A Comparison of Element-based and Path-based Approaches to Indexing XML Data*

Michal Krátký, Radim Bača

Department of Computer Science, VŠB – Technical University of Ostrava
17. listopadu 15, 708 33 Ostrava–Poruba
{michal.kratky,radim.baca}@vsb.cz
Czech Republic

Abstract. The mark-up language XML (Extensible Mark-up Language) is recently understood as a new approach to data modeling. A well-formed XML document or a set of documents is an XML database and the associated DTD or schema specified in the language XML Schema is its database schema. Implementation of a system enabling us to store and query XML documents efficiently (so called native XML databases) requires a development of new techniques that make it possible to index an XML document in a way that provides an efficient evaluation of a user query. Most of XML query languages are based on the language XPath and use a form of path expressions for composing more general queries. In the paper we compare element-based and path-based approaches to indexing XML data. In the case of element-based approaches query is evaluated step by step. Each step produces a lot of elements which may be refused in the next evaluation step. In the paper we show that the previously published multi-dimensional path-based approach overcomes conventional element-based approaches.

Key words: indexing XML data, XPath, element-based approach, path-based approach, multi-dimensional data structures

1 Introduction

The mark-up language *XML* (*Extensible Mark-up Language*) [20] is recently understood as a new approach to data modelling [16]. A *well-formed* XML document or a set of documents is an XML database and the associated DTD or schema specified in the language *XML Schema* [23] is its database schema. Implementation of a system enabling us to store and query XML documents efficiently (so called *native XML databases*) requires a development of new techniques [16, 5].

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 consider no attribute `IDREFS` now). To obtain specified data from an XML database a

* Work is partially supported by Grant of GACR No. 201/06/P113.

number of special query languages have been developed, e.g. *XPath* [22], and *XQuery* [21]. A common feature of these languages is a possibility to formulate paths in the XML graph. Such a path is a sequence of element or attribute names from the root element to a leaf. Regular expressions provide a valuable method for paths specifications. In fact, most of XML query languages are based on the XPath language that uses a form of path expressions for composing more general queries. The XPath defines a family of 13 *axes*, i.e. relationship types in that an actual element can be associated to other elements represented in the XML tree. The family of axes defined in the XPath is designed to allow the set of graph traversal operations that are seen to be atomic in XML document trees.

In the past, there were many considerations about use of existing relational or object-relational DBMSs for storing and querying XML data. Since a tree is accessed during evaluation of a query, conventional approaches through the conventional database languages SQL or OQL fail or they are not too efficient. Consequently, a form of indexing is necessary.

Recently there are several approaches to indexing XML or, more general, semistructured data. Some of them are based on a traditional *relational technology* (e.g. *Lore* [15] and *XISS* [14]), the others use special data structures for representation of XML data like *trie* (e.g. *Index Fabric* [7] and *DataGuide* [17]) or multi-dimensional data structures (e.g. *XPath Accelerator* [9]). The latter approach uses R-trees but also B-trees as database indices in environment of a relational DBMS. As it was expected, R-trees outperforms B-trees in this proposal. The work [7] presents an index over the prefix-encoding of the paths in an XML document tree, in which each leaf u_L of the document tree is prefixed by the sequence of element tags that one encounters during a path traversal from the document root to u_L. A more complete summary of various approaching to indexing XML data is e.g. [6].

In the course of the development of XML databases the need for a benchmark framework has become more and more evident: many different ways to store and query XML data have been suggested in the past, e.g. *XMark* [18] and *XML Data Repository* [19].

From the other point of view we distinguish element-based (e.g. *XPath Accelerator* [9] or *XISS* [14]) and path-based (e.g. [13, 11]) approaches to indexing XML data. In the case of an element-based approach a query is evaluated step by step. Each step produces a lot of elements which may be refused in the next evaluation step.

In the paper we compare XPath accelerator (XPA) element-based approach and the multi-dimensional (MDA) path-based approach. We show that the previously published multi-dimensional path-based approach (e.g. [13, 11]) overcomes conventional element-based approaches. In Section 2 we describe XPA as a typical representative of element-based approaches. In Section 3 we describe MDA to indexing XML data. Section 4 reports on results of experiments for selected XPath queries. In conclusion we summarize the paper content and outline possibilities of a future work.

2 XPath Accelerator (XPA)

XPA [9] is an indexing method for efficient evaluation of XPath queries. XPA is an element-based approach to indexing semi-structured data applying multi-dimensional data structures like R-tree or UB-tree but it can be realized in a relational database as well.

2.1 Model of XML documents

Every XML document can be modelled as a tree where one tree node corresponds to exactly one element or attribute of XML document. For support all XPath axes XPA assigns 5-dimensional descriptor $desc(v)$ to every node v

$$desc(v) = \langle pre(v), post(v), par(v), att(v), tag(v)\rangle \, .$$

The first attribute in the descriptor is *preorder rank* $pre(v)$ of node v. $Pre(v)$ corresponds to a *document order* of node v. *Document order* is defined as an order in which nodes appear in XML serialization of a document. Otherwise said, *document order* corresponds to the order in which nodes come in sequential reading of text representation of XML document. $Pre(v)$ is assigned to node v before any of its children is visited in sequential reading. The second attribute is *postorder rank* $post(v)$ of node v. This number is assigned to node v after all its children are visited. Let v and v' be evaluated nodes of XML tree. Then

- v' is descendant of $v \quad \Leftrightarrow \quad pre(v) < pre(v') \;\wedge\; post(v') < post(v)$,
- v' is following of $v \quad \Leftrightarrow \quad pre(v) > pre(v') \;\wedge\; post(v') < post(v)$.

Similarly, relations *ancestor* and *preceding* can be evaluated between any two nodes. If every node v of the tree has assigned pair $(pre(v), post(v))$ then we are able to determine four major axes *descendant*, *ancestor*, $following$ and *preceding* for the node v with a single query. We call this node a context node. The determination of axe for a context node means finding all nodes which are in the appropriate relation with the context node.

Example 1 (Evaluation of four major axis for context node).

In Figure 1(a) we can see an XML document modelled as a tree. All nodes are evaluated with the preorder and postorder rank. In Figure 1(b) we see *pre/post* plane divided into four regions where every region corresponds to one axis. The division is made for a context node h. Major axes for the context node h contain the following nodes:

- a, f in axis $ancestor :: *$,
- k in axis $following :: *$,
- i, j in axis $descendant :: *$,
- b, c, d, e, g in axis $preceding :: *$.

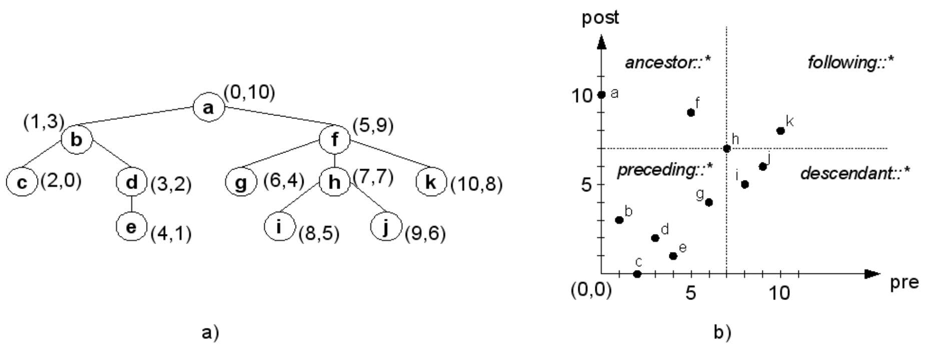

Fig. 1. (a) Evaluation of an XML tree with pairs $(pre(v), post(v))$. (b) Node distribution in *pre/post* plane and four major axes for a context node h.

Descriptor $desc(v)$ for a node v includes an attribute $par(v)$ for support of axes *parent*, *child*, $following - sibling$ and $preceding - sibling$. The attribute stores the parent's preorder rank of a node v. Boolean attribute $att(v)$ is true if the node v is *attribute*. The attribute *att* is included to support of *attribute* axis. Finally, attribute *tag* stores element (or attribute) name *id*. There is an algorithm which can compute the descriptor *desc* for every node of an XML tree during single sequential reading of an XML document.

We use a term index for mapping term to its *id*. Every term which occurs during parsing XML document is faced with term index and mapped to appropriate *id*. Attribute and element names are stored in XPA index as a part of descriptor *dest*, but also terms in an element content or attribute value should be stored somehow. We decided to use the inverted list [2]. Value of every element or attribute v is parsed into single words which are mapped to *id*s. We store pairs $(id, pre(v))$ for every word in the inverted list. Consequently, we can retrieve $pre(v)$ of all nodes to be contain searched word.

2.2 Querying in the XPA index

XPA index is done after we map the whole XML document into 5-dimensional space. We resolve *location steps* of XPath query step by step. The *location step* consists of name of the axis, name of the node ($nodeName$) and predicate. The predicate is optional but in the case there is some predicate we have to solve *axis::nodeName* part and predicate separately and then we have to union the results. We designed implementation of XPA so that it is possible to handle nested predicates. Solving *axis::nodeName* part of one *location step* is realized using query upon 5-dimensional space. We find all nodes inside 5-dimensional cube as it is shown in Table 1 in more detail.

Wildcard '*' in Table 1 means that this attribute can have any value to match corresponding axis, but value of attribute *tag* depends on value of $nodeName$ of

Table 1. Intervals of each attribute for evaluation of corresponding XPath axe.

axe	**pre**	**post**	**par**	**att**	**tag**
child	$(pre(v), \infty)$	$[0, post(v))$	$pre(v)$	false	*
descendant	$(pre(v), \infty)$	$[0, post(v))$	*	false	*
descendant-or-self	$[pre(v), \infty)$	$[0, post(v))$	*	false	*
parent	$[par(v), par(v)]$	$[post(v), \infty)$	*	false	*
ancestor	$[0, pre(v))$	$(post(v), \infty)$	*	false	*
ancestor-or-self	$[0, pre(v)]$	$[post(v), \infty)$	*	false	*
following	$(pre(v), \infty)$	$(post(v), \infty)$	*	false	*
preceding	$[0, pre(v))$	$[0, post(v))$	*	false	*
following-sibling	$(pre(v), \infty)$	$(post(v), \infty)$	$par(v)$	false	*
preceding-sibling	$[0, pre(v))$	$[0, post(v))$	$par(v)$	false	*
attribute	$(pre(v), \infty)$	$[0, post(v))$	$pre(v)$	true	*

location step. When the index applies R-trees or other multi-dimensional data structure retrieving of all nodes inside 5-dimensional cube can be performed by a single range query.

XPath query is evaluated from one context node v_c. XPath query consists of a sequence of *location steps*. Query processing is done in these phases:

1. We obtain a set of nodes S_1 as a result of evaluation of the first *location step* from context node v_c. We set $i = 1$.
2. The set S_i is established as a set of context nodes for the following step.
3. We evaluate $(i + 1)$th *location step* for every context node from the set S_i and the result is a set of nodes S_{i+1}. We increment i by one.
4. Phases 2 and 3 are repeated until the last *location step* of XPath query is evaluated.
5. Set of nodes S_i is the result of the XPath query.

That means running many range queries during every phase 3. With increasing number of *location steps* the execution time of the query increases as well. Size of the set S_i which is created during each *location step* may be much larger then the size of the XPath query result. Such inefficiency leads to unnecessary execution time overhead.

3 Multi-dimensional Approach to Indexing XML Data

In [13, 11, 12] MDA was introduced. This path-based approach applies to indexing XML data paged and balanced multi-dimensional data structures like *UB-trees* [3], *R-trees* [10], *R^*-trees* [4], and *BUB-trees* [8].

3.1 Model of XML documents

As mentioned above an XML document may be modelled by a tree, whose nodes correspond to elements and attributes. String values of elements or attributes or empty values occur in leafs. An attribute is modelled as a child of the related element. Consequently, an XML document may be modelled as a set of paths from the root node to all leaf nodes. Note, unique number $id_U(u_i)$ of a node u_i (element or attribute) is obtained by counter increments according to the *document order* [9]. Unique numbers may be obtained using an arbitrary numbering schema. Of course, document order must be preserved.

Let $\mathcal{P}$ be a set of all paths in a XML tree. The path $p \in \mathcal{P}$ in an XML tree is sequence $id_U(u_0), id_U(u_1), \ldots, id_U(u_{\tau_P(p)-1}), s$, where $\tau_P(p)$ is the length of the path p, s is `PCDATA` or `CDATA` string, $id_U(u_i) \in D = \{0, 1, \ldots, 2^{\tau_D} - 1\}$, τ_D is the chosen length of binary representation of a number from 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 $u_0, u_1, \ldots, u_{\tau_P(p)-2}$ represent elements always.

```
<!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. 2. (a) DTD of documents which contain information about books and authors. (b) Well-formed XML document valid w.r.t DTD.

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 the labelled path 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 $\tau_P(p) = \tau_{LP}(lp) + 1$.

Example 2 (Decomposition of XML tree to paths and labelled paths).
In Figure 2 we see an example of an XML document. In Figure 3 we see an XML tree modelling the XML document. We see that this XML document contains paths:

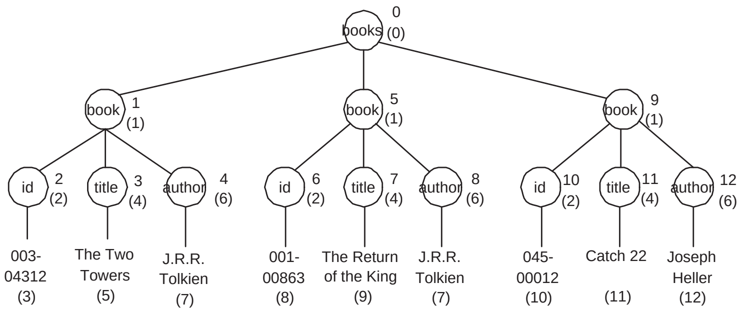

Fig. 3. Example of XML tree with unique numbers $id_U(u_i)$ of elements and attributes u_i and unique numbers $id_T(s_i)$ of names of elements and attributes and their values s_i (values in parenthesis).

– `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`.

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 of n-dimensional space representing a labelled path*** *is defined $t_{lp} = (id_T(s_0), id_T(s_1), \ldots, id_T(s_{\tau_{LP}(lp)})) \in \Omega_{LP}$, where $id_T(s_i)$ is a unique number of term s_i, $id_T(s_i) \in D$. A unique number $id_{LP}(lp_i)$ is assigned to lp_i.* ■

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 $id_U(u_0), id_U(u_1), \ldots, id_U(u_{\tau_{LP}(lp)}), s$ 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 of n-dimensional space representing path*** *is defined $t_p = (id_{LP}(lp), id_U(u_0), \ldots, id_U(u_{\tau_{LP}(lp)}), id_T(s)) \in \Omega_P$.* ■

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). The 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 too. This index can be implemented by the B-tree.

In Figure 3 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 counter increments) are stored in the labelled path index.

   In Figure 3 we see that the document contains three unique labelled paths `books,book,id`; `books,book,title`; and `books,book,author`. We create 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} `0`, `1`, and `2`.

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

   In Figure 3 we see unique numbers of elements. Let us take the path to the value `The Two Towers`. Relevant labelled path `book,book,title` has got id_{LP} 1 (see labelled path index). We get point `(1,0,1,3,5)` after inserting unique numbers of labelled path id_{LP}, unique numbers of elements id_U and 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 a multi-dimensional data structure queries. The multi-dimensional data structures provide a nature processing of *point* or *range queries* [3]. 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.

3.2 Queries for values of elements and attributes

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

1. **Finding unique numbers id_T of query's term in the term index**.
2. **Finding labelled paths' id_{LP} of query in the labelled path index**. We search the unique numbers in a multi-dimensional data structure using point or range queries.
3. **Finding points in the path index**. We find points representing paths in this index using range queries. Now, we often want to retrieve (using labelled paths and term index) names or values of elements and attributes.

In Figure 4(a) we see that we can model the query (a) as a tree. Consequently, we can create the range queries in the same way as the XML tree is decomposed to vectors of multi-dimensional spaces.

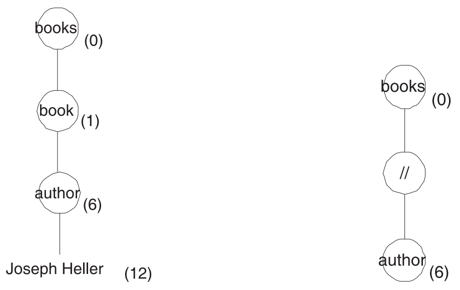

Fig. 4. Trees modelling XPath queries for values of elements or attributes: (a) `/books/books[author='Joseph Heller']` (b) `/books//author`.

Example 3 (Evaluation plan of the XPath query /books/book[author="Joseph Heller"]).

By the query we want to retrieve all books written by `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 points `(2,0,0,0,12)` and $(2, max_D, max_D, max_D, 12)$, where max_D is the maximal value of 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 multi-dimensional space's domain and the second point contains the maximal values of the domain.

We need to distinguish labelled paths and paths belonging to element or attribute. We solve it using flags added to points. Similarly, we can solve indexing of more XML documents, which can be valid w.r.t different schema. Hence, the multi-dimensional approach is hopeful for implementation of a native XML database.

4 Experimental results

In our experiments we compare XPA element-based approach with MDA path-based approach. We show the element-based XPA is less effective than MDA.

Both approaches are based on multi-dimensional data structures (R-tree [10] and Signature R-tree [12], respectively). The framework *ATOM* [1] is applied in our implementation of data structures. Although compared approaches are very different we can compare some same parameters (e.g. DAC). Consequently, we show the main disadvantage of element-based approaches. Single steps are evaluated step by step during a query evaluation. Each step produces a lot of elements which may be refused in the next evaluation step. We show the number of the refused elements is rather large in the case of XPA.

In our experiments[1] we use the XMARK collection [18]. The collection contains one file of the size 111MB. It includes 2,082,854 elements. Table 2 shows statistics of XPA indices. In Table 3 tested queries are put forward. These queries were selected wilfully. The first one includes 180 elements in the result, whereas the second one includes 7.5× elements more than the first query, that is 1,350.

Table 2. Statistics of XPA indices (element index, inverted list, and term index)

	XPA index	**Inverted list**	**Term indexes**
Tree level	4	3	1 - 4
Number of items	2,081,550	8,130,422	376,906
Number of inner nodes	1,378	1,574	1,038
Number of list nodes	31,980	71,638	6,559
Average filling [%]	74.8	65.9	61
Size of inner node [B]	2,028	2,044	
Size of leaf node [B]	2,048	2,048	
Item size of inner node [B]	40	16	
Item size of leaf node [B]	24	12	
Dimension	5	2	

Table 3. Two XPath queries evaluated in our experiments

Query	**XPath query**	Result Size
Q1	`/site/closed_auctions/closed_auction/annotation/description/parlist/listitem/parlist/listitem/text/emph/keyword/`	180
Q2	`/site/regions/africa/item[location='United']`	1,350

Tables 5 and 6 show results of evaluation of queries Q1 and Q2, respectively, in XPA. Tables includes surveyed parameters for each *location step*. In Table 4 such parameters are described.

Inefficiency of element-based approach is obvious in the difference between **Nodes** and **Useful** values. In the case of Q1 55,383 elements are retrieved but

[1] The experiments were executed on an Intel Pentium ®4 2.4Ghz, 1GB DDR400, under Windows XP.

Table 4. Surveyed parameters during query evaluation in XPA

Nodes	Number of nodes in the result set after evaluation of one *location step*
Useful	Number of nodes which leads to at least one node in the next *location step*
Time	Time for processing the step
DAC	Number of access in indices

the result contains only 180 elements. In the case of Q2 27,493 elements are retrieved but the result contains only 1,350 elements.

Table 5. Statistics of query Q1 evaluated with XPA

Step	**Nodes**	**Useful**	**Time [s]**	**DAC**
site	1	1	0.02	5
closed_auctions	1	1	0	5
closed_auction	9,750	9,750	1.9	1,386
annotation	9,750	9,750	4.6	50,594
description	9,750	2,934	5	50,252
parlist	2,934	2,934	4.64	49,773
listitem	8,512	1,713	1.9	15,448
parlist	1,713	1,713	4.02	43,114
listitem	4,964	4,964	1.02	8,872
text	4,964	1,890	2.11	24 999
emph	2,864	173	1.97	24,806
keyword	180	**180**	0.95	14,070
Sum	**55,383**	36,003	28.27	283,324

Table 6. Statistics of query Q2 evaluated with XPA

Step	**Nodes**	**Useful**	**Time**	**DAC**
site	1	1	0	5
regions	1	1	0	5
africa	1	1	0	5
item	550	550	0.08	27
location ∼= 'United'	26,940	**1,350**	3.34	5,673
Sum	**27,493**	1,903	3.48	5,716

Table 8 shows results of queries Q1 and Q2 in the case of MDA. Surveyed parameters are put forward in Table 7. We can see the time and DAC of query evaluation is lower than in the case of XPA. The advantage of MDA is obvious.

Table 7. Surveyed parameters during query evaluation in MDA

DAC_t	Number of access in term index
DAC_p	Number of access in labeled path index
DAC_{lp}	Number of access in path index
DAC	Whole number of access in indices
Time	Time of query evaluation

Table 8. Statistics of queries Q1 and Q2 evaluated with MDA

	Q1	**Q2**
DAC_t	211	60
DAC_{lp}	1	144
DAC_p	723	2,400
DAC	934	2,604
Time [s]	0.49	0.26

5 Conclusion

In the paper we compare XPA element-based approach and MDA path-based approach. In the case of an element-based approach a query is evaluated step by step. Each step produces a lot of elements which may be refused in the next evaluation step. Results of our experiments prove the previously published MDA path-based approach overcomes conventional element-based approaches. In our future work, we would like further to improve the abilities and the efficiency of MDA. In particular, we are going to develop an implementation of another complex XML querying such XPath and XQuery query languages defined it. We would like to use data types described by XML Schema for querying and develop an efficient implementation of approximate querying of XML documents.

References

1. Amphora Research Group (ARG). Amphora Tree Object Model (ATOM), `http://arg.vsb.cz/`, 2006.
2. R. Baeza-Yates and B. Ribiero-Neto. *Modern Information Retrieval.* Addison Wesley, New York, 1999.
3. R. Bayer. The Universal B-Tree for multidimensional indexing: General Concepts. In *Proceedings of World-Wide Computing and Its Applications'97 (WWCA'97), Tsukuba, Japan*, Lecture Notes in Computer Science. Springer–Verlag, 1997.
4. 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*, pages 322–331. ACM Press, 1990.
5. R. Bourret. XML and Databases, 2001, `http://www.rpbourret.com/xml/XMLAndDatabases.htm`.
6. A. B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems.* Addison Wesley Professional, 2003.

7. B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 341–350. Morgan Kaufmann, 2001.
8. R. Fenk. The BUB-Tree. In *Proceedings of 28rd VLDB International Conference on Very Large Data Bases (VLDB'02), Hongkong, China*. Morgan Kaufmann, 2002.
9. T. Grust. Accelerating XPath Location Steps. In *Proceedings of the 2002 ACM SIGMOD, Madison, USA*. ACM Press, June 4-6, 2002.
10. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, Annual Meeting, Boston, USA*, pages 47–57. ACM Press, June 1984.
11. M. Krátký, J. Pokorný, T. Skopal, and V. Snášel. The Geometric Framework for Exact and Similarity Querying XML Data. In *Proceedings of First EurAsian Conference, EurAsia-ICT 2002, Shiraz, Iran*, volume 2510 of *Lecture Notes in Computer Science*. Springer–Verlag, October 27-31, 2002.
12. M. Krátký, J. Pokorný, and V. Snášel. Implementation of XPath Axes in the Multi-dimensional Approach to Indexing XML Data. In *Current Trends in Database Technology, International Workshop on Database Technologies for Handling XML information on the Web, DataX, Int'l Conference on Extending Database Technology (EDBT 2004)*, volume 3268 of *Lecture Notes in Computer Science*. Springer–Verlag, 2004.
13. M. Krátký, J. Pokorný, and V. Snášel. Indexing XML data with UB-trees. In *Proceedings of Advances in Databases and Information Systems, ADBIS 2002, 6th East European Conference, Bratislava, Slovakia*, volume Research Commmunications, pages 155–164, September 8-11, 2002.
14. Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB'01)*. Morgan Kaufmann, 2001.
15. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *ACM SIGMOD Record*, 26(3):54–66, 1997.
16. J. Pokorný. *XML: a challenge for databases?*, pages 147–164. Kluwer Academic Publishers, Boston, 2001.
17. J. W. R. Goldman. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 436–445. Morgan Kaufmann, 1997.
18. A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April, 2001, `http://monetdb.cwi.nl/xml/`.
19. University of Washington's database group. The XML Data Repository, 2002, `http://www.cs.washington.edu/research/xmldatasets/`.
20. W3 Consortium. Extensible Markup Language (XML) 1.0, W3C Recommendation, 10 February 1998, `http://www.w3.org /TR/REC-xml`.
21. W3 Consortium. XQuery 1.0: An XML Query Language, W3C Working Draft, 12 November 2003, `http://www.w3.org/TR/xquery/`.
22. W3 Consortium. XML Path Language (XPath) Version 2.0, W3C Working Draft, 15 November 2002, `http://www.w3.org/TR/xpath20/`.
23. W3 Consortium. XML Schema Part 1: Structure, W3C Recommendation, 2 May 2001, `http://www.w3.org/TR/xmlschema-1/`.