Storing XML Data In a Native Repository

Kamil Toman

Dept. of Software Engineering
Charles University, Faculty of Mathematics and Physics
Malostranské náměstí 25
118 00 Praha 1
E-mail: `ktoman@ksi.mff.cuni.cz`

Abstract. This paper is concerned with storing XML data in a native repository suitable for querying with modern languages such as XPath or XQuery. It contains a description of the experimental database, SXQ-DB, its basic principles and system internals. Some of query evaluation techniques and problems related with those methods in relation to amount of stored information are mentioned.

1 Introduction

The XML language [1] was first published in 1998 but it has already become very popular. In the first place it is used as a standard for electronic interchange of application data and also as a flexible format allowing to store various information in a human readable form. With the expansion of Internet the data are gathered from various locations thus we cannot rely on their homogenity. On the contrary, we need to adapt applications to be able to handle them.

XML documents are logically formated documents which lessen the difference between pure text without any explicit formatting and rigidly structured data stored traditionally in relational databases.

Contents of XML documents are split up to smaller parts—elements—which are specifically named and which form one logical unit. From this point of view we can look on XML data as a database however it does not have a given hard-set structure and the structure of XML document itself provides a portion of complete information.

XML documents are often bound to their respective DTDs (*Document Type Definitions*). The purpose of DTD is to define the legal building blocks of an XML document. It defines possible structures together with a list of legal elements and attributes which might appear in the document. XML documents with a common DTD are called *document collections.*

To retrieve XML data from XML databases several new XML query languages have been proposed but only the minority survived. The most studied XML query languages with the most recent experimental implementations are XPath [6] and XQuery [5]. Both of them use *path expressions* as one of their basic constructs. Path expressions allow users to navigate through arbitrary paths of the XML tree and to address some portions of documents.

Despite many attempts to store XML data in relational, object-relational or object-oriented databases all existing approaches fail to supply sufficient functionalities to effectively manage and query XML data. Often, to process a simple path expression a sort of XML tree traversal is necessary. This might result in complex highly nested SQL queries which are hard to be effectively executed. Similarly in object-oriented databases, OQL is also not very suitable for expressing XML queries because it does not cover all basic constructs of XPath or XQuery.

In order to efficiently answer database queries the traditional DBMS leverage the usage of various indices. However, these index structures are tightly bound to a rigid database schema. That is something what, to some extent, prevents us from using them for XML indexing.

The database systems specially designed to store XML data are called *native XML repositories* or *native XML databases*. The storage is maximally adapted for tree shaped data contained in XML documents and as such it can be implemented in a practically arbitrary way. In comparison with traditional systems the index structures in native repositories are even more important. Often there is no way how to evaluate XML queries without a particular type of index. It does not, however, mean that a native XML database has to be implemented all over from the beginning. Some parts of such systems like the transaction manager, access control etc. can be often adopted from existing object or relational database systems with minimum changes.

In this text the new native XML database SXQ-DB (*Simple XQuery DataBase*) is described. In Section 2 the overall architecture and some implementation details of its XML storage module are discussed. Later on this section the query processing module is described. In Section 3 the brief overview how other systems process XML queries is presented. Conclusions summarize the contribution of this paper and give outlook to future work.

2 Native XML Database

SXQ-DB [3] is an experimental database suitable to store and manage collections of XML documents. Its current implementation consists of a native XML storage and a simple implementation of a non-trivial XML query language.

The goal of the work was to design a general and extensible architecture usable for testing various implementations of database operations and for verification basic qualities of the generic XML framework XMLCollection [2]. Unlike other projects the most important aspect was not the overall performance but the high-level design and the evaluation of the used data model with respect to more complex constructions of XML query languages. The accent was also on the fact that all accessed XML data were stored in the external memory.

As the XML query language has been implemented the language SXQ (*Simple XQuery*) which has been designed to cover the most important aspects of XQuery.

2.1 Overall Architecture

Due to given qualities and requirements of the initial XML framework the application operates on collections of XML documents characterized by a common DTD. These documents are stored in the external memory in a special binary format which is appropriate for more flexible access to data and hastens the effective successive processing.

The important decision on the implementation of the system was the choice of the modular architecture which allows an easy addition or a replacement of any system component.

Unlike programs managing XML files mostly in internal memory the module providing data querying is strictly separated from the actual XML repository. The Query Processing Module ensures syntactic analysis, processing and evaluating a query, the XML repository serves just for manipulation with persistent XML data. This design also allows the usage of several different XML storage modules without imposing other changes to the system. The overall architecture is depicted on Figure 1.

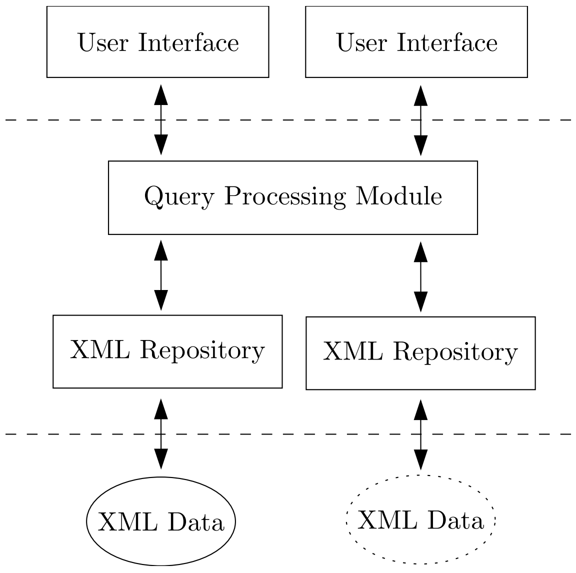

Fig. 1. Overall Architecture of SXQ-DB

2.2 XML Repository

The essential part of the SXQ-DB is the native XML repository implemented by utilizing the application framework XMLCollection. This module intermediates via an application interface information about data and structure of stored XML documents to other modules. It also provides access to individual elements and attributes stored in external memory.

The data repository also allows to limit the system resources—for example the size and number of system buffers or the maximal number of simultaneously accessed objects. Unlike other modules we put the accent on effectivity of data management.

Representation of XML Document. The used data model generally adheres the model as defined in XML Information Set [4] augmented by constructions defined by XQuery Data Model [5]. In our representation we use only those qualities which are required for evaluation of the SXQ language. In brief, we look on XML documents as oriented trees where to each vertex is associated a type of the node and a label. The vertices relevant to a common parent are ordered left-to-right. This ordering is imposed by the *global document order* requirement.

Text values are not assigned to all elements and attributes but only to special (artificial) nodes. This approach is primarily advantageous because all elements and attributes can be accessed the same way including elements with *mixed contents*.

Node Identification. In order to reasonably access individual nodes of XML tree we need to select a system of node identification. By a *numbering scheme* of the logical document model we understand a function which assigns a unique binary identifier (binary string) to each node of an XML tree. This identifier then can be used as a reference to the given node in an index or while a query evaluation.

The most common and also the simpliest numbering scheme in systems managing XML data is the *sequential* numbering scheme. The individual identifiers are assigned to particular nodes starting from one immediately after the nodes have been inserted to the system. The primary advantage of this scheme is its inherent simplicity and the maximal size of identifier which is at most $1 + \lfloor \log_2(n) \rfloor$ bits where n is number of nodes inserted to the system (including already deleted nodes).

The major disadvantage of the sequential numbering scheme is that it does not provide structural information – relations between nodes must be stored in separate.

The best way seems to be to leverage one of existing structural number schemes [7], [11], [14] which allow very effectively to determine the relation of arbitrary two nodes of the XML tree just from the information contained in their respective identifiers and thus they allow effective query evaluation using for example *structural joins* [11]. The maximum size of an identifier is still at most $2\lfloor 1 + \log_2(n) \rfloor$ bits.

On the other hand we have to take into account that we might need not only to change contents of individual XML nodes but also the document structure. Unfortunately all currently known structural numbering schemes are basically static – the numbering of nodes is based on the fact that we know or at least we assume the potential shape of the XML tree. If there are substantial differences from the anticipated XML data it is necessary to renumber the whole XML tree.

At the same time, neither contemporary techniques used for XML data management nor languages specially developed for XML document actualization like XUpdate [13] give us enough information about possible shapes of inserted subtrees or about scale of modifications to be done on the stored documents.

In general, it is possible to create a structural numbering scheme which is usable even in case when we know nothing about shapes of inserted trees. However, as proven in [14], the worst-case maximum size of resulting identifiers assigned to individual nodes is $O(n)$ bits.

Occasional renumbering of nodes in some XML subtrees does not imply an insurmountable problem because it can be partially prevented by a sensible utilization of structural statistics. Much bigger problem is that the renumbering of XML nodes imposes changes in practically all indexes that might be built up upon stored data. As XML indices are often relatively complex the total overhead related to their actualization might be enormous.

For this reason it is sensible to choose a compromise. For the direct numbering of nodes (in the core of XML repository) we use the basic sequential numbering scheme, plus we define a secondary numbering scheme (secondary identifiers) which will hold the structural information useful for quering.

In all indices we will use only references to primary node identifiers thus we avoid forced updates of indices if the structure changes. The disadvantage of this approach is slower evaluation of structural joins because of another level of logical mapping and also increased requirements for the disk space to manage primary and secondary identifiers.

Document Collections. We can also take advantage of the above approach to store the whole collection of XML documents at once. Because we use a simple sequential scheme as a primary identification mechanism the shape and the size of the tree does not pose a problem. Thus we can look on the whole collection as one XML document. It suffices to create two new special types of elements – the artificial root and document root elements. The abstraction of XML collection illustrates Figure 2.

The usage of the artificial collection root has one small additional advantage – it will always exist even if the collection is empty and can serve as a stable entry point with a fixed identifier.

Elements `DOCUMENT` allow mutual differentiation between individual XML documents and their attributes may also be used to keep user's information about stored documents like title, author, date etc. The information then can be later accessed even by the standard constructions of the query language.

Besides API simplification, the storage of the whole XML collection in one tree is often advantageous when the indices are built up. For example, if we create a word index it will be certainly more space efficient to build it up upon the whole collection than to build individual indices for every document in the

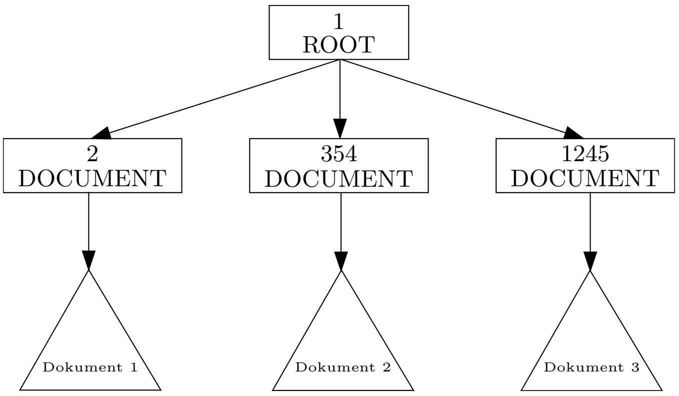

Fig. 2. Representation of XML Collection

collection. In case we want to query the whole collection at once it is also much more effective.

XML Node Types. Because we manage only XML documents with a common DTD we can also use numbers instead of labels for all types of elements and attributes (within different namespaces). The translation tables then can be stored together with DTD of the collection. This has two advantages: this approach to type identification is much more economical than storing text labels directly and it is also much easier and faster to use them during evaluation.

Architecture of XML Repository. The implemented XML repository can be logically divided into several modules. Each of them operates independently and ensures a different type of functionality:

– the DTD Storage module holds the DTD of the collection, name mapping tables, types and logical mapping of identifiers,
– the Element Storage module maintains the relations between XML nodes,
– the Value Storage module manages text values associated with elements and attributes,
– the rest of modules are mostly repository indices.

Strict module separation allows to hide unimportant details from the rest of the system. The communication between individual modules proceeds only via general application interface.

Naturally, this does not prevent us from using a common infrastructure. It is implemented as a separate module as well. The advantage is that potential changes in basic infrastructure can be done just once. The overview of the architecture of the XML repository module is demonstrated on Figure 3.

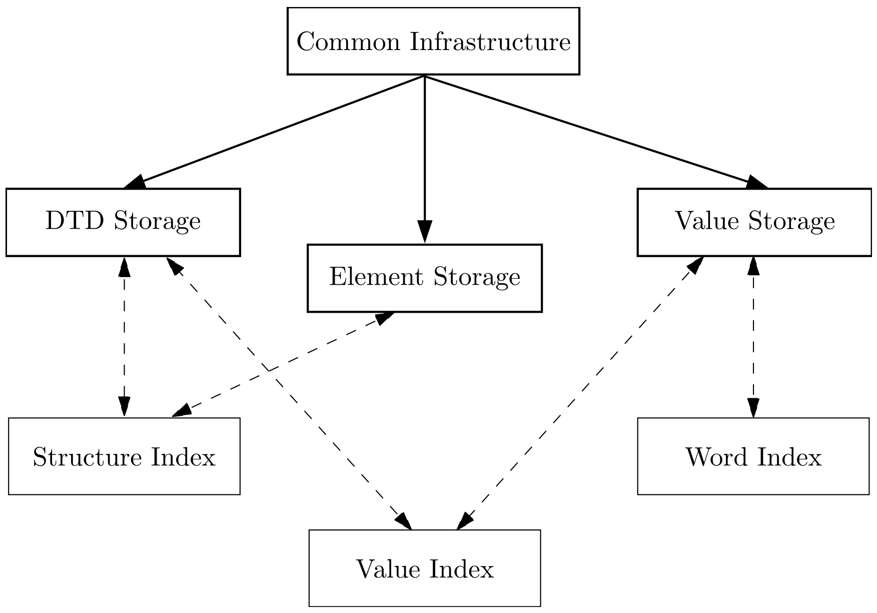

Fig. 3. Representation of XML Collection

The core modules cover only the basic functionality of the repository. For real utilization we have to take into consideration various indices as well. However, the indices might differ substantially from each other not only in implementation details but also by usage. It is hard to design a common interface which would allow a direct integration of an arbitrary index into the system.

To resolve this problem we expose only descriptions of events that may occur instead of detailed application interface between the core modules and indices – for instance a creation or a deletion of an XML node, its value actualization etc.

Each index is registered at particular modules so it can react to arisen events. That means we neither need to anticipate what data the index requires nor we need to know anything about its functionality. On the other hand we must ensure that the implementation of each index will be able to work with data structures of modules where it is registered.

Each index can propagate all events towards the rest of the modules which are built up on other than core modules of the system. Thanks to this mechanism we are able to sustain the whole system up-to-date.

Physical Access To External Memory. The important feature of the original implementation was to limit the system resources, the memory consumption in the first place. This property is ensured by the general *paging mechanism.*

Storage of Document Structure. The storage of a XML document structure is based on modeling local relations between XML nodes where every vertex “knows” only its direct neighbours. The representation of more complex relations is left to structural indices or to the query processing module.

The XML nodes are assigned indentifiers by the simple sequential numbering scheme and together with their types and identifiers of adjacent nodes are stored into fixed-length records in a binary file.

In order to access the nodes effectively we need to be able to quickly localize the information about individual nodes of the XML tree. To achieve this we index all records in a B^+-tree.

Secondary Object Cache. If there are some nodes which are accessed much more frequently during a query evaluation the above described method of locating nodes has still great overhead. The position of every located node must be at first looked up in the external index (possibly unbuffered) and then the respective position must be computed. The located page has to be loaded into main memory and requested data obtained from the computed offset.

For that reason a secondary object cache is implemented. Queries for information about an XML node are directed at first to this cache and only if it does not already contain the requested information the mechanism described above is used.

Notice that information about XML nodes is mostly short-lived. We often need to reach the record of the node just to find its neighbours or to check its value. If we implemented objects holding such information in a standard way frequent allocations and repeatedly released memory would kill the application performance. All cache objects are therefore kept in the main memory at all times and only if needed they are reinitialized with new data. The number of cache objects is given by the configuration of the XML repository module.

2.3 Query Processing Module

The XQuery language was created only recently and despite its basic features have their origin in previous proposals of XML query languages there is still no general technique how to evaluate all its queries. There is still nothing like relation algebra for classic relational database systems.

This is one of the reasons why classic navigational methods are still used for the evaluation of more general XML queries. More effective techniques like structural joins can be used only for special cases—mostly for path expressions and indispensable minimum of conditional expressions. Furthermore, expressions which can be evaluated by structural joins are very hard to distinguish from those which cannot be evaluated this way. Many of practical implementations avoid this problem simply by supporting only a limited set of XML queries [11], [20], [8]. The rest is modestly ignored.

Unlike other implementations our goal was to support all basic constructs of XPath and XQuery languages, not only path expressions. The implementation of SXQ language thus reminds a simple compiler of a general programming language. The architecture of the module and the individual phases of query processing is demonstrated on Figure 4.

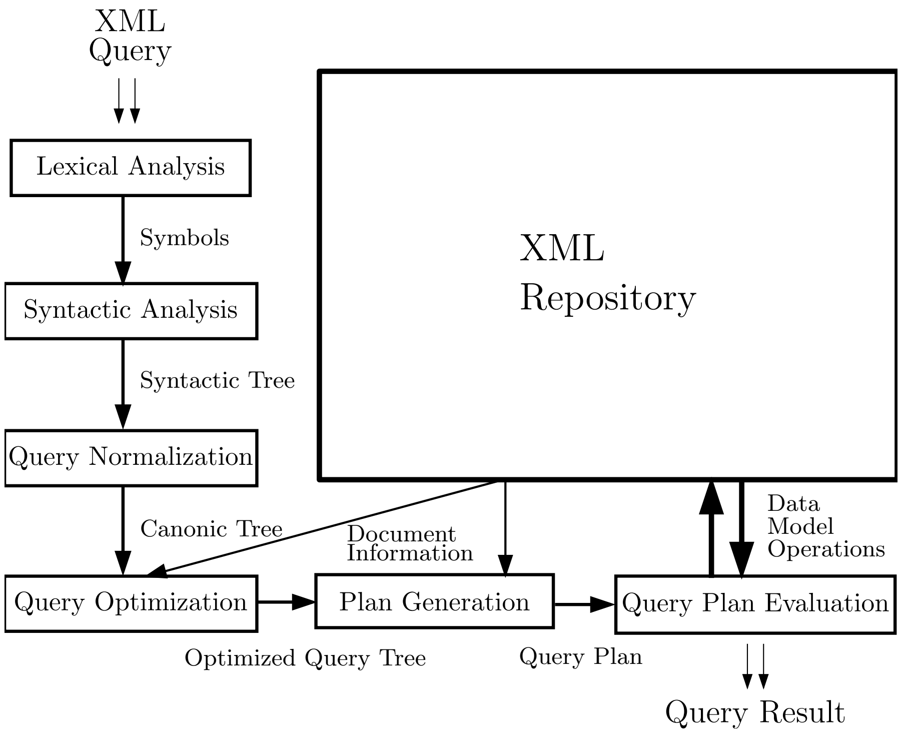

Fig. 4. SXQ Query Processing

At first the module disassembles the query to individual lexical elements which are subsequently used for syntactic analysis. In this phase context dependent keywords are resolved. In current implementation this is done via a finite automaton. The output of this stage is the syntactic tree which represents the independent form of a given query. However this tree is yet too complex for further processing. For that reason it is at first normalized into a *canonic tree.*

The canonic tree can be distilled from the syntactic tree by applying successive sequence of formal rewriting rules eliminating compound operations and "syntactic sugar", i.e. operations which are not indispensable and can be equivalently described using basic operations. Part of a query normalization is also a unification of expressions—for instance, we can reorder some constructs to adhere a fixed form though formally the expressions do not depend on the order of terms. The important thing is that the canonic tree is semantically equivalent to syntactic tree but substantially simplified and also with rather more rigid structure. The canonic tree is more suitable for logical optimization, apart from other things, because by tree normalization we radically decrease the number of different shapes an XML query tree might have.

Similarly, the logical optimization usually constitutes of a set of rewriting rules. Unlike the previous ones the goal is not to simplify the structure of the canonic tree but to reduce the time needed for the query evaluation. The rules are often heuristic but the processing time generally should not be much longer if the conditions were mispredicted. As an example of such a logical optimization we can mention e.g. invariant motion (a separation and a movement of some parts of the query away from a repeatedly evaluated expressions) or constraint motion (evaluation of constraints and conditions as soon as possible). The overview of such rewriting rules can be found in [12]. This phase might also include the elimination of common subexpressions of the query.

The logically optimized query tree is then passed to a generator of query plans. This module constructs possible procedures of query evaluation and accordingly to information supplied by XML repository it chooses the optimal plan.

This plan of query evaluation is consecutively executed by the computation engine which makes up the result of the query.

3 Query Processing In Other XML Database Systems

Tree pattern queries or correlated path expressions are the most accented constructs of XPath and XQuery querying languages. A pattern trees representing parent-child, ancestor-descendant relations between XML nodes bound with some additional constraints are to be matched against a source XML tree or a XML document collection.

The currently used evaluation techniques use extensive indices built mostly as combinations of structural path summaries [15], value indexing and tree traversal (Lore [16]) or identifier schemes (XISS [11]). However the storage efficiency is often not considered in these approaches.

Earlier systems relied on tree traversal techniques and structural indices like DataGuides or T-indices which are very inefficient when they are stored in the external memory. These methods have been surpassed with more modern structural joins (XISS, eXist [20]) which compose the tree patterns by pairwise matching parent-child and ancestor-descendant relations between candidate XML nodes. However the most commonly used indices used for structural joins can generally exceed the size of the whole source XML tree not giving any additional information besides the transitive ancestor-descendant relationship [17].

A few other indexing schemes like SphinX [18] or APEX [19] reduce the size of resulting indices by deliberately not covering all necessary information at the expense of generality or guaranteed performance. Though in practice they may perform quite well.

A novel approach of processing XML queries is being developed for project Timber [9] which is based on a complete and closed algebra named TAX which is a generalization of the current relational algebra for tree structures. The project still uses the old object manager Shore to manage the XML persistence but a transition to a native XML repository Natix [10] is planned.

4 Conclusions and Future Work

In this text, we described concepts and the implementation of SXQ-DB, the experimental native XML database. We demonstrated some advantages of its modular architecture and showed the basic data flow in the system. We also outlined some problems concerned with XML node insertions, numbering schemes and XML query evaluations, the tree pattern matching queries in the first place.

Future work in this area should probably be focused on two things: to find a more general way how to express and evaluate the most common XML queries and also to reduce space needed for structural and term indices used by the database application. Some recent more advanced proposals of XML indexing like multidimensional trees and UB-trees [21] are also subjects to be studied.

References

1. XML CoreWorking Group: Extensible Markup Language (XML). (2000) `http://www.w3.org/XML/`
2. M. Kopečný: Implementan prosted pro kolekce XML dat. Thesis (In Czech), MFF UK (2002)
3. K. Toman: XML data na disku jako databáze. Thesis (In Czech), MFF UK (2003)
4. J. Cowan, R. Tobin: XML Information Set. (2001) `http://www.w3.org/TR/xml-infoset`
5. M. Marchiori: XML Query Specifications. (2003) `http://www.w3.org/XML/Query#specs`
6. J. Clark, S. DeRose: XML Path Language (XPath) Version 1.0. (1999) `http://www.w3.org/TR/xpath`
7. P. F.'Dietz: Maintaining order in a linked list. Proc. of the Fourteenth Annual ACM Symposium on Theory of Computing: 122-127. (1982)
8. A. Sahuguet: Kweelt, the Making-of Mistakes Made and Lessons Learned. Technical report, Department of Computer and Science, University of Pensylvania. (2000)
9. H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, C. Yu: TIMBER: A Native XML Database. (2002) `http://www.eecs.umich.edu/db/timber`
10. C. Ch. Kanne, G. Moerkotte: Efficient Storage of XML Data. Poster abstract in Proc. ICDE: 198. (2000)
11. Q. Li, B. Moon: Indexing and Querying XML Data for Regular Path Expressions. VLDB Conference: 361-370 (2001)
12. M. Grinev, S. Kuznetsov: Towards an Exhaustive Set of Rewriting Rules for XQuery Optimization: BizQuery Experience Advances in Databases and Information Systems (2002)
13. XUpdate Working Group: XUpdate – XML Update Language. (2003) `http://www.xmldb.org/xupdate/`

14. E. Cohen, H. Kaplan, T. Milo: Labeling Dynamic XML Trees. Symposium on Principles of Database System (PODS): 271-281 (2002)
15. R. Goldman, J. Widom: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. VLDB Conference (1997)
16. J. McHugh, J. Widom, S. Abiteboul, Q. Luo, A. Rajaraman: Indexing Semistructured Data. Technical report, Stanford University (1999)
17. C. Zhang, G. He, D. J. DeWitt, J. F. Naughton: On supporting containment queries in relational database management systems. In SIGMOD International Conference on Management of Data: 425-436 (2001)
18. L. K. Poola, J. R. Haritsa: Schema-consious XML Indexing. Indian Institute of Science, Dept. of Computer Science & Automation (2001)
19. Ch.-W. Chung, J.-K. Min, K. Shim: APEX: An Adaptive Path Index for XML data. ACM SIGMOD (2002)
20. W. Meier: eXist: An Open Source Native XML Database. (2002) `http://exist-db.org`
21. M. Krátký, J. Pokorný, V. Snášel. Indexing XML Data with UB-Trees. ADBIS (2002)