-wise distances between the PM-tree pivots $P_i \in \mathcal{P}$ and query examples $Q_i \in \mathcal{Q}$ is computed. The PM-tree based algorithm (see Section 3.4) then utilizes the following three filtering concepts (Sections 3.1–3.3).

3.1 Deriving Piv-MDDRs

Besides the M-tree's B-MDDRs and Par-MDDRs derived from a routing/ground $entry(R, \cdots, \mathrm{HR/PD})$, an additional MDDR can be derived from the set of rings HR/PD maintained by the entry, called *Piv-MDDR* (pivot MDDR). The Piv-MDDR can be derived using the query-to-pivot matrix, as
$\mathrm{MDDR}_{Piv} = \langle LB^{Q_1}_{Piv}, UB^{Q_1}_{Piv}\rangle \times \cdots \times \langle LB^{Q_m}_{Piv}, UB^{Q_m}_{Piv}\rangle$, where
$\mathrm{LB}^{Q_i}_{Piv} = max_{P_j \in \mathcal{P}}\{\delta(P_j, Q_i) - \mathrm{HR}^{max}_{P_j}, \mathrm{HR}^{min}_{P_j} - \delta(P_j, Q_i), 0\}$, and
$\mathrm{UB}^{Q_i}_{Piv} = min_{P_j \in \mathcal{P}}\{\delta(P_j, Q_i) + \mathrm{HR}^{max}_{P_j}\}$.

Similarly as the M-tree's Par-MDDR, the derivation of Piv-MDDR requires no extra distance computation, however, Piv-MDDRs are much more compact than Par-MDDRs. This results in more effective filtering of routing/ground entries by skyline objects or some dominating MDDRs. Moreover, the Piv-MDDR is often even more compact than the direct B-MDDR, because the PM-tree's rings reduce the volume of the original M-tree's sphere. In Figure 5 see an example of Piv-MDDR, Par-MDDR and B-MDDR, when 2-pivot PM-tree and 2 query examples are used.

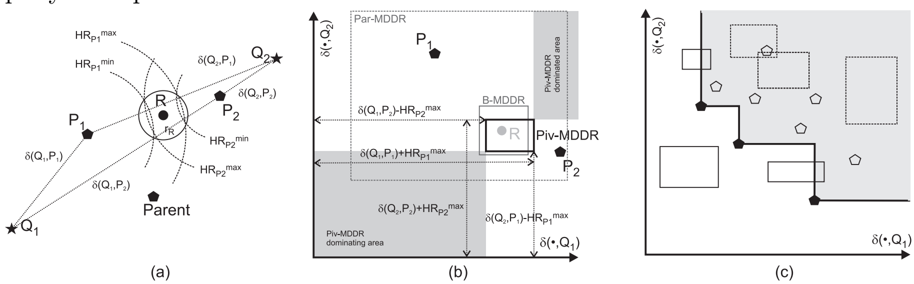

Fig. 5. A PM-tree routing entry in (a) metric space and (b) mapped to Piv-, Par-, and B-MDDR. (c) A pivot skyline.

3.2 Pivot-Skyline filtering

If the pivots P_i come from the database (i.e., $P_i \in \mathcal{P} \subset \mathbb{S}$), the MDDRs that are about to be inserted into the heap can be checked for a dominance by the pivots. Since the query-to-pivot matrix is computed at the beginning of every metric skyline query processing, the transformation of the pivots into the "query space" requires no additional distance computations. Moreover, to reduce the number of pivots used for dominance checking, we can determine the so-called *pivot skyline* – those pivot objects, which constitute a metric skyline within the pivot set $\mathcal{P}$ itself, see an example in Figure 5c.

The filtering by use of pivot skyline is beneficial in the early phase of the metric skyline processing, when the set of determined skyline objects is still empty. In the experiments we show that such an early phase is the dominant phase of the entire skyline processing – 80-90% of the total distance computations is performed before the first skyline object is found. Hence, pruning the heap by use of the pivot skyline greatly helps to reduce the heap size and, consequently, the number of operations on the heap. **Note:** As the number of determined skyline objects grows, the objects in the pivot skyline become dominated by the "regular" skyline objects. Hence, in order to effectively use the pivots for dominance checking, we keep just those pivots in the pivot skyline, that are not dominated by the already determined skyline objects. Thus, at the moment when all skyline objects are known, the pivot skyline becomes empty.

3.3 Deferred heap processing

In the original M-tree algorithm, the priority heap contains just L_1-ordered B-MDDRs (together with the associated routing/ground entries). When an entry is to be inserted into the heap, its B-MDDR must be determined, see Steps 3,4 of the algorithm in Section 2.2. We call this approach a *non-deferred heap processing.*

However, the non-deferred heap processing is not optimal in terms of the number of distance computations. In order to save some distance computations, we propose the *deferred heap processing* for the metric skyline, inspired by the Hjaltason's & Samet's incremental nearest neighbor algorithm, which is optimal in the number of distance computations [9]. The modified heap is generalized such that it may contain not only B-MDDRs of routing/ground entries, but also the intersections of their Piv-MDDR and Par-MDDR (denoted as Piv-MDDR ∩ Par-MDDR). The deferred heap processing then deals with two situations:

(1) An entry equipped by B-MDDR is popped from the heap. Then,
(a) If the entry is a ground entry, it becomes a skyline object.
(b) If the entry is a routing entry, its child node is fetched, while for every entry in the child node the Piv-MDDR ∩ Par-MDDR is checked for a dominance by the skyline set. Every not-dominated child entry is equipped by its Piv-MDDR ∩ Par-MDDR and inserted into the heap.
(2) An entry equipped by Piv-MDDR ∩ Par-MDDR is popped from the heap and checked for a dominance by the skyline set. If not dominated, the entry's B-MDDR is determined and, if still not dominated, inserted back into the heap.

Listing 1 *(Algorithm of PM-tree based metric skyline query)*

```
MSQuery()
{
Input: PM-tree PM, query points Q, type ('M-tree', 'PM-tree',
                              'PM-tree+PSF', 'PM-tree+PSF+DEF')
Output: Result MSS containing skyline points


  if (type is not 'M-tree')
    P2Q_DM = evaluate the query-to-pivot matrix
    // pivots must be DB objects
    PSL = evaluate pivot skyline (using P2Q_DM)
  Insert all routing entries + their Piv-MDDR ∩ B-MDDR from the
                                   PM-tree root into the heap H
  while (H is not empty)
    currentEntry = pop entry from the heap H
    if (currentEntry is not equipped by 'B-MDDR')
      FilterAndInsert(currentEntry, currentEntry, type, true)
    else if (currentEntry is of type 'ground entry' and is equipped by
                                                          'B-MDDR')
      Insert currentEntry into MSS
      H.FilterDominatedObjectsBy(currentEntry.MDDR)
      PSL.FilterDominatedObjectsBy(currentEntry.MDDR)
    else
      N = fetch child node of currentEntry
      for each childEntry in N
        FilterAndInsert(childEntry, currentEntry, type, false)
}

FilterAndInsert(newEntry, parentEntry, type, deferred)
{
  if (not deferred)
    Equip newEntry by its Par-MDDR
    if (type is not 'M-tree')
      Update newEntry.MDDR by intersection with newEntry's Piv-MDDR
  if (Filter(newEntry, type))
    return
  if (type = 'PM-tree+PSF+DEF' and not deferred)
    Insert newEntry into H
    return
  Equip newEntry by its B-MDDR
  if (Filter(newEntry, type))
    return
  Insert newEntry into H
}

Filter(newEntry, type)
{
  for each O_i in MSS
    if (newEntry.MDDR is dominated by O_i)
      return true
  if (type is 'M-tree' or 'PM-tree')
    return false
  for each O_i in PSL
    if (newEntry.MDDR is dominated by O_i)
      return true
  return false
}
```

3.4 The algorithm

In Listing 1 the algorithm for metric skyline query is presented, including the original M-tree variant as well as the proposed PM-tree extensions.

The input attribute type allows to set the MSQ variant as follows: type = 'M-tree' is the original M-tree based algorithm, type = 'PM-tree' is the basic PM-tree based algorithm using the Piv-MDDR filtering (as described in Section 3.1), type = 'PM-tree+PSF' additionally uses the pivot-skyline filtering (as described in Section 3.2), and type = 'PM-tree+PSF+DEF' additionally uses the deferred heap processing (as described in Section 3.3).

4 Experimental evaluation

We performed an extensive experimentation with the three new variants of the PM-tree based metric skyline processing, comparing them against the original M-tree based method. Instead of the number of dominance checks (as included in the original contribution [3, 4]), we have observed other 4 measures of costs spent by the MSQ processing – the number of distance computations, the number of operations on the heap, the maximal allocated size of the heap, and finally the I/O costs.

In addition to the absolute numbers presented in the figures below, we also relate the number of distance computations spent by (P)M-tree MSQ processing to the costs of MSQ processed by simple *sequential search*, which takes $|\mathcal{Q}| \cdot |\mathbb{S}|$ distance computations for every query.

4.1 The testbed

We have used two databases, a subset of the *CoPhIR* database [7] of MPEG7 image features extracted from images downloaded from `flickr.com`, and a synthetic database of polygons. The CoPhIR database, consisting of one million feature vectors, was projected into two subdatabases, the *CoPhIR_12* database, consisting of 12-dimensional color layout descriptors, and the *CoPhIR_76* database, consisting of 76-dimensional descriptors (12-dimensional color layout and 64-dimensional color structure). As a distance function the Euclidean (L_2) distance was employed.

The *Polygons* database was a synthetic randomly generated set of 250,000 2D polygons, each polygon consisting of 5–15 vertices. The Polygons should serve as a non-vectorial analogy to clustered points. The first vertex of a polygon was generated at random. The next one was generated randomly, but the distance from the preceding vertex was limited to 10% of max. distance in the space. We used the Hausdorff distance [10] for measuring the distance between two polygons, so here a polygon could be interpreted as a cloud of points.

4.2 Experiment settings

The query costs were always averaged for 200 metric skyline queries, while the query examples followed the distribution of database objects. As the parameters we observed various database sizes, the (P)M-tree node capacities, the number of query examples, and the number of PM-tree leaf pivots. The (P)M-tree node capacities ranged from 20 to 40 routing/ground entries, the index sizes took 200MB–2GB, the P(M)-tree heights were 3–5 (4–6 levels). The minimal (P)M-tree node utilization was set to 20% of node capacity. The number of PM-tree leaf pivots ranged from 30 to 1000, while the number of inner pivots ranged from 15 to 500. Unless otherwise stated, the number of MSQ query examples was 2, the (P)M-tree node size was 20, the number of leaf pivots was 1000 for CoPhIR and 300 for Polygons (the number of inner pivots was half the number of leaf pivots).

4.3 The results

In the first set of experiments, the number of PM-tree leaf pivots was increasing, see Figure 6. When considering Polygons database, the M-tree's MSQ got to 17% of distance computations needed by simple sequential search on the Polygons database. However, for the highest number of pivots the PM-tree's MSQ reduced the M-tree costs by another 35%. The heap size required by PM-tree reached only up to one third of the heap size required by the M-tree. The impact of pivot-skyline filtering (the +PSF(+DEF) variants) on the maximal heap size was significant.

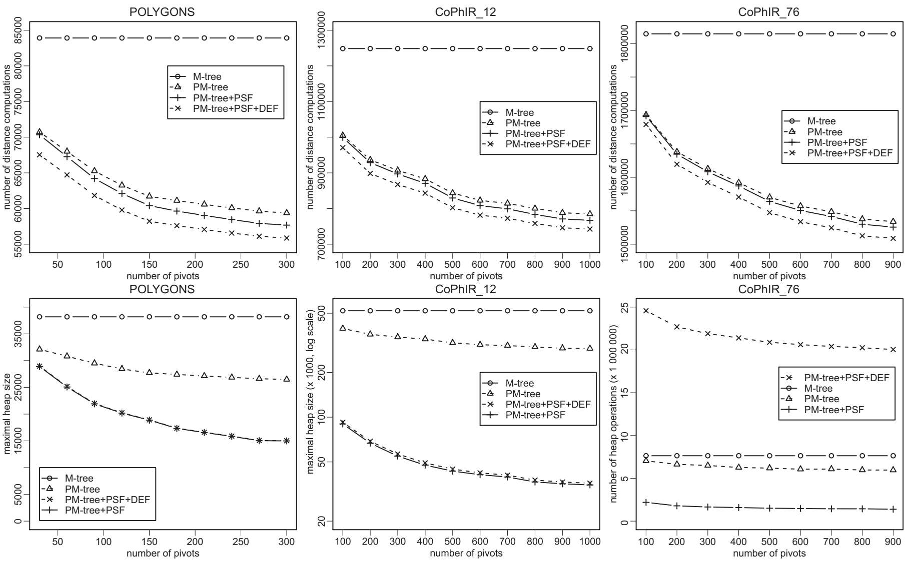

Fig. 6. Increasing number of pivots: distance computations, maximal heap size

The same situation is presented for the Cophir_12 database. The results are even better as for Polygons – the number of distance computations for PM-tree+PSF+DEF variant was reduced to 60% of M-tree costs, while the maximal heap size was reduced down to 8% of the heap size required by M-tree (note the log.scale in the figure).

Finally, the same situation is presented for the high-dimensional Cophir_76 database. Because of the high dimensionality, the M-tree performance was poor – it got to 91% distance computations required by simple sequential search. The PM-tree performed better, achieving 75% of the sequential search's distance computations. The PM-tree+PSF+DEF variant performs poorly when looking at the number of heap operations, due to the deferred heap processing, i.e., repeated insertions of MDDRs into the heap (see Section 3.3). On the other hand, the +DEF variant steadily achieves the lowest distance computation costs (as expected). The PM-tree+PSF variant performs the best, achieving 25% of the heap operations spent by M-tree.

The second set of experiments focused on the increasing database size. In Figure 7 the results for Cophir_76 database are presented. The trend of increasing distance computations is obvious for all MSQ processing methods. However, the situation is dramatically different for the number of heap operations and the maximal heap size, where the PM-tree+PSF beats the M-tree by a factor of 17 in heap operations, and by a factor of 7 in the maximal heap size. On the other hand, PM-tree+PSF+DEF suffers from a high number of heap operations.

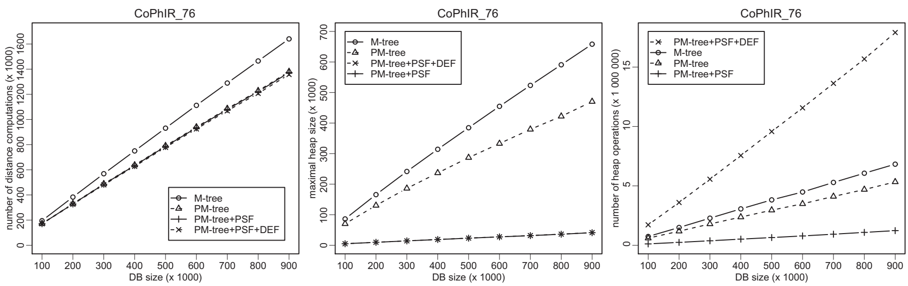

Fig. 7. Increasing size of Cophir_76 database: (a) Distance computations (b) Maximal heap size (c) Heap operations

In the third set of experiments, the results for increasing number of query examples used in metric skyline queries are presented on the Cophir_12 database, see Figure 8. Because the number of skyline objects grows substantially with the increasing number of query examples (retrieving 50, 400, 1750, 4570 skyline objects for 2-, 3-, 4-, and 5-example MSQs), the overall MSQ costs grow substantially as well. Nevertheless, the PM-tree MSQ processing is still much cheaper than the M-tree in the heap size and operations, even for 5 query examples. However, note that for 5 query examples the distance computations of all the methods come close to the costs of simple sequential search.

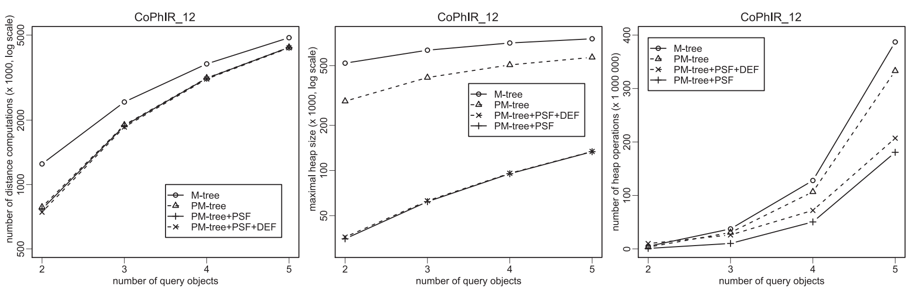

Fig. 8. Increasing number of query examples: (a) Distance computations (b) Maximal heap size (c) Heap operations

Although the I/O costs do not represent a dominant performance component in similarity search[1], in the last experiment we present the I/O costs as a supplementary result (CoPhIR_12, 2 query examples). In particular, in Figure 9a we give the numbers of logical seeks[2] spent by skyline processing (the seek operation is the most expensive one when fetching a page/PM-tree node from the disk). The PM-tree based MSQ processing spent just 64% of seek operations required by the M-tree. As for the distance computation costs, also the I/O costs were decreasing with increasing number of pivots.

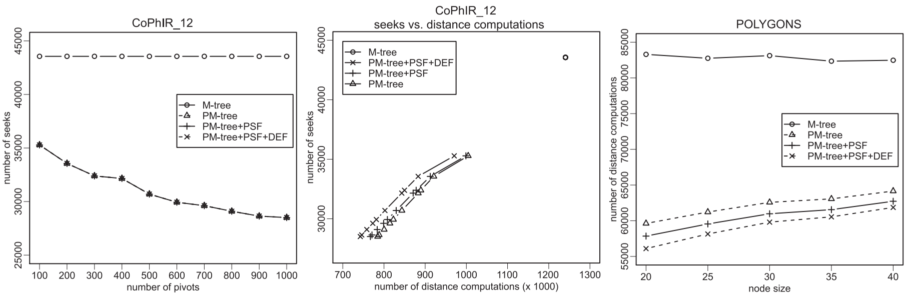

Fig. 9. Increasing number of pivots: (a) I/O costs (b) I/O costs vs. distance computations. (c) Increasing size of (P)M-tree nodes.

In Figure 9b the I/O costs vs. computation costs are shown. As in the first chart, the pairs ⟨I/O costs, distance computations⟩ were obtained for different numbers of pivots employed by PM-tree. Since the (P)M-tree indices consisted of 79,584 nodes, note that the I/O costs correspond to fetching 55% of all the index nodes for M-tree and 35% for PM-tree (1000 pivots). Also note there is linear correlation between the distance computations and I/O costs. 55%. Finally, the Figure 9c shows the performance of (P)M-tree depending on the node size.

4.4 Summary

The experimentation with M-tree and PM-tree based metric skyline processing has shown that the PM-tree outperforms the M-tree implementation up to 2 times in the number of distance computations, almost 20 times in the number of heap operations and the maximal heap size, and almost 2 times in the I/O costs. The results for maximal heap size are exceptionally important, because a large size of the heap (which is a main-memory structure) would prevent from processing of metric skyline queries on very large databases.

5 Conclusions

In this paper we have proposed a PM-tree based implementation of metric skyline query, a recently proposed multi-example query concept suitable for advanced

[1] A single distance computation is generally supposed to be much more expensive than a single I/O operation.

[2] We did not consider any node caching in this experiment.

similarity search in multimedia databases. We have shown that the PM-tree based implementation of metric skylines significantly outperforms the existing M-tree based implementation in all observed costs – the time, space, and I/O costs.

Acknowledgments. This research was supported by Czech Science Foundation (GAČR) Project 201/09/0683 and by institutional research plan MSM0021620838.

References

1. S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430, Washington, DC, USA, 2001. IEEE Computer Society.
2. E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
3. L. Chen and X. Lian. Dynamic skyline queries in metric spaces. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 333–343, New York, NY, USA, 2008. ACM.
4. L. Chen and X. Lian. Efficient processing of metric skyline queries. *IEEE Trans. on Knowl. and Data Eng.*, 21(3):351–365, 2009.
5. P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB'97*, pages 426–435, 1997.
6. R. Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999.
7. F. Falchi, C. Lucchese, R. Perego, and F. Rabitti. CoPhIR: COntent-based Photo Image Retrieval [http://cophir.isti.cnr.it/CoPhIR.pdf], 2008.
8. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In B. Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.
9. G. Hjaltason and H. Samet. Incremental similarity search in multimedia databases, computer science dept. tr-4199, univ. of maryland, college park, 2000.
10. D. Huttenlocher, G. Klanderman, and W. Rucklidge. Comparing images using the hausdorff distance. *IEEE Patt. Anal. and Mach. Intell.*, 15(9):850–863, 1993.
11. H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann, 2006.
12. T. Skopal. Pivoting M-tree: A Metric Access Method for Efficient Similarity Search. In *Proceedings of the 4th annual workshop DATESO, Desná, Czech Republic, ISBN 80-248-0457-3, also available at CEUR, Volume 98, ISSN 1613-0073,* `http://www.ceur-ws.org/Vol-98`, pages 21–31, 2004.
13. T. Skopal. Unified framework for fast exact and approximate search in dissimilarity spaces. *ACM Transactions on Database Systems*, 32(4):1–46, 2007.
14. T. Skopal, J. Pokorný, and V. Snášel. Nearest Neighbours Search using the PM-tree. In *DASFAA '05, Beijing, China*, pages 803–815. LNCS 3453, Springer, 2005.
15. Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 744–755. VLDB Endowment, 2004.
16. P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach (Advances in Database Systems).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.