Implementation of XPath Axes in the Multi-dimensional Approach to Indexing XML Data[⋆]

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

[1] Department of Computer Science, VŠB – Technical University of Ostrava,
{michal.kratky,vaclav.snasel}@vsb.cz

[2] Department of Software Engineering, Charles University in Prague,
pokorny@ksi.ms.mff.cuni.cz

Czech Republic

Abstract. XML (Extensible Mark-up Language) has been recently understood as a new approach to data modelling. An implementation of a system enabling us to store and query XML documents efficiently requires the development of new techniques which make it possible to index an XML document in a way that provides an efficient evaluation of a user query. Most XML query languages are based on the language XPath and use a form of path expressions for composing more general queries. XPath defines a family of 13 axes, i.e. relationship types in which an actual element can be associated to other elements in the XML tree. Previously published multi-dimensional approaches to indexing XML data use paged and balanced multi-dimensional data structures like UB-trees and R*-trees. In this paper we revise the approaches and introduce a novel approach to the implementation of an XPath subset.

Keywords: indexing XML data, XPath axes, multi-dimensional data structures, UB-tree, R*-tree, BUB-forest, Signature R*-tree.

1 Introduction

XML [18] has been recently understood as a new approach to data modelling. A collection of *well-formed* XML documents is an XML database and the associated *DTD* or *XML Schema* is its database schema. An implementation of a system enabling us to store and query XML documents efficiently (so called *native XML databases*) requires the development of new techniques.

An XML document is usually modelled as a graph the nodes of which correspond to XML elements and attributes. The graph is mostly a tree (we suppose that none of the attributes is of `IDREF`/`IDREFS` type). To obtain specified data from an XML database, a number of special query languages have been developed, e.g. *XPath* [17] and *XQuery* [16]. A common feature of these languages is the possibility to formulate paths in the XML graph. In fact, most XML query languages are based on the XPath language that uses a form of path expressions for composing more general queries. The

[⋆] Work is partially supported by Grant of GACR No. 201/03/0912.

XPath defines a family of 13 *axes*, i.e. relationship types in which an actual element (*context node*) can be associated with other elements in the XML tree. The family of axes is designed to allow a set of graph traversal operations which are seen to be atomic in XML document trees.

In the past, there were many considerations about the straightforward use of existing relational or object-relational DBMSs for storing and querying XML data. Since a tree is accessed during the evaluation of a query, conventional approaches through the conventional database languages SQL or OQL fail or they are not enough efficient. Recently there have been several approaches to indexing XML data. Some of them are based on a traditional *relational technology* (e.g. *XISS* [12]), other use special data structures for the representation of XML data as a *trie* (e.g. *Index Fabric* [4] and *DataGuide* [14]) or multi-dimensional data structures (e.g. *XPath Accelerator* [6]). The latter approach uses R-trees but also B-trees as database indices in the environment of a relational DBMS. As expected, R-trees outperform B-trees in that proposal. The work [4] presents an index over the prefix-encoding of the paths in an XML document tree. A more complete summary of various approaches to indexing XML data is e.g. [3].

Previously published multi-dimensional approaches (e.g. [8, 9]) to indexing XML data use paged and balanced multi-dimensional data structures like *UB-trees* [1], *R-trees* [7], *R*-trees* [2], and *BUB-trees* [5]. In this paper we revise these approaches and combine them in such a way that an implementation of an XPath subset is possible. Our proposal is more general and enables an efficient accomplishment of querying text content of an element or attribute value as well as queries based on regular path expressions and axes of the XPath specification. We compared BUB-trees with R*-trees and their respective signature variants. The results confirm that our signature extension of multidimensional data structures is significantly better than the R*-tree. As these structures are paged and balanced, they are appropriate for indexing a huge amount of large XML documents and can serve as an alternative to the implementation of a native XML database.

In Section 2 we describe and revise a multi-dimensional approach to indexing XML data. This approach enables an efficient implementation of XPath axes (Subsection 2.3). We mainly focus on a class of queries employing exact matching for an element content or an attribute value. Section 3 describes multi-dimensional data structures *BUB-forest* [10] and *Signature R*-tree* [11], which are employed for indexing XML data. Section 4 provides some information about the complexity of query evaluation with multi-dimensional indexing. Section 5 reports on experimental results for selected XPath queries. In conclusion we summarize the results and outline future research directions.

2 Multi-dimensional Approach to Indexing XML Data

In [8, 9] a multi-dimensional approach to indexing XML data was introduced. We revise this approach in a way, which allows an efficient implementation of XML query languages based on the XPath.

2.1 Model of XML Documents

As far as an XML document modelled by a tree is concerned, we can view it as a set of paths from the root node to all leaf nodes. Suppose the unique number $id_N(u_i)$ of a

```
<!DOCTYPE books [
  <!ELEMENT books(book)>
  <!ELEMENT book(title,author)>
  <!ATTLIST book id CDATA #REQUIRED>
  <!ELEMENT title(#PCDATA)>
  <!ELEMENT author(#PCDATA)>
]>
```

```
<?xml version="1.0" ?>
<books><book id="003-04312">
    <title>The Two Towers</title>
    <author>J.R.R. Tolkien</author></book>
  <book id="001-00863">
    <title>The Return of the King</title>
    <author>J.R.R. Tolkien</author></book>
  <book id="045-00012">
    <title>Catch 22</title>
    <author>Joseph Heller</author></book>
</books>
```

Fig. 1. (a) DTD of documents which contain information about books and authors. (b) Well-formed XML document valid w.r.t. DTD

node u_i (element or attribute) which is obtained by counter increments according to the document order [6].

Let $\mathcal{P}$ be a set of all paths in a XML tree. A *path* $p \in \mathcal{P}$ in an XML tree is a sequence $id_N(u_0), id_N(u_1), \ldots, id_N(u_{\tau_P(p)-1}), s$, where $\tau_P(p)$ is the length of p, s is PCDATA or CDATA string, $id_N(u_i) \in D = \{0, 1, \ldots, 2^{\tau_D} - 1\}$, τ_D is the chosen length of binary representation of numbers from the domain D. Node u_0 is always the root node of the XML tree. Since each attribute is modelled as a super-leaf node with CDATA value, nodes $u_0, u_1, \ldots, u_{\tau_P(p)-2}$ always represent elements. A *labelled path* lp for a path p is a sequence $s_0, s_1, \ldots, s_{\tau_{LP}(lp)}$ of names of elements or attributes, where $\tau_{LP}(lp)$ is the length of lp, and s_i is the name of the element or attribute belonging to the node u_i. Let us denote the set of all labelled paths by $\mathcal{LP}$. A single labelled path belongs to a path, one or more paths belong to a single labelled path. If the element or attribute is empty, then $\tau_P(p) = \tau_{LP}(lp)$, else $\tau_P(p) = \tau_{LP}(lp) + 1$.

Signatures for coding of all strings, path and path content were used in the approach [8, 9]. Here, we use path, labelled path and term index.

Definition 1 (Point of n-dimensional Space Representing a Labelled Path).
Let $\Omega_{LP} = D^n$ be an n-dimensional space of labelled paths, $|D| = 2^{\tau_D}$, and $lp \in \mathcal{LP}$ be a labelled path $s_0, s_1, \ldots, s_{\tau_{LP}(lp)}$, where $n = max(\tau_{LP}(lp), lp \in \mathcal{LP}) + 1$. ***Point t_{lp} of n-dimensional space Ω_{LP} representing labelled path*** *is defined as $(id_T(s_0), id_T(s_1), \ldots, id_T(s_{\tau_{LP}(lp)}))$, where $id_T(s_i)$ is a unique number of term s_i, $id_T(s_i) \in D$. A unique number $id_{LP}(lp)$ is assigned to lp.* ■

Definition 2 (Point of n-dimensional Space Representing a Path).
Let $\Omega_P = D^n$ be an n-dimensional space of paths, $|D| = 2^{\tau_D}$, $p \in \mathcal{P}$ be a path $id_N(u_0), id_N(u_1), \ldots, id_N(u_{\tau_{LP}(lp)}), s$, and lp a relevant labelled path with the unique number $id_{LP}(lp)$, where $n = max(\tau_P(p), p \in \mathcal{P}) + 2$. ***Point t_p of n-dimensional space Ω_P representing path*** *is defined as $(id_{LP}(lp), id_N(u_0), \ldots, id_N(u_{\tau_{LP}(lp)}), id_T(s))$.* ■

Example 1 (Decomposition of the XML Tree to Paths and Labelled Paths).
In Figure 2 we see an XML tree modelling the XML document in Figure 1(b). This XML document contains paths:
– `0,1,2,'003-04312'`; `0,5, 6,'001-00863'`; and `0,9, 10,'045-00012'` belong to the labelled path `books,book,id`,

– `0,1,3,'The Two Towers'`; `0,5,7,'The Return of the King'`; and `0,9,11,'Catch 22'` belong to the labelled path `books, book,title`,

– `0,1,4,'J.R.R. Tolkien'`; `0,5,8, 'J.R.R. Tolkien'`; and `0,9,12,'Joseph Heller'` belong to the labelled path `books, book,author`.

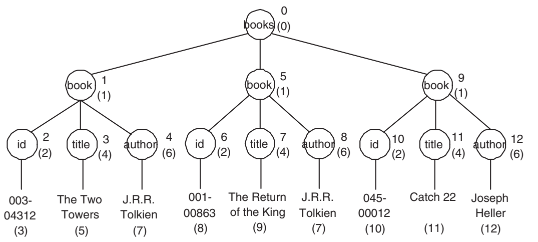

Fig. 2. Example of an XML tree with the unique numbers $id_N(u_i)$ of elements and attributes u_i and the unique numbers $id_T(s_i)$ of names of elements and attributes and their values s_i (values in parenthesis)

We define three indexes:

1. Term Index. This index contains a unique number $id_T(s_i)$ for each term s_i (names and text values of elements and attributes). Unique numbers can be generated by counter increments according to the document order. We want to get a unique number for a term and a term for a unique number as well. This index can be implemented by the B-tree.

In Figure 2 we see the XML tree with unique numbers of terms in parenthesis.

2. Labelled Path Index. Points representing labelled paths together with labelled paths' unique numbers (also generated by incrementing a counter) are stored in the labelled path index.

For three labelled paths in Figure 2 we create three points `(0,1,2)`; `(0,1,4)`; and `(0,1,6)` using id_T of element's and attribute's names. These points are inserted into a multi-dimensional data structure with id_{LP} equal to `0`, `1`, and `2`, respectively.

3. Path Index. Points representing paths are stored in the path index.

In Figure 2 we see unique numbers of elements. Let us take the path to the value `The Two Towers`. The relevant labelled path `book,book,title` has got id_{LP} 1 (see above). We get the point `(1,0,1,3,5)` after inserting a unique number of the labelled path id_{LP}, unique numbers of elements id_N and the term `The Two Towers`. This point is stored in a multi-dimensional data structure.

An XML document is transformed to points of vector spaces and XML queries are implemented using queries of multi-dimensional data structure. The multi-dimensional data structures provide a nature processing of *point* or *range queries* [1]. The point query probes if the vector is or is not present in the data structure. The range query searches all points in a query box $T_1 : T_2$ defined by two points T_1, T_2. Consequently, we can create the queries in the same way as the XML tree is decomposed to vectors of multi-dimensional spaces.

2.2 Queries for Values of Elements and Attributes

Now, we describe an implementation of a query for values of elements and attributes as well as a query defined by a simple path based on an ancestor-descendent relation. Query processing is performed in three phases which are connected:

1. **Find unique numbers id_T of the query's terms in the term index**.
2. **Find labelled paths' id_{LP} of the query in the labelled path index**. We search the unique numbers in a multi-dimensional data structure using point or range queries.
3. **Find points in the path index**. We find points representing paths in this index using range queries. Now, the result is formatted using unique numbers $id_N(u_i)$ of nodes u_i from these points.

Example 2 (Evaluation Plan of the XPath Query `/books/book[author="Joseph Heller"]`).

1. Find id_T of terms `books`, `book`, `author`, and `Joseph Heller` in the term index.
2. Find a unique number id_{LP} of the labelled path `books,book,author` in the labelled path index, which was transformed to the point representing the labelled path. We retrieve $id_{LP} = 2$ of labelled path by the point query `(0,1,6)`.
3. Create two points defining a query box, which searches points relevant to this query. The query box is defined by the points `(2,0,0,0,12)` and `(2,`max_D`,`max_D`,`max_D`,12)`, where max_D is the maximal value of the domain D of space Ω_P. id_{LP} of the labelled path retrieved during the last phase is located in the first points' coordinates. id_T of term `Joseph Heller` is located in the last points' coordinates. Since, we search points with arbitrary values of 2^{nd}–4^{th} coordinates, the first point contains the minimal values of a multi-dimensional space's domain and the second point contains the maximal values of the domain.

We need to distinguish between labelled paths and paths belonging to an element or attribute. We deal with this using flags added to points. Similarly, we can deal with the indexing of more XML documents, which can be valid w.r.t. different schemas.

2.3 Implementation of XPath Axes

The basic XPath query expression in non-reduced notation is `axis::tag[filter]`, which provides by evaluation on the context node u a set of nodes u', where the relation given by the `axis` contains (u, u'), tag for u' is `tag`, the condition assigned by `filter` assumes the value TRUE on u'.

Let us denote the context node by u, the level of a node u_i in the XML tree by $l(u_i)$. Obviously, $l(u_0) = 0$. Without loss of the generality we denote l as $l(u)$. The query result in the path index contains points representing paths from the root to leafs. These points contain the unique numbers of all ancestors $u_0, \ldots, u_{l-1}$ of the current node u, i.e. $id_N(u_0), \ldots, id_N(u_{l-1})$.

Implementation of Axes:

– `ancestor` – nodes lie on the path from u to the root node. So far as we want to retrieve the ancestors of a node from the point representing path, we retrieve relevant id_N from this point. Unique numbers $id_N(u_0), \ldots, id_N(u_{l-1})$ are obtained for the axis `ancestor`, $id_N(u_0), \ldots, id_N(u_{l-1}), id_N(u)$ for `ancestor-or-self` (u and nodes lie on the path from u to the root node), and $id_N(u_{l-1})$ for `parent` (first node on the path from u to the root node).

– `descendant` – all nodes, which the node u is the parent for. Now, we use unique numbers of u node's ancestors: $id_N(u_0), \ldots, id_N(u_{l-1})$, as well as $id_N(u)$ and $l(u)$. During the range query's points creation we use the knowledge that all descendants have got the same ancestors as the node u and their parent is the node u. We search all descendants of the node u by the range query in the path index: $(0, id_N(u_0), \dots, id_N(u_{l-1}), id_N(u), 0, \dots, 0) : (max_D, id_N(u_0), \dots, id_N(u_{l-1}), id_N(u), max_D, \dots, max_D)$.

– `child` – direct descendants of the node u. A naive approach is to perform axis `descendant` and id_N of children to obtain in $(l(u)$+3$)^{th}$ coordinates of the result's points. Let us imagine inefficient searching of the root node's children for example. A more efficient implementation is based upon finding one point, which contains id_N of a child, redefinition of the range query and processing of this query.

Algorithm:

1. Perform range query $(0, id_N(u_0), \dots, id_N(u_{l-1}), id_N(u), 0, \dots, 0) : (max_D, id_N(u_0), \dots, id_N(u_{l-1}), id_N(u), max_D, \dots, max_D)$. Processing of this query will be finished after the acquisition of one or no relevant point. The identified point contains id_N of a child in $(l(u)$+3$)^{th}$ coordinate.
2. Since we do not know which child was retrieved (from document order point of view) we must define two queries for the acquisition of both preceding and following children.

Denote the acquired child as u_c ($id_N(u_c) = id_N(u_{l+1})$), the child with the highest identified id_N, but smaller than $id_N(u_c)$ as u_{pc}, the child with the lowest identified id_N, but greater than $id_N(u_c)$ as u_{fc}. Definition of two range queries ($id_N(u_{pc}) = -1$ and $id_N(u_{fc}) = max_D + 1$ after finding the first child):

a) $(0, id_N(u_0), \dots, id_N(u_{l-1}), id_N(u), id_N(u_{pc}) + 1, 0, \dots, 0) : (max_D, id_N(u_0), \dots, id_N(u_{l-1}), id_N(u), id_N(u_c) - 1, max_D, \dots, max_D)$
b) $(0, id_N(u_0), \dots, id_N(u_{l-1}), id_N(u), id_N(u_c) + 1, 0, \dots, 0) : (max_D, id_N(u_0), \dots, id_N(u_{l-1}), id_N(u), id_N(u_{fc}) - 1, max_D, \dots, max_D)$

Processing the simple queries will be finished after the acquisition of one or no relevant point. In the case that the first/second query retrieves no point, the first/last child was identified in the former query.

3. We continue with step 2 until these range queries retrieve some points.

– `preceding-sibling` – siblings of node u preceding in the document order. We search all points representing paths pertaining to elements with the same ancestors as node u and $(l(u)$+2$)^{th}$ coordinate $< id_N(u)$. A naive approach to finding the preceding-siblings of node u is to perform the range query: $(0, id_N(u_0), \dots, id_N(u_{l-1}), 0, 0, \dots, 0) : (max_D, id_N(u_0), \dots, id_N(u_{l-1}), id_N(u) - 1, max_D, \dots, max_D)$. An efficient implementation is similar to the implementation of the `child` axis.

– `following-sibling` – siblings of node u following in the document order. We search all points representing paths pertaining to elements with the same ancestors as node u and $(l(u)$+2$)^{th}$ coordinate $> id_N(u)$. A naive approach to finding the following-siblings of node u is to perform the range query: $(0, id_N(u_0), \dots, id_N(u_{l-1}), id_N(u) + 1, 0, \dots, 0) : (max_D, id_N(u_0), \dots, id_N(u_{l-1}), max_D, \dots, max_D)$. An efficient implementation is similar to the implementation of the `child` axis.

– `preceding` – nodes preceding to node u (except ancestors) in the document order. We search the nodes preceding the node u by the range query: $(0, 0, \dots, 0) : (max_D, id_N(u_0) - 1, \dots, id_N(u_{l-1}) - 1, id_N(u) - 1, max_D, \dots, max_D)$.

– `following` – nodes following to node u (except descendants) in the document order. We search the nodes following the node u by the range query: $(0, id_N(u_0) + 1, \ldots, id_N(u_{l-1}) + 1, id_N(u) + 1, 0, \ldots, 0) : (max_D, \ldots, max_D)$.

In the above described range queries no restrictions were given to the unique numbers of labelled paths. In practice, the first coordinates of range query's points contain a particular id_{LP} instead of an interval $\langle 0; max_D \rangle$.

3 Index Data Structures

Due to the fact that an XML document is represented as a set of points in the multi-dimensional approach, we use multi-dimensional data structures like UB-tree, BUB-tree, R-tree, and R*-tree for their indexing. (B)UB-tree data structure uses *Z-addresses* (*Z-ordering*) [1] for mapping a multi-dimensional space into a single-dimensional one. Intervals on *Z-curve* (which is defined by this ordering) are called *Z-regions*. (B)UB-tree stores points of each Z-region on one disk page (tree's leaf) and a hierarchy of Z-regions forms an index (inner nodes of tree). In the case of indexing point data, an R-tree and its variants cluster points into *minimal bounding boxes* (*MBB*s). Leafs contain indexed points, super-leaf nodes include definition of MBBs and remaining inner nodes contain a hierarchy of MBBs. (B)UB-tree and R-tree support *point* and *range queries* [1], which are used in the multi-dimensional approach to indexing XML data. The range query is processed by iterating through the tree and filtering of irrelevant tree's parts (i.e., regions) which do not intersect a query box.

One more important problem of the multi-dimensional approach is the unclear dimension of spaces of paths and labelled paths. Documents with the maximal length of path being 10 exist, but documents with the maximal path length 36 may appear as well (see [13]). A naive approach is to align the dimension of space to the maximal length of path. For example, points of dimension 5 will be aligned to dimension 36 using a blank value (often zero number) in 6^{th}–36^{th} coordinates. This technique increases the size of the index and the overhead of data structure as well. In [10] *BUB-forest* data structure was published. This data structure deals with the problem of indexing points of different dimensions. BUB-forest contains several BUB-trees, each of them indexes a space of different dimension. We can index points representing paths and labelled paths regardless of worsening efficiency by indexing XML documents with very different length of paths. We can use the same approach for other data structures, e.g. R-trees.

The range query used in the multi-dimensional approach is called the *narrow range query*. Points defining a query box have got coordinates for some dimensions the same, whereas the size of interval defined by the coordinates of the first and second point for other dimensions nears to the size of space's domain. Notice, regions intersecting a query box during processing of a range query are called *inter-sect regions* and regions containing at least one point of the query box are called *relevant regions*. We denote their number by N_I and N_R, respectively. Many irrelevant regions are searched during processing of the narrow range query in multi-dimensional data structures. Consequently, a ratio of relevant and intersect regions, so called *relevance ratio* c_R, becomes much lower than 1 with an increasing dimension of indexed space. In [11] the *Signature R-tree* data structure was depicted. This data structure enables the efficient processing of the narrow range query. Items of inner nodes contain a definition of (super)region and n-dimensional signature of tuples included in the (super)region (see Figure 2). A superposition of tuples' coordinates by operation `OR` creates the signature. Operation `AND` is used for better filtering of irrelevant regions during processing of the narrow range query. Other multi-dimensional data structures, e.g. (B)UB-tree, are possible to extend in the same way.

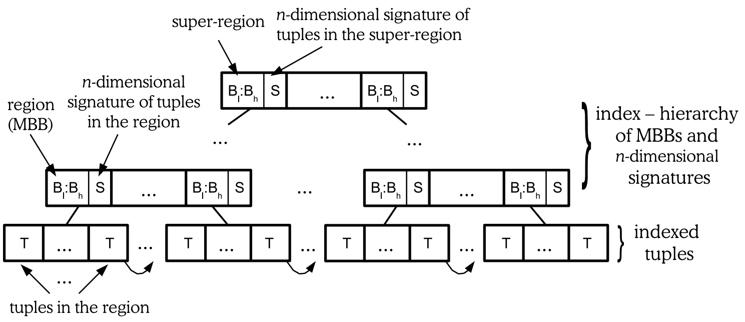
super-region
n-dimensional signature of tuples in the super-region
n-dimensional signature of tuples in the region
region (MBB)
B_l:B_h
S
T
...
index – hierarchy of MBBs and n-dimensional signatures
indexed tuples
tuples in the region

Fig. 3. A structure of the Signature R-Tree

4 Cost Analysis

A point query is often used for searching in the term and the labelled path index. The complexity of the point query is $O(log(m))$, where m is the number of the indexed objects. The efficiency of the multi-dimensional approach mainly depends on the efficiency of range query processing in the path index. Complexity of the general range query algorithm is $O(N_I \times \log_c m)$, where c is the node's capacity. It holds $c_R \ll 1$ (see previous section) for the narrow range query, particularly for increasing dimension of indexed space. In the case of the Signature R-tree (and (B)UB-tree as well) is the complexity $O(N_{RQ} \times \log_c m)$, where N_{RQ} is the number of searched regions (leaf nodes). Our experiments show $N_I \gg N_{RQ} \geq N_R$. In other words, the space complexity of the algorithm is enhanced for the reduction of the time complexity. Some XML queries are implemented by a sequence of q range queries. The complexity of the sequence is $O(\sum_{i=1}^{q} N_{RQ} \times log_c(m))$.

Conventional approaches, like XISS [12] and XPath Accelerator [6], index particular elements (and attributes). Simple query for values of elements as well as a query defined by a simple path based on an ancestor-descendent relation is processed using a consecutive filtering of elements which are not in relation ancestor-descendent as long as the result is retrieved. In Section 2.2 we can see that, in our approach, such a query is processed using one query in a data structure and the filtering of a large number of irrelevant elements does not approach.

5 Experimental Results

In our experiments[1], we used Protein Sequence Database XML document [15]. The document size is 683 MB. It includes 21,305,818 elements and 1,290,647 attributes. Approximately 17 mil. paths were obtained from this document. With respect to the frequency of the path lengths, the multi-dimensional forests with two trees indexing spaces of dimension $n = 7$ and $n = 9$ were created for indexing XML data. We used BUB-tree, R*-tree, and their signature variants with the length of n-dimensional signature $n \times 64$. The underlying table summarizes the index characteristics, square brackets indicate an increase of index volume for signature multi-dimensional trees. The average utilisation 62% was reached.

Dimen-sion n	**Number of points**	**Index size** [MB] **BUB-tree**	**Sig. BUB-tree**	**R*-tree**	**Sig. R*-tree**	**Number of inner nodes (BUB-tree)**	**Number of leaf nodes (BUB-tree)**
7	8,268,357	440.9	471 [+7%]	478.6	512.2 [+7%]	10,917	214,842
9	8,739,522	562.1	635.2 [+13%]	603.1	680.7 [+13%]	17,751	270,065

First, queries for values of elements and attributes as well as a query defined by a simple path based on an ancestor-descendent relation similar to `ProteinDatabase/ProteinEntry[reference/refinfo/authors/author ='Smith, E.L.']` were tested. For each space, queries with smaller (bellow 10) and larger (10^3–10^4) size of results were selected. In all cases, the ratio of number of searched leaf nodes and number of all leaf nodes, disk access cost (DAC), and query processing time were measured. The results of query processing are presented in Table 1. Evidently, the R*-tree proves to have better properties than BUB-trees during the narrow range processing. Signature variants of these data structures provide better efficiency than classical data structures. In the case of Signature R*-tree only 0.14% of all leaf nodes were searched and the average time of query processing turns out to be 70 ms.

Table 1. Results of queries for values of elements and attributes as well as a query defined by a simple path based on an ancestor-descendent relation in the path index

	Searched leaf nodes [%]				**DAC**				**Time** [s]			
	BUB tree	Sign. BUB-tree	R* tree	Sign. R*-tree	BUB tree	Sign. BUB-tree	R* tree	Sign. R*-tree	BUB tree	Sign. BUB-tree	R* tree	Sign. R*-tree
Avg.	1.15	0.22	0.29	**0.14**	7,566	794	917	**445**	2.9	0.5	0.12	**0.07**

Second, the efficiency of XPath axes implementation was tested. In all cases, DAC and query processing time for the Signature R*-tree were measured. The results of query processing are presented in Table 2. Ancestors axes are processed by no disk access.

[1] The experiments were executed on an Intel Pentium®4 2.4Ghz, 512MB DDR333, under Windows XP.

Table 2. Results of XPath axes queries in the path index

Axis	Number of resultant		Searched leaf	DAC	DAC for simple	Time
	elements	points	nodes [%]		range query	[s]
`descendant`	1,121	982	0.20	621	-	0.06
`child`	9	9	0.05	225	25	0.1
`descendant-or-self`	1,245	1,015	0.23	648	-	0.05
`parent`	1	-	-	-	-	-
`following`	2,487	2,017	0.45	1,387	-	0.1
`preceding`	2,312	1,803	0.39	1,124	-	0.09
`following-sibling`	5	5	0.04	187	27	0.09
`preceding-sibling`	7	7	0.03	165	24	0.09
Avg.	**1,026.6**	**585**	**0.20**	**622.4**	**25.3**	**0.08**

Since some axes (`child`, `following-sibling`, and `preceding-sibling`) are implemented by a sequence of range queries, DAC for one range query are presented in Table 2 as well. The volume of searched leaf nodes is again very low.

6 Conclusion

In this paper the multi-dimensional approach to indexing XML data and an efficient implementation of XPath axes were described. The BUB-forest was employed for indexing heterogeneous XML data with the root to leaf paths of widely differing lengths. Our experiments prove that the approach can serve as an alternative to implementing native XML databases. In our future work, we would like to further improve the abilities and the efficiency of the multi-dimensional approach. In particular we will develop an implementation of another complex XML queries which are defined by XML query languages such as XPath and XQuery.

References

1. R. Bayer. The Universal B-Tree for multidimensional indexing: General Concepts. In *Proceedings of WWCA'97, Tsukuba, Japan*, 1997.
2. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331.
3. A. B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems*. Addison Wesley Professional, 2003.
4. B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *Proceedings of the 27th VLDB Conference*, 2001.
5. R. Fenk. The BUB-Tree. In *Proceedings of 28th VLDB Conference*, 2002.
6. T. Grust. Accelerating XPath Location Steps. In *Proceedings of ACM SIGMOD 2002, Madison, USA*, June 4-6, 2002.

7. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOD 1984, Boston, USA*, pages 47–57, June 1984.
8. M. Krátký, J. Pokorný, T. Skopal, and V. Snášel. The Geometric Framework for Exact and Similarity Querying XML Data. In *Proceedings of First EurAsian Conferences, EurAsia-ICT 2002, Shiraz, Iran*. Springer–Verlag, LNCS 2510, 2002.
9. M. Krátký, J. Pokorný, and V. Snášel. Indexing XML data with UB-trees. In *Proceedings of ADBIS 2002*, volume Research Commmunications, pages 155–164.
10. M. Krátký, T. Skopal, and V. Snášel. Multidimensional Term Indexing for Efficient Processing of Complex Queries. *Kybernetika, Journal of the ACR, accepted*, 2004.
11. M. Krátký, V. Snášel, J. Pokorný, and P. Zezula. Efficient Processing of Narrow Range Queries in the R-Tree. Technical Report ARG-TR-01-2004, *http://www.cs.vsb.cz/arg*, 2004.
12. Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of 27th VLDB International Conference*, 2001.
13. L. Mignet, D. Barbosa, and P. Veltri. The XML Web: a First Study. In *Proceedings of Twelfth International World Wide Web Conference, WWW 2003*. ACM, 2003.
14. J. W. R. Goldman. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of 23rd VLDB Conference*, 1997.
15. University of Washington's database group. The XML Data Repository, 2002, `http://www.cs.washington.edu/research/xmldatasets/`.
16. W3 Consortium. XQuery 1.0: An XML Query Language, W3C Working Draft, 15 November 2002, `http://www.w3.org/TR/xpath/`.
17. W3 Consortium. XML Path Language (XPath) Version 2.0, W3C Working Draft, 15 November 2002, `http://www.w3.org/TR/xpath20/`.
18. W3 Consortium. Extensible Markup Language (XML) 1.0, 1998, `http://www.w3.org/TR/REC-xml`.