The Geometric Framework for Exact and Similarity Querying XML Data

Michal Krátký[1], Jaroslav Pokorný[2], Tomáš Skopal[1], and Václav Snášel[1]

[1] Department of Computer Science, VŠB-Technical University of Ostrava, Czech Republic
michal.kratky@vsb.cz, jaroslav.pokorny@ksi.ms.mff.cuni.cz
[2] Department of Software Engineering, Charles University, Prague, Czech Republic
tomas.skopal@vsb.cz, vaclav.snasel@vsb.cz

Abstract. Using the terminology usual in databases, it is possible to view XML as a language for data modeling. To retrieve XML data from XML databases, several query languages have been proposed. The common feature of such languages is the use of regular path expressions. They enable the user to navigate through arbitrary long paths in XML data. If we considered a path content as a vector of path elements, we would be able to model XML paths as points within a multidimensional vector space. This paper introduces a geometric framework for indexing and querying XML data conceived in this way. In consequence, we can use certain data structures for indexing multidimensional points (objects). We use the UB-tree for indexing the vector spaces and the M-tree for indexing the metric spaces. The data structures for indexing the vector spaces lead rather to exact matching queries while the structures for indexing the metric spaces allow us to provide the similarity queries.

1 Introduction

Using the terminology usual in databases, it is possible to view XML as a language for data modelling. The notions like XML database and XML query language logically extend this idea [6,14]. So called native XML databases are implemented in increasing extent. To reach a quality of conventional relational databases, appropriate tools for manipulating have been designed. Among many attempts to query languages over XML data, the language XQuery [15] seems to be the leading approach now. The common feature of such languages is the use of regular path expressions. They enable the user to navigate through arbitrary long paths in XML data. Obviously, in the next step to XML databases some appropriate index structures have to be constructed for their data. Particularly, paths can be objects of indexing. In [9], we consider a path content as a vector of path elements. Then we can model XML paths as points within a multidimensional vector space. To speed-up access to such vectors, either various multidimensional trees (such as the R*-tree [4], X-tree [5] or UB-tree [2]), or metric trees can be used for their indexing (e.g., the M-tree [1] and the mvp-tree [7]). Only few these data structures have been used for indexing XML data. In

M.H. Shafazand and A M. Tjoa (Eds.): EurAsia-ICT 2002, LNCS 2510, pp. 35–46, 2002.

[9], we used UB-trees for indexing path contents for more efficient exact querying XML data. In this work we pursue a different, in some sense complementary, direction that is based on M-trees. Metric trees only require the distance between points to be a metric, thus they can be used even when no vector representation exists. We show how M-trees can be used for indexing XML paths and how similarity querying XML data can be supported. Section 2 introduces to us the geometric framework used in this paper. We shortly describe necessary basics of vector and metric spaces. Section 3 contains the vector model for indexing and querying XML data. The approach is based on the notion of path content. The main contribution of the paper – a similarity indexing XML data with M-tree – is contained in Section 4. We introduce briefly M-trees and propose a cumulated metric based in the Hamming metric for indexing XML paths. The section is completed with experimental evaluation of M-tree index applied on a real XML data set. In conclusions we summarize the approach.

2 Geometric Framework

In our approach to indexing and querying XML data we exploit the properties of two geometric models. Both of these models treat the XML data as objects/points within a space. In the first case within a *vector space* and in the second case within a *metric space*. As we will see, each of the models is suitable for a different purpose. We can say that they are complementary to each other.

There are two initial problems. First, we need to find a technique of transformation (so-called feature transformation) of the XML data into objects within a vector or metric space. Second, we need to find the data structures for storage and effective querying XML data according to the given model.

2.1 Vector Spaces

Vector model treats the XML data as points within multidimensional vector space. This approach allows us to index values and even the structure of XML documents and provides an ability of *exact matching* range queries. High vector space dimension (greater than approx. 20) is unfortunately associated with *curse of dimensionality* which has a negative influence on the range queries efficiency (see [3]). A representative data structure for the vector model is the UB-tree (see [2]). We discuss the vector model for indexing and querying XML data in Section 3.

2.2 Metric Spaces

In a metric space there are generally neither the dimension nor the vectors. However, in this paper we share the same representation of objects for the metric spaces and for the vector spaces – i.e. multidimensional points. An important difference is that each metric space has defined a metric – i.e. function measuring a distance (or similarity) between every two objects. This function d must satisfy following conditions:

$$d(o_i, o_i) = 0 \tag{1}$$
$$d(o_i, o_j) > 0 \qquad (o_i \neq o_j) \tag{2}$$
$$d(o_i, o_j) = d(o_j, o_i) \tag{3}$$
$$d(o_i, o_k) + d(o_k, o_j) \geq d(o_i, o_j) \tag{4}$$

The presence of the metric prompts that the metric model provides an ability of *similarity queries*. A representative data structure for the metric model is the M-tree, see section 4.

3 The Vector Model for Indexing and Querying XML Data

In our approach to indexing XML documents we model the XML data as points within multidimensional vector space and thus we can use certain index structures for multidimensional indexing (for example UB-tree). This approach was introduced in [9]. The data structures for indexing the vector spaces lead rather to *exact queries*.

We distinguish between indexing XML data with and without "mixed content" in [9]. Here we show only the latter case. The example of DTD for documents without "mixed content" and an XML document valid w.r.t. the DTD are in Figure 1a) and 1b), respectively. We will not consider the attributes of elements in our approach.

Example 1 (Querying XML document).
The example of the DTD and the valid XML document is in Figure 1. The path `accounts/account/name` denotes a query for obtaining all account customer names from the document.

```
<!DOCTYPE accounts [
  <!ELEMENT accounts (account*)>
  <!ELEMENT account (id, name)>
  <!ELEMENT id (#PCDATA)>
  <!ELEMENT name (#PCDATA)>
]>
```

a)

```
<?xml version="1.0" ?>
<accounts>
 <account><id>1234-8952</id>
   <name>Thomas Newell</name></account>
 <account><id>1234-4123</id>
  <name>David Moore</name></account>
 <account><id>5842-5321</id>
  <name>David Moore</name></account>
</accounts>
```

b)

Fig. 1. a) An example of DTD for XML documents without "mixed contents". b) An example of valid XML document without "mixed contents".

3.1 Indexing Path Contents and XML Structure

In our approach to indexing XML documents, we consider the *n-dimensional points representing path contents for XML structure indexing* of all paths from the root to all its leafs. The dimension n of the space is equal to the length of the maximal path in XML-tree, i.e. the number of edges from the root to its leaf element. To estimate the number n from DTD, we will consider only the "nonrecursive" DTDs in our approach.

Definition 1 (path content).
*Given a path $e = e_1/e_2/\ldots/e_k$, $e \in \mathcal{X}_P$, $\mathcal{X}_P$ is **set of paths**, the **path content** is defined as a sequence of string values $s = s_1/s_2/\ldots/s_k$, $s \in \mathcal{X}_{PC}$, $\mathcal{X}_{PC}$ is **set of path contents**. Each s_i, except s_k, can be empty (ϵ).*

Because string values can have a different length, it is necessary to use a procedure, which maps different strings into binary numbers of the same length. We use the signatures in our approach (e.g. [10]). The main idea of signatures is to reflect the data items into bit patterns and store them in a separate file which acts as a filter to eliminate the non-qualifying data items for an information request. We will denote the function generating signatures by $\mathrm{sig}(x)$, where x is a variable of string type.

The XML document is represented by m points within n-dimensional space, where m is the considered number of path contents. All these points are inserted into any index structures for multidimensional indexing. All complete paths contents are stored in other data structures. It is important to create binding between the elements of XML document having the same parent. We can create this binding using the elements unique numbers in the point representing path content for XML structure indexing. Of course, it is possible to index even paths (see Section 3.2).

Example 2 (Transformation of XML data to n-dimensional points).
We will show the transformation of the XML document from 1b) to the points of multidimensional space. We see the space has $n = 3$. We determine the length of the domains as 64b. This signature value is large enough for the signature s_i. But generally, there is not cause for domain cardinalities to be the same. The cardinality of domain for signatures of #PCDATA and for unique numbers of root elements can be different for example. The important role plays here the analysis of DTD.

If we are browsing through document in Figure 1b), then the following path contents are obtained: ϵ/ϵ/`1234-8952`, ϵ/ϵ/`Thomas Newell`, ϵ/ϵ/`1234-4123`, ϵ/ϵ/`David Moore`, ϵ/ϵ/`5842-5321` and ϵ/ϵ/`David Moore`. It is necessary to group these path contents according to the relationship to particular `accounts` and `account` elements. Therefore we nest the unique numbers of `accounts` and `account` elements into 1^{st} (2^{nd} respectively) coordinate of points representing path contents. The points representing path contents will be `(0,0,sig("1234-8952"))`, `(0,0,sig("Thomas Newell"))`, `(0, 1, sig("1234-4123"))`, `(0, 1, sig("David Moore"))`, `(0, 2, sig("5842-5321"))`, and `(0, 2, sig("David`

Moore")). The points in 3-dimensional space are depicted in Figure 3.1. These points are inserted into the indexing structure. If we index paths (see Section 3.2), then we will work with 4-dimensional space.

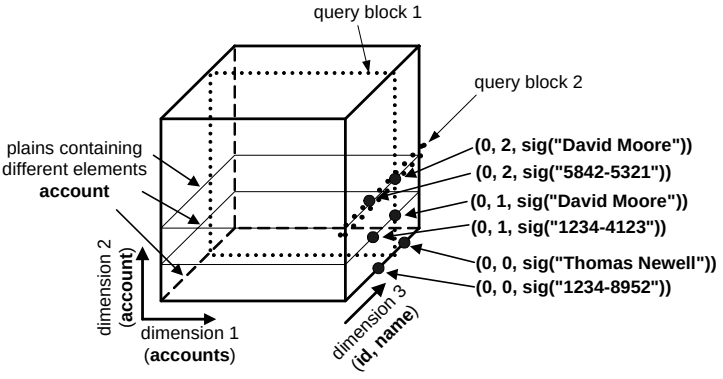

Fig. 2. The 3-dimensional space with indexed XML document without "mixed contents".

Example 3 (Querying XML document).
We show now how it is possible to query the XML document from Figure 1b) transformed to the points within multidimensional space by above mentioned technique. Let as take the query `accounts/account[name='David Moore']`, i.e. we want to get all `account` elements for `David Moore`'s account. First we need to transform this query to the *range query*. It means to find all the points from Figure 3.1, that are contained by *query block 1*. It is necessary to determine the coordinates of two points defining the query block. By means of range query we get the points from the 3-dimensional space which represent the unique numbers of parent elements of `name` element with content `David Moore`. We get the result set and if user will want to obtain the contents of child elements of any `account` element, for example, then the query block like *query block 2* from Figure 3.1 is effected for their retrieval. To distinguish the points representing the path contents for different paths it is necessary to index even the paths (see Section 3.2).

3.2 Indexing Paths

The indexing XML data as it is proposed in Section 3.1 considers only a path content. If the XML document is transformed to points of a space in this way, the element tags are lost. If we consider the XML document from Figure 1 then we will be not able to distinguish the points representing the path contents for paths `accounts/account/id` or `accounts/account/name`.

We consider a binary relation PPC [9] between paths and their path contents. All points representing paths will be inserted to other index structure. Besides

the point coordinates and pointers to data structures containing the whole paths we insert even the path unique numbers in another dimension of the space which contains the path contents. In fact, the relation PPC is built by adding other dimension to the space which contains path contents, i.e. the dimension of space will be $n+1$. It is hereby possible to index even the documents valid to different DTD in one index structure in this way.

Example 4 (Indexing paths).
We get two different paths `accounts/account/id` and `accounts/account/name` from the XML document in Figure 1b). So we get two points (we get two paths) representing paths in 3-dimensional space (paths contain three elements). These points are inserted into other indexing structure. The point `(sig("accounts"), sig("account"),sig("id"))` representing path `accounts/account/id` is inserted with unique number 0 and point representing path `(sig("accounts"), sig("account"),sig("name"))` with unique number 1. The points representing paths are in Figure 3. The points representing the path contents have last coordinate equal to the unique number of the associated path.

We see the space to have $n = 4$ (one dimension will be for unique numbers of paths). The gained path contents are in Example 2. Let as take the path content ϵ/ϵ`/1234-8952` and point representing the path contents `(0,0,sig("1234-8952"))` for example. The path unique number of path `accounts/account/id` from index structures which contain points representing paths is append as fourth coordinate to the point. We get point `(0,0,sig("1234-8952"),0)` in this manner. The all six points gained by the same way are inserted into index structure containing path contents.

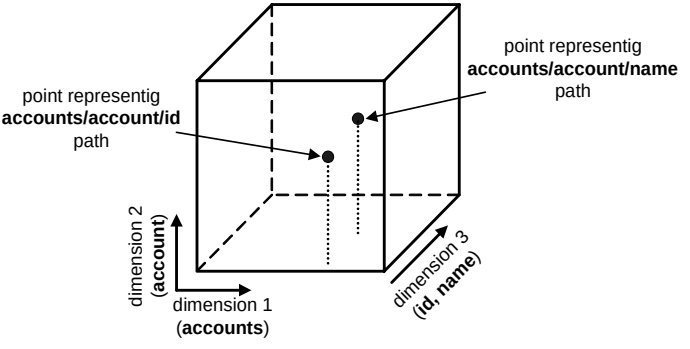

Fig. 3. The 3-dimensional space with points representing paths.

Example 5 (Querying XML document).
It is important to get by a point query the unique number of the desired path from index structure containing paths. After that we get desired points by a range query from the indexing structure containing path contents. It is necessary to

work with four dimensions in the case of defined coordinates of points determining the query block.

4 Similarity Queries

Another aspect of indexing XML data, in addition to the structural indexing, is the *similarity indexing*. In such an XML index we can query for XML objects that are similar to a query object.

Properties of metric spaces, where the metric represents the notion of similarity, are suitable formal basis for indexing similarities inside XML data. Following subsection describes a data structure M-tree which allows to index general objects of metric spaces.

4.1 M-Tree

Data structure M-tree (introduced in [1] and closely discussed in [13]) was developed for indexing and querying objects within metric spaces. Its main characteristic is that M-tree allows to process similarity queries. It is, in fact, dynamic, persistent, paged and balanced tree like e.g. the B-tree. The difference is in the semantics of the nodes. Indexed objects themselves, i.e. *ground objects*, lie in the leaf nodes. The inner nodes contain *routing objects* that represent a hierarchy of specific metric regions.

- The record of a routing object O_r in inner node contains:
  1. a ground object O_r (its significant properties respectively). This ground object determines the center of the metric region.
  2. pointer $ptr(T(O_r))$ to its own subtree $T(O_r)$ – i.e. *covering tree*
  3. value $r(O_r)$ – *covering radius* of the metric region
  4. value $d(O_r, P(O_r))$ – distance to the parent routing object $P(O_r)$

  Notes:
  The ground object in the routing object (inner node) is one of the ground objects remaining in the child leaf nodes of $T(O_r)$. The distance function d is a metric of a metric space.
- The record of a ground object looks similarly, but it also contains $oid(O_j)$ – identifier of the whole object (stored outside of M-tree) – instead of covering tree and covering radius.

Hierarchy of M-tree is based on partition of the metric space onto metric subregions which do not have to be strictly disjunct. This regions are formed by the routing objects O_r where the child routing objects (their regions respectively) and the child ground objects of its covering tree $T(O_r)$ are within the distance $r(O_r)$ to the center of O_r. Formally,

$$\forall O_i \in T(O_r), \qquad d(O_r, O_i) \leq r(O_r)$$

The precalculated distance value $d(O_r, P(O_r))$ to the parent object along with the covering radius $r(O_r)$ allow to eliminate the *untouched regions* from the process of an operation on M-tree (i.e. searching, insertion, deletion). Structure of the M-tree and the routing object relations are depicted in figure 4.

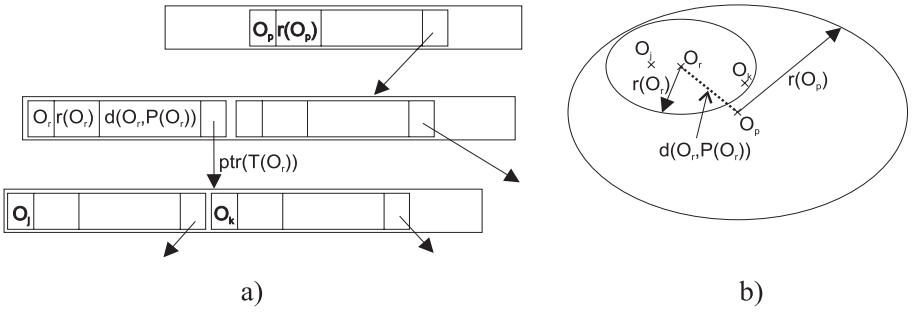

Fig. 4. (a) Nodes of M-tree contain object records. (b) Routing objects – metric regions.

Searching the M-tree. We must take into account two factors of complexity when we make some operation on the M-tree. The first one is the number of *accesses to disk pages* (number of regions being searched respectively) and the second one (specifically to M-tree) is the number of *distance calculations*. The goal is to minimize both these factors.

We can meet two kinds of queries by metric trees. The *range queries* search for all the objects within certain distance to the query object. The *k-nearest neighbours queries* search for the first k nearest objects to the query object. In both cases we can see a tendency to order the metric space – relative to the query object.

Managing the regions. The crucial factor of the M-tree's cost-effectiveness is a "good layout" of the metric regions stored within the M-tree. As we have said earlier, the regions can overlap another ones. This property arises from the M-tree's universality which is caused due to specifying only a metric of the metric space. High "overlap rate" leads, in the worst case, to sequential search – i.e. to linear complexity.

With the design of the M-tree there were also developed some techniques for minimizing this "overlap rate". The first technique is "embedded" into the phase of a tree node(page) splitting and consists of a choice of *split policy* and a mechanism of creating the best routing object – *promoting* phase. This is the dynamic technique. The second technique, more efficient, is the *bulk loading* algorithm. This algorithm takes at the beginning the whole collection of objects and loads all of them into the empty M-tree at once. The loading is based on preliminary clustering where prospective regions of objects are created at once. This is the static method.

Summary. M-tree is balanced, highly parametrizable data structure making possible to index objects of a metric space. The M-tree operations are performed with approximately logarithmic time complexity (if well build) but the M-tree doesn't represent a complete linear order like other trees (B-tree, UB-tree, ...) do. On the other side, M-tree is more general than the *Spatial Access Methods* based on vector spaces.

4.2 Indexing XML Data with M-Tree

If we consider XML paths as simple objects, we can index such objects into a metric space or actually into the M-tree. For example, path `BOOK/AUTHOR/SURNAME` is object to store within M-tree. All paths in given XML document(s) can be transformed in this way into a collection of this simple XML objects. XML objects can also have assigned to every element tag its element content, which will increase the number of unique objects. For example, `BOOK{technical}/AUTHOR{writer}/ SURNAME{Walsh}`, but furthermore, for simplicity, we will ignore the possibility of any content.

XML object o_i (path) can be represented as a variable vector of strings (element tags), $o_i = (o_i^1, o_i^2, \cdots, o_i^{l_i})$.

Choosing metric for paths. Metric chosen for XML indexing must take as arguments two XML objects (paths) and calculate distance between them. We propose as an example *cumulated metric* which is defined as:

$$D(o_i, o_j) = \sum_{k=1}^{max(l_i,l_j)} d(o_i^k, o_j^k)$$

where $d(x, y)$ is an ordinary metric (e.g. Hamming metric) between two strings.

Hamming metric [8] adds up the mismatching pairs of characters where the first character of a pair is located on a position in the first string while the second character is on the same position in the second string. Formally,

$$d_H(x, y) = \sum_{i=1}^{min(|x|,|y|)} sgn(|x[i] - y[i]|) + ||x| - |y||$$

For example, $d_H($`AUTHORS`, `AUTOMATON`$) = 0 + 0 + 0 + 1 + 1 + 1 + 1 + 2 = 6$

Example 6.

Let d be the Hamming metric. Then

$D($`BOOK/AUTHOR/SURNAME`, `BOOK/AUTHOR/FIRSTNAME`$) = 0 + 0 + 8 = 8$

Let d be the discrete (yes/no) metric. Then

$D($`BOOK/PREFACE/TITLE`, `BOOK/BOOKINFO/TITLE`$) = 0 + 1 + 0 = 1$

Note: In this section, the paths used in examples are generated according to the DocBook DTD, see [12].

Processing queries. We have defined objects of metric space (XML paths) as well as metric (cumulated metric) thus we have accomplished the requirements for indexing with the M-tree.

We can distinguish two types of queries:

1. *similarity queries*. An object o_i in query result is within some distance r (*query radius*) to the query object o_q, i.e. the M-tree is traversed with condition $D(o_q, o_i) \leq r$. This kind of query allows to obtain the similar XML paths.

Example 7 (cumulated Hamming metric).
Query object = `BOOK/PART/CHAPTER/PARA/ACRONYM`, $r = 6$
Query result = {`BOOK/PART/CHAPTER/PARA/ACRONYM` (distance 0)
`BOOK/PART/CHAPTER/PARA/SCREEN` (distance 4)
`BOOK/PART/CHAPTER/TITLE/ACRONYM` (distance 5)
`BOOK/PART/CHAPTER/PARA/FILENAME` (distance 6) }

2. *exact matching queries.* An object o_i in query result must exactly match the query object o_q, i.e. the M-tree is traversed with the condition $D(o_q, o_i) = 0$. This is the special case of similarity query with $r = 0$ – no differences are allowed.

Notes:

- The query object is not expressed by any query language, its structure is the same as the structure of any ground object.
- The syntax of query object can be extended with keyword "*", where using this keyword on the k-th coordinate of object vector brings evaluation of $d(o_q^k, o_i^k)$ always as 0 (match).
  Example: D(`BOOK/AUTHOR/*`, `BOOK/AUTHOR/FIRSTNAME`) $= 0 + 0 + 0 = 0$.
  This extension allows to treat the exact matching queries as range queries.
- The objects in query result give only the information about existence of such paths in XML tree but the objects cannot tell the exact location. This lack of "context" can be removed with additional property of XML object – unique identifier of the last path element pointing into an external data structure (e.g. the source XML tree or UB-tree index). This improvement makes possible the consequential navigation in the external XML tree.

4.3 Testing with M-Tree

We have performed particular tests with M-tree – XML path indexing and XML similarity queries. XML data we have indexed was a XML file containing the documentation to DocBook. The size of this file was about 3MB.

In the first phase, we have transformed the whole file into collection of XML objects (unique paths) – 972 unique paths were extracted. Second, we have inserted all of these objects one-by-one into the M-tree. Page size of the M-tree was 1kB and cumulated Hamming metric was chosen. Each object (path vector) of the M-tree was aligned on size of 256 bytes.

Table 1. M-tree statistics

Level	Pages count	Avg. radius
0 (root)	3	207.33
1	5	183.00
2	10	135.40
3	16	121.75
4	23	102.74
5	34	85.18
6	53	65.79
7	83	54.02
8	141	37.14
9	233	22.95
10	376	13.12
11 (leafs)	590	6.01

After the indexing phase, the M-tree has acquired following statistics: pages(nodes) count: 1568, leafs count: 590. Table 1 shows for each level of the M-tree its pages count and average radius of all routing objects(regions) within the level.

Furthermore, disk access costs test was performed. A series of queries was produced by specifying the query object as:

("BOOK/PART/CHAPTER/SECT1/SECT2/PARA/ACRONYM") and by increasing the query radius from $r = 0$ (exact matching query) to $r = 32$. The results are shown in figure 5.

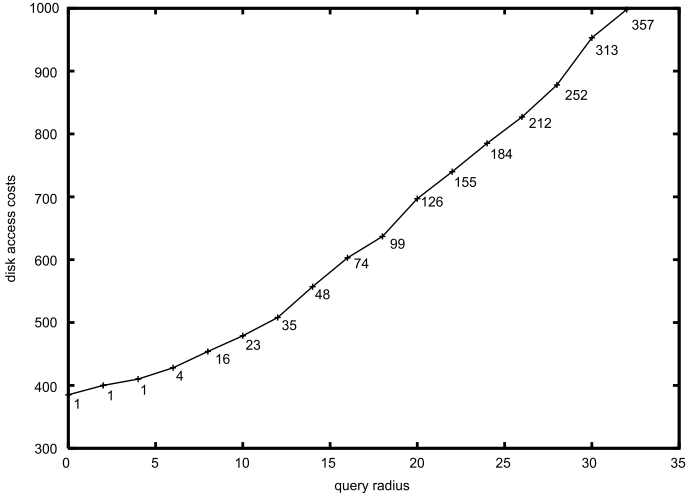

Fig. 5. Results of disk access costs test. The numbers below particular results are the total numbers of objects returned in particular query result (objects similar to the query object within the current radius).

5 Conclusions and Outlook

In this paper we have shown that XML data can be modelled in multidimensional vector spaces and in metric spaces. We use the UB-tree for indexing the vector spaces and the M-tree for indexing metric the metric spaces in our approach of indexing XML data. The data structures for indexing the vector spaces lead rather to the *exact queries* while structures for indexing of the metric spaces allow us to provide *similarity queries*. In the course of writing this paper some interesting questions appeared, e.g. new metric designs or different feature transformations. Their solution will be the topic of our future work. Furthermore, presented data structures are independent and our intention is to integrate them into a single hybrid data structure providing a possibility of XML data storage and also efficient exact and similarity querying.

Acknowledgments. This research was supported in part by GACR grant 201/00/1031.

References

1. Ciaccia P, Pattela M., Zezula P.: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. *Proc. 23rd Athens Intern. Conf. on VLDB (1997)*, 426–435.
2. Bayer R.: The Universal B-Tree for multidimensional indexing: General Concepts. In: *Proc. Of World-Wide Computing and its Applications 97 (WWCA 97)*. Tsukuba, Japan, 1997.
3. Böhm C., Berchtold S., Keim D.A.: Searching in High-dimensional Spaces – Index Structures for Improving the Performance of Multimedia Databases. *ACM*, 2002
4. Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B.: The R*-tree: An efficient and robust access method for points and rectangles. In: *Sigmod'90*, Atlantic City, NY, 1990, pp. 322–331.
5. Brechtold, S., Keim, A., Kriegel, H.-P.: The X-tree: An index structure for high-dimensional data. In: *Proc. of 22nd Intern. Conference on VLDB'96*, Bombay, India, 1996, pp. 28–39.
6. Bourret, R.: *XML and Databases*. `http://www.rpbourret.com/xml/ XMLAndDatabases.htm`. 2001.
7. Bozkaya, T., Özsoyoglu, M.: Distance-based indexing for high-dimensional metric spaces. In: *Sigmod '97*, Tuscon, AZ, 1997, pp. 357–368.
8. Baeza-Yates, R., Ribeiro-Neto, B.: *Modern Information Retrieval*. Addison Wesley, New York, 1999.
9. Krátký M., Pokorný, J., Snášel V.: Indexing XML data with UB-trees. *ADBIS 2002*, Bratislava, Slovakia, accepted.
10. Lee, D.L., Kim, Y.M., Patel, G.: Efficient Signature File Methods for Text Retrieval., *Knowledge and Data Engineering*, Vol. 7, No. 3, 1995, pp. 423–435.
11. Markl, V.: *Mistral: Processing Relational Queries using a Multidimensional Access Technique*, `http://mistral.in.tum.de/results/ publications/Mar99.pdf`, 1999
12. *The DocBook open standard*, Organization for the Advancement of Structured Information Standards (OASIS), 2002, `http://www.oasis-open.org/committees/docbook`
13. M. Patella: *Similarity Search in Multimedia Databases*. Dipartmento di Elettronica Informatica e Sistemistica, Bologna 1999 `http://www-db.deis.unibo.it/~patella/MMindex.html`
14. Pokorný, J.: XML: a challenge for databases?, Chap. 13 In: *Contemporary Trends in Systems Development* (Eds.: Maung K. Sein), Kluwer Academic Publishers, Boston, 2001, pp. 147–164.
15. *XQuery 1.0: An XML Query Language*. W3C Working Draft 20 December 2001, `http://www.w3.org/TR/2001/ WD-xquery-20011220/`