INFORMATICA, 2012, Vol. 23, No. 2, 247–282
© 2012 *Vilnius University*

# Storing XML Data – The *ExDB* and *CellStore* Way in the Context of Current Approaches

Pavel LOUPAL[1], Irena MLÝNKOVÁ[2], Martin NEČASKÝ[2], Karel RICHTA[2], Pavel STRNAD[3]

[1]*Department of Software Engineering, Faculty of Information Technology, Czech Technical University in Prague*
*Thakurova nam. 9, 160 00 Praha 6, Czech Republic*
[2]*Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague*
*Malostranske nam. 25, 118 00 Praha 1, Czech Republic*
[3]*Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague*
*Karlovo nam. 13, 121 35 Praha 2, Czech Republic*
*e-mail: pavel.loupal@fit.cvut.cz, mlynkova@ksi.mff.cuni.cz, necasky@ksi.mff.cuni.cz, richta@ksi.mff.cuni.cz, pavel.strnad@fel.cvut.cz*

Received: December 2010; accepted: November 2011

**Abstract.** In this paper we describe possible approaches how to store XML data, which is a key aspect for their further processing. One popular technique for managing XML data is to map the data to an existing database system, e.g., to the relational or object-relational database management system. We describe possible ways how to store XML data in relational databases, because relational systems are still widely used for various purposes, including XML data management. But XML data are trees, not tables, so the main focus of this article is oriented to native XML databases. We describe general properties of such kind of databases and, in particular, explain possible solutions on two experimental native XML database management systems – *ExDB* and *CellStore*. Both have been proposed, implemented and optimized in our research groups in recent years for experimental purposes.

**Keywords:** XML data management, XML-enabled databases, native XML databases, *ExDB*, *CellStore*.

## 1. Introduction

With the growing popularity of XML, it is clear that there will be requests for large repositories of XML data. One popular technique for managing XML data is to map these data to an existing database systems, e.g., to the relational database management systems (RDBMS) or object-relational database management systems (ORDBMS). We speak about so-called *XML-enabled databases*. However, such a mapping often results in either an unnormalized relational representation or in a very large number of tables, due to the flexible nature of XML, with attributes and sub-elements frequently missing, and repetition of sub-elements being allowed. Therefore, the great challenge is to develop native XML databases, in which XML data can be stored directly, retaining its natural tree structure. We speak about so-called *native XML approaches* and respective native XML database management systems (NXDBMS). At the same time, we need in such native XML management systems all the benefits of relational database management, such as declarative querying and set-at-a-time processing. Such approaches have to use special indices, numbering schemas, and/or structures suitable particularly for tree structure of XML data. Expectably, the highest-performance techniques should be the native ones, since they are proposed particularly for XML processing and do not need to artificially adapt existing structures to a new purpose. On the other hand, the most practically used ones are methods which exploit features of (O)RDBMSs. The reason for their popularity is that (O)RDBMSs are still regarded as universal and powerful data processing tools which can guarantee a reasonable level of reliability and efficiency.

## 1.1. *Contribution*

In this paper we describe two different approaches to the problem of storing XML data. Either we can use existing resources and DBMSs, or we can create new tools. Hence, firstly, we provide a general overview of current approaches and strategies in both the areas and conclude it with a summary of features of current most popular implementations.

Since traditional (O)RDBMSs are complex systems having a long history and especially commercial support (Oracle Database, 2010; Microsoft SQL Server, 2008; DB2 Product Family), creating new native XML systems has a meaning only if they bring new features and a new quality. For these purposes it is necessary to carry out robust and reliable experiments to be able to compare these different approaches. In our research groups we have proposed, implemented and optimized two NXDBMSs – *ExDB* (Loupal, 2006) and *CellStore* (Vraný *et al.*, 2008). Both are based on similar ideas, but use a different environment – *ExDB* uses Java, *CellStore* uses Smalltalk. In the second part of this paper, we provide their description and comparison in the context of current best known storage strategies.

The aim of this paper is to provide both a general study of the current approaches to storing XML data and an overview and description of the two systems we have proposed and implemented.

## 1.2. *Outline*

The rest of the text is structured as follows: Section 2 introduces the features, advantages, and disadvantages of miscellaneous possibilities of exploiting general XML-enabled databases. You can find ways, how XML documents can be transformed into relational format. Section 3 deals with the basic features of native DBMSs and how they differ from relational systems. Section 4 contains description of the *ExDB* DBMS and Section 5 describes the *CellStore* DBMS. In Section 6 we describe benchmark configuration and comparison of the two systems. We have to deal with two technologies – Java and Smalltalk, each requiring different setup and behaving in a specific manner. Finally, conclusions with possible future research directions are attached in Section 7.

## 2. XML-Enabled Databases

Before we focus on the key aim of this paper – native XML approaches – we describe the currently most commonly used approaches to management of XML data, i.e., exploitation of (O)RDBMSs. Even though these approaches are proven to be less efficient than the native XML ones, they still have two unbeatable advantages – long theoretical and implementation history.

In general the basic idea of XML processing based on an (O)RDBMS is relatively simple. The XML data are firstly stored into relations – we speak about so-called *XML-to-relational mapping*. Then, each XML query posed over the data stored in the database is *translated* to a set of SQL queries (which is usually a singleton). And, finally, the resulting set of tuples is transformed to an XML document. We speak about *reconstruction* of XML fragments.

Consequently, the primary concern of the database-oriented XML techniques is the choice of the way XML data is stored into relations. On the basis of exploitation or omitting information from XML schema we can distinguish so-called *generic* and *schema-driven* methods. From the point of view of the input data we can distinguish so-called *fixed* methods which store the data purely on the basis of their model and *adaptive* methods, where also sample XML documents and XML queries are taken into account to find a more efficient storage strategy. And there are also techniques based on user involvement which can be divided to *user-defined* and *user-driven*, where in the former case a user is expected to define both the relational schema and the required mapping, whereas in the latter case a user specifies just local mapping changes of a default storage strategy.

Approaching the aim form another point of view the SQL standard has been extended by a new part SQL/XML (ISO/IEC 9075-14:2003, 2003) which introduces new *XML data type* and operations for XML and relational data manipulation within SQL queries. It involves functions such as, e.g., `XMLELEMENT` for creating elements from relational data, `XMLATTRIBUTES` for creating attributes, `XMLDOCUMENT` or `XMLFOREST` for creating more complex structures, `XMLNAMESPACES`, `XMLCOMMENT` or `XMLPI` for creating more advanced parts of XML data, `XMLQUERY`, `XMLTABLE` or `XMLEXISTS` for querying over XML data using XPath (Clark and DeRose, 1999; Berglund *et al.*, 2007) or XQuery (Boag *et al.*, 2007), etc.

As we have mentioned in the introduction, the native XML databases differ from the XML-enabled ones in the fact that they do not adapt an existing technology to XML, but exploit techniques suitable for XML tree structure. Most of them use a kind of numbering schema, i.e., an index that captures the XML structure. Examples of such schemas are *Dietz encoding* (Dietz, 1982), *Dewey encoding* (Tatarinov *et al.*, 2002), *interval encoding* (Li and Moon, 2001), *prefix encoding* (Cohen *et al.*, 2002), *ORDPATHS* (O'Neil *et al.*, 2004) or *APEX* (Chung *et al.*, 2002). (We discuss them in more detail in Section 3.) And such indices can be also exploited in relational databases to optimize query processing.

### 2.1. *Generic vs. Schema-Driven Methods*

*Generic mapping methods* (Florescu and Kossmann, 1999; Kuckelberg and Krieger, 2003) do not use (possibly) existing XML schema of stored XML documents. They are usually based on one of the following two approaches – creating a general (O)R schema into whose relations any XML document regardless its structure can be stored, or a special kind of (O)R schema into whose relations only a certain collection of XML documents having a similar structure can be stored. The former methods model an XML document as a tree $T$ according to, e.g., the DOM model (Document Object Model), while the latter reflect its special "relational" structure.

A typical representative of a generic mapping is a group of methods called *structure-centered mapping* (Kuckelberg and Krieger, 2003). It considers all nodes of the tree $T$ having the same structure defined as a tuple $v = (t, l, c, n)$, where $t$ is the type of the node (e.g., element, attribute, text, $\ldots$), $l$ is the node label, $c$ is the node content and $n \in \{1, \ldots, n\}$ is the list of successor nodes. The paper considers the problem how to realize mapping of the lists of successor nodes. It proposes several kinds of storage strategies focusing on speeding up the performance of access. In *Foreign Key Strategy* each tree node $v$ is simply mapped to a tuple with a unique identifier and a foreign key reference to the parent node. The method is quite simple and the stored tree can easily be modified. Nevertheless, its disadvantage is evident – the retrieval of the data involves many self-join operations. In *Depth First (DF) Strategy*, conversely, each node of $T$ is given an index value (a couple of minimum and maximum DF values), which represents its position in $T$. The DF values are determined when traversing $T$ in a depth first manner. A counter is increased each time another node is visited. If a node $v$ is visited for the first time, its minimum DF value $v_{\min}$ is set to the current counter value. When all child nodes have been visited, the maximum DF value $v_{\max}$ is set to the current counter value (see Fig. 1).

Using DF values relationships of nodes (e.g., sibling order, element-subelement relationship, etc.) can easily be determined just by comparisons. For example, a node $v$ is a descendant of node $u$, if $u_{\min} < v_{\min}$ and $v_{\max} < u_{\max}$. Moreover, as the nodes can be totally ordered according to DF values, retrieving a part of a document is linear. The weak point of this strategy is document update – in the worst case it requires to update DF values of all nodes of the tree.

On the other hand, *schema-driven mapping methods* (Shanmugasundaram *et al.*, 1999; Runapongsa and Patel, 2002) are based on an existing schema $S_1$ of stored XML documents, written in DTD (Bray *et al.*, 2006) or XML Schema (Thompson *et al.*, 2004; Biron and Malhotra, 2004), which is mapped to (O)R database schema $S_2$. The data from XML documents valid against $S_1$ are then stored into relations of $S_2$. The purpose of these methods is to create optimal schema $S_2$, which consists of reasonable amount of relations and whose structure corresponds to the structure of $S_1$ as much as possible.

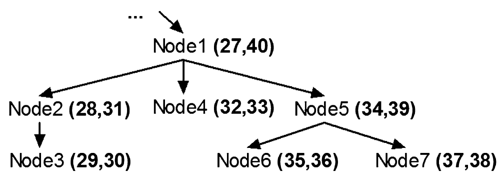


Fig. 1. An example of a generic-tree.

```
<!ELEMENT author(name?,surname)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT surname(#PCDATA)>
<!ELEMENT book(author*,title)>
<!ATTLIST book published CDATA>
<!ELEMENT title(#PCDATA)>
<!ELEMENT article(author)>
<!ATTLIST article paper CDATA>
```

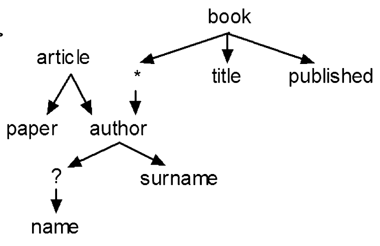


Fig. 2. An example of a DTD graph.

All of these methods try to improve the basic mapping idea "to create one relation for each element composed of its attributes and to map element-subelement relationships using keys and foreign keys".

Schema-driven mapping methods have several common basic principles resulting from information stored in the XML schema. The most important ones are:

- Subelements with `maxOccurs = 1` are (instead of to separate tables) mapped to tables of parent elements (so-called *inlining*).
- Elements with `maxOccurs > 1` are mapped to separate tables (so-called *outlining*). Element-subelement relationships are mapped using keys and foreign keys.
- Alternative subelements are mapped to separate tables (analogous to the previous case) or to one universal table (with many nullable fields).
- If it is necessary to preserve the order of sibling elements, the information is mapped to a special column.
- Elements with mixed content are usually not supported.
- A reconstruction of an element requires joining several tables.

The best-known and probably the first representative of schema-driven mapping methods is a group of three algorithms for mapping a DTD to relational schema called *Basic*, *Shared*, and *Hybrid* (Shanmugasundaram *et al.*, 1999). The main idea, further used in all the successive approaches, is based on a definition of a directed graph, so-called *DTD graph*, which represents the processed DTD. Nodes of the graph are elements (which appear exactly once), attributes, and operators (which appear as many times as in the DTD). Edges of the graph represent element-attribute, element-subelement or element-operator and operator-subelement relationships (see Fig. 2).

The algorithms try to gradually improve the idea "to create one relation for each element" and they differ according to the amount of redundancy they may cause.

## 2.2. *Fixed vs. Adaptive Methods*

All the previously described approaches represented so-called *fixed* methods, i.e., methods which provide the target mapping regardless the target application. *Adaptive* methods (Klettke and Meyer, 2001; Bohannon *et al.*, 2002; Xiao-Ling *et al.*, 2003; Zheng *et al.*, 2003) focus on the idea that each application, represented using sample data and operations (i.e., queries), requires a different storage strategy to achieve optimal efficiency. So before they provide the resulting mapping, they analyze the given sample data and operations and adapt the target schema to them.

A representative of flexible schema-driven mapping methods is an algorithm proposed in system *LegoDB* (Bohannon *et al.*, 2002). First the method defines a fixed mapping of XML Schema structures (for processing simplicity rewritten into syntactically simpler, but semantically equivalent $p$-schemas) to relations. The flexibility is based on the idea to explore a space of possible XML-to-relational mappings and to select the best one according to given statistics including information about a sample set of XML documents and queries. In order to select the best mapping the system in turns applies the following two steps to the source $p$-schema, until a good result is achieved:

1. Any possible XML-to-XML transformation is applied to the $p$-schema.
2. XML-to-relational transformations are applied to the new $p$-schema and against the resulting relational schema the given queries are estimated.

As the space of possible $p$-schemas can be large (possibly infinite), the paper also proposes a greedy evaluation strategy that explores only the most interesting subset. The XML-to-XML transformations used in the algorithm are: *inlining/outlining*, *union factorization/distribution*, *repetition merge/split*, *wildcards rewriting*, etc. The XML-to-relational transformations are similar to those described in the previously mentioned fixed methods.

## 2.3. *User-Defined vs. User-Driven Methods*

Both user-defined and user-driven approaches are based on the same idea as adaptive methods, i.e., to provide a target schema which is optimal for a particular application. However they achieve this aim using a different strategy – "to leave the whole process in hands of a user". *User-defined* (Amer-Yahia, 2003) mapping methods were the first approaches supported by the commercial systems, probably due to simple implementation. This approach requires that the user first defines $S_2$ and then expresses required mapping between $S_1$ and $S_2$ using a system-dependent mechanism, e.g., a special query language, a declarative interface, etc. At first sight the idea is correct – users can decide what suits them most and are not restricted by features and especially disadvantages of a particular technique. The problem is that such approach assumes users skilled in two complex technologies – (object-)relational databases and XML. Furthermore, for more complex applications the design of an optimal relational schema is generally an uneasy task.

At present, most of existing systems support a kind of *user-driven mapping* (Balmin and Papakonstantinou, 2005; Amer-Yahia *et al.*, 2004; Mlýnková, 2007) where the effort a user is expected to make is lowered. The main difference is that the user can influence a default fixed mapping strategy using annotations which specify the required mapping for particular schema fragments. The set of allowed mappings is naturally limited but still enough powerful to define various mapping strategies. Each of the techniques is characterized by the following four features:

- an initial XML schema $S_{init}$,
- a set of allowed fixed XML-to-relational mappings $\{f^i_{map}\}_{i=1,\ldots,n}$,
- a set of annotations $A$, each of which is specified by name, target, allowed values, and function, and
- a default mapping strategy $f_{def}$ for not annotated fragments.

Probably the first approach which faces the mentioned issues is proposed in system *ShreX* (Du *et al.*, 2004). It allows users to specify the required mapping and it is able to check *correctness* and *completeness* of such specifications and to complete possible incompleteness. The mapping specifications are made by annotating the input XML Schema definition with a predefined set of annotations, i.e., attributes from namespace called `mdf`. The set of annotating attributes $A$ is listed in Table 1.

As we can see, the set of allowed XML-to-relational mappings $\{f^i_{map}\}_{i=1,\dots,n}$ involves inlining and outlining of an element or an attribute, *Edge mapping* (Florescu and Kossmann, 1999) strategy, and mapping an element or an attribute to a CLOB column. Furthermore, it enables one to specify the required capturing of the structure of the whole schema using one of the following three approaches:

- *Key, Foreign Key, and Ordinal Strategy (KFO)* – each node is assigned a unique integer ID and a foreign key pointing to parent ID, the sibling order is captured using an ordinal value
- *Interval Encoding* – a unique `{start,end}` interval is assigned to each node corresponding to preorder and postorder traversal entering time
- *Dewey Decimal Classification* – each node is assigned a path to the root node described using concatenation of node IDs along the path

As side effects can be considered attributes for specifying names of tables or columns and data types of columns. Not annotated parts are stored using user-predefined rules, whereas such mapping is always a fixed one.

Table 1

Annotation attributes for ShreX

| Attribute | Target | Value | Function |
|---|---|---|---|
| `outline` | Attribute or element | `true`, `false` | If the value is `true`, a separate table is created for the attribute/element. Otherwise, it is inlined to parent table. |
| `tablename` | Attribute, element, or group | `string` | The string is used as the table name. |
| `columnname` | Attribute, element, or simple type | `string` | The string is used as the column name. |
| `sqltype` | Attribute, element, or simple type | `string` | The string defines the SQL type of a column. |
| `structurescheme` | Root element | `KFO`, `Interval`, `Dewey` | Defines the way of capturing the structure of the whole schema. |
| `edgemapping` | Element | `true`, `false` | If the value is `true`, the element and all its subelements are mapped using Edge mapping. |
| `maptoclob` | Attribute or element | `true`, `false` | If the value is `true`, the element/attribute is mapped to a CLOB column. |

## 2.4. *Comparison of XML Enabled Databases*

To conclude this section we provide a comparison of support of XML technologies and respective strategies in the current most popular XML-enabled databases – *Oracle DB* (Oracle Database, 2010), *IBM DB2* (DB2 Product Family), and *Microsoft SQL Server* (Microsoft SQL Server, 2008), sometimes denoted as the "Big Three". Table 2 provides an overview of the XML-related functions that are (not) supported in the three systems and their key characteristics.

As we can see, in general, all the three vendors follow the same pattern and try to support as much XML functionality as possible. The most advanced and, at the same time, standard-conforming support has *Oracle DB*, whereas the *SQL Server* traditionally ignores the proposed standards the most. Under a closer investigation we can see that there are some significant observations and differences in respective areas of XML support.

Firstly, all three systems support a kind of XML data type (called either `XML` or `XML-Type`) and several types of respective storage strategies. In particular, the XML data type can be naturally stored into a kind of `LOB` (we speak about a *non-structured* storage), although such kind of storage is trivial and suitable only for a specific type of applications. Hence, there are two other types of storage strategies – shredding into relations (called *structured*) and native XML storage (called *binary* or *native*). In the former case usually a

Table 2
Overview and comparison of key XML features of selected XML enabled databases

| Feature | *Oracle DB* | *DB2* | *SQL Server* |
|---|---|---|---|
| XML data type | `XMLType` | `XML` | `XML` |
| | Structured, binary, non-structured | Binary, structured | `LOB`, native, structured |
| Mapping | User-defined | User-defined | User-defined |
| | Names, data types and storage strategies (`VARRAY` vs. `LOB`) | Relations, columns, conditions, expressions | Relations, columns, keys, relationships |
| Querying | XQuery, SQL/XML | XQuery, SQL/XML, SQL embedded to XQuery | XQuery, SQLXML (`OPENXML`, `FOR XML`) |
| Indexing | `XMLIndex` | Region index, column path index, XML index | Primary, secondary (`PATH`, `PROPERTY`, `VALUE`), full-text |
| | ORDPATHS, path index, axes index | Indexing particular XPath expressions | ORDPATHS |
| Other operations | Validity checking, XSL transformations | Validity checking, XSL transformations | Validity checking, XSL transformations only via an external tool |
| Updating | Own functions for inserting, replacing, deleting nodes | XQuery Update Facility | Own function with parameter for inserting, replacing, deleting nodes |

kind of user-driven mapping is supported, where the user can specify, e.g., names of relations and columns, data types of columns, decide on inlining/outlining or storing a whole XML fragment into a LOB column, types of storage strategies or various additional SQL conditions and expressions.

As for the query capabilities, all the systems support the XQuery language and its embedding in SQL to enable working with both XML and SQL data at the same time. To further increase this ability, both *Oracle DB* and *DB2* support the SQL/XML standard, while *SQL Server* provides own set of functions called SQLXML (which involve constructs `OPENXML` and `FOR XML` providing SQL views over XML data and XML views over SQL relations respectively). Surprisingly, *DB2* also supports a new feature – embedding SQL queries into XQuery – which is not a part of the SQL/XML standard. All the three systems support several kinds of indices that enable to speed up XML querying. Naturally, they are highly related to the selected storage strategy. If structural storage is selected, classical $B^+$ trees are used. If a native storage is used, native XML indeces (such as *ORDPATHS* (O'Neil *et al.*, 2004)) are exploited. All the systems also support a kind of XML full-text search, requiring respective indices as well.

Considering other operations with XML data, all systems naturally support validity checking and XSL transformations. On the other hand, considering update operations, each of them has its own approach. *Oracle DB* provides a set of update functions, *SQL Server* provides a single function with multiple parameters and only *DB2* already supports the XQuery Update Facility.

As we can see, all the three current database leaders try to exploit the relational aspects as much as possible. However, at the same time, they realize that native approaches are more suitable for semi-structured XML data and, hence, try to extend the systems towards these strategies. We provide their overview in the following section.

## 3. Native XML Databases

A *native XML database system* is a database system whose internal structure is especially designed for XML data management. XML data are stored in a format which is maximally adapted according to specific characteristics of XML data – hierarchical and irregular structure potentially mixed with unstructured data. Advantages of an NXDBMS in comparison to storing XML data into an (O)RDBMS are obvious. An XML document may be stored "as it is" without complicated transformation into relational tables and, therefore, may be efficiently retrieved from the system in its original form. Moreover, an NXDBMS directly supports XML query languages such as XPath and XQuery and, therefore, it is not necessary to translate XML queries to SQL queries.

On the other hand, there are also fundamental disadvantages of NXDBMSs we need to count with. In comparison to RDBMS technologies, NXDBMS technologies are relatively new and, therefore, not so well developed. Therefore, NXDBMSs rarely provide a support for, e.g., transactions or advanced query optimizers.

Contrary to relational queries, XML queries deal with not only data items of stored XML documents but also structural relationships between their XML nodes. Therefore, an NXDBMS can exploit well-known query evaluation algorithms from the theory of relational databases. However, they also need novel algorithms for evaluating structural queries. The most important kind of these algorithms widely studied in current literature are so-called *structural join algorithms*. In this section, we describe the most important representatives of these algorithms. Before this we introduce the notion *numbering schema* which is for structural joins crucial.

## 3.1. *Numbering Schemas*

Obviously, the amount of XML data might be extensive and an NXDBMS cannot store it all in the main memory. It is, therefore, necessary to identify the stored XML nodes so that we can locate them on the disc. The identification system is generally called *numbering schema*.

The simplest numbering schema is a *sequential numbering schema*. It numbers XML nodes starting from $1$ and each new XML node is assigned with the last assigned identifier increased by $1$. The advantage of this numbering schema is its simplicity. It does not require any complex management by the database system. On the other hand, it does not provide any assistance to the database system when evaluating queries. As we have noticed, XML queries are not evaluated only on the basis of data values of XML nodes, but also on the basis of structural relationships between them. The problem is that a sequential numbering schema does not provide such kind of information. Having numbers of two XML nodes, we need to traverse the XML document to find out whether, e.g., the nodes are in the ancestor/descendant relationship.

To solve the problem with structural information, various kinds of numbering schemas, called *structural numbering schemas* were introduced in the literature. They are designed to quickly recognize whether two given XML nodes are in the ancestor/descendant relationship. Formally, a structural numbering schema is a pair $(p, L)$, where $L$ is a function which assigns each XML node $v$ with a number $L(v)$ and $p$ is a predicate such that $p(L(u),\ L(v)) = \texttt{true}$ iff $u$ is an ancestor of $v$.

An example of a structural numbering schema is *Dietz schema* (Dietz, 1982). It assigns each XML node $v$ with a pair $L(v) = (\mathit{pre}(v), \mathit{post}(v))$, where $\mathit{pre}(v)$ returns the order of $v$ in the pre-order traversal of the XML tree while $\mathit{post}(v)$ returns its order in the post-order. Having two nodes $u$ and $v$, $p(L(u),\ L(v)) = \texttt{true}$ iff $\mathit{pre}(u) < \mathit{pre}(v)$ and $\mathit{post}(u) > \mathit{post}(v)$. In other words, $u$ is an ancestor of $v$ iff $u$ appears earlier than $v$ in the pre-order traversal than $v$ and, vice versa, later than $v$ in the post-order traversal.

As shown, Dietz schema allows for deciding the structural ancestor/descendant relationship effectively. A problem arises when management of identifiers comes to the scene – inserting a new XML node into an XML tree affects Dietz identifiers of the ancestors of the new XML node, as well as all XML nodes which are after it in the pre-order traversal. These affected nodes must be renumbered which may be a non-trivial and time-consuming task. A solution to this problem is to use an *interval schema*. It assigns each XML node $v$ with an interval $L(v) = (\mathit{start}(v), \mathit{end}(v))$, such that $L(v)$ is contained in the interval $L(u)$ for each ancestor $u$ of $v$ and two intervals of any sibling nodes do not

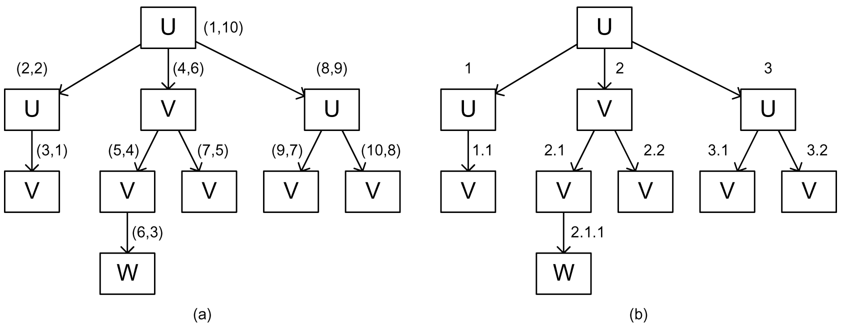


Fig. 3. An example of XML document with nodes numbered by (a) Dietz and (b) Dewey encoding.

overlap. The intervals may be "blew up" so that there is a space for new incoming nodes without the necessity of renumbering the existing ones.

Another partial solution to the node insertion problem provides *Dewey encoding* (Tatarinov *et al.*, 2002). It assigns each non-root XML node $v$ with a code $L(v) = L(u).position(v)$, where $u$ is the parent of $v$ and $position(v)$ is the position of $v$ among the children of $u$. The root XML node is assigned with an empty code. Insertion of a new XML node affects the codes of following siblings and their descendants which is better than in the case of previous schemas. However, the resulting codes are longer and with a variable length. A solution to this problem may be found in (Bača *et al.*, 2010).

A sample XML document with nodes numbered by Dietz and Dewey encoding schemas is depicted in Fig. 3.

## 3.2. *Structural Join Algorithms*

As we have outlined, an NXDBMS strongly depends on ability to evaluate queries which query structural relationships between nodes. For example, an XPath query is typically a structural query. Its simplest form searches for XML elements or attributes with specified names and with specified structural relationships between them. In this section, we consider only ancestor-descendant (AD) and parent-child (PC) relationships.

When evaluating a structural query, the query is firstly represented as a tree $Q$ called *twig pattern*. Its nodes have names and represent the queried nodes in XML documents. Its edges represent the required structural relationships and are, therefore, of two types: AD and PC. A structural join algorithm then searches for all occurrences of $Q$ in a given XML tree (or XML trees). An occurrence of a query $Q$ with nodes $q_1, \dots, q_n$ is a tuple $u_1, \dots, u_n$ of XML nodes so that $u_i$ has the same name as $q_i$ and each pair $u_i$ and $u_j$ is a PC or AD structural relationship iff there is an edge of that type between $q_i$ and $q_j$, respectively.

In general, each structural join algorithm works as follows: It assigns each node $q$ in the twig pattern $Q$ with an ordered stream of XML nodes from the input XML document. The sequence contains only XML nodes with the name equal to the name of $q$. The algorithm then sequentially reads the input streams and searches for twig pattern occurrences. It is clear that the crucial property of any structural join algorithm is its ability to determine whether two given nodes are in PC or AD relationship. This might be achieved easily by selecting a suitable numbering schema, e.g., the ones introduced in Section 3.1.

Occurrences of a twig pattern in an XML document may be searched in various ways. Each way has significant advantages but drawbacks as well. The majority of approaches firstly separate $Q$ into smaller components, evaluate these components individually and then merge the intermediate results into the final output. We distinguish two groups of such approaches. Approaches in the first group evaluate each edge in $Q$ separately. This group is called *binary structural join algorithms* as each edge specifies a binary structural relationship. Approaches in the second group evaluate each root-to-leaf path separately. This group is called *holistic structural join algorithms*. Finally, there are also algorithms which evaluate $Q$ as a whole. This group is called *one-way structural join algorithms*.

### 3.2.1. *Binary Structural Join Algorithms*

The simplest idea to evaluate a twig pattern $Q$ is to evaluate each edge of $Q$ separately and then merge the intermediate solutions to the final solution. We speak about so-called *binary structural join algorithms*. The approaches concentrate solely on the first part of the problem, i.e., finding the intermediate solutions. Merging intermediate solutions is not so interesting since it can be solved by classical merging algorithms. Historically, binary structural joins algorithms represent the oldest and already obsolete algorithms. On the other hand, their principles have strongly influenced state-of-the art holistic structural join algorithms.

The first attempt in this area was introduced in Zhang *et al.*, (2001). It showed that an algorithm specially designed for twig query matching can significantly outperform classical relational join algorithms. In Al-Khalifa *et al.* (2002), the authors introduced a binary structural join algorithm `STACK-TREE` which put basics of many later algorithms. It sequentially reads two input streams $T_u$ and $T_v$ associated with two nodes $q_u$ and $q_v$ connected by an evaluated edge $e$ from $Q$. For the current XML nodes $C_u$ and $C_v$ from the input sequences, it decides whether they are in the required PC or AD relationship. If so, it puts the found pair on the input. For a given XML node from $T_u$ the algorithm searches for all nodes in $T_v$ which form an occurrence. A problem occurs when there are two nodes $u_1$ and $u_2$ in $T_u$ s.t. $u_1$ is an ancestor of $u_2$. In that case, all XML nodes from $T_v$ which form an occurrence with $u_1$ might also form an occurrence with $u_2$ and, therefore, a part of $T_u$ must be accessed twice. Therefore, the authors proposed to use a stack to cache nodes from $T_u$ which are in the AD relationship. This prevents from repeated traversal of $T_u$.

The authors showed that `STACK-TREE` is time and space optimal in case of twig patterns with AD edges only. If a twig pattern contains also a PC edge, the time complexity degrades. This is because `STACK-TREE` can join XML nodes only on the AD relationship and, then, it must check whether they are also in the PC relationship. However, the optimal algorithm would skip the nodes which are not in PC relationship without joining them. This problem was partially solved for holistic structural join algorithms and we describe the solutions later in this section.

Another disadvantage of `STACK-TREE` is that it may read XML nodes in $T_u$ or $T_v$ which cannot form an occurrence. It reads them from the disc and skips them. It would be more optimal if it would not be necessary to access them at all. This can be achieved by using a suitable indexing structure. There were introduced several solutions which allow us to skip XML nodes in $T_u$ to the first ancestors of $C_v$ and, vice versa, to skip XML nodes in $T_v$ to the first descendant of $C_u$. The first attempt in this area used a classical $B^+$ tree (Chien *et al.*, 2002). More optimal solutions are XR-tree (Jiang *et al.*, 2003b) and XB-tree (Bruno *et al.*, 2002) which are used also for holistic structural join algorithms.

### 3.2.2. *Holistic Structural Join Algorithms*

All binary structural join algorithms can solve only a binary relationship which is usually a part of a more complex twig pattern. However, only parts of the intermediate solutions of the binary relationships contribute to the final solution of the whole twig pattern. Therefore, these algorithms can produce unusable intermediate results. This behavior can be partly minimized by selection of an appropriate order of the joins (Wu *et al.*, 2003). However, such a solution needs expensive statistics about the XML documents.

This problem is partly solved by another family of structural join algorithms called *holistic structural join algorithms*. (Note that algorithms in this family still do not evaluate a twig pattern as a whole.) They are based on the idea of *decomposition* of the twig pattern to root-to-leaf paths. Even though they can produce intermediate occurrences that do not contribute to the final solution, the intermediate solutions are much smaller than in case of binary structural join algorithms. Moreover, there have been proposed various techniques that further minimize them.

In Bruno *et al.* (2002), the authors proposed two holistic structural join algorithms called `PATH-STACK` and `TWIG-STACK`. This was the first work which introduced the family of holistic structural joins. `PATH-STACK` directly extends binary `STACK-TREE`. More specifically, it extends the idea of caching intermediate XML nodes which possibly contribute to any occurrence of the twig pattern. The caching is realized in a stack $S_q$ assigned to each twig pattern node $q$. The algorithm reads all input streams sequentially and stores those which are in the AD relationship on the corresponding stacks similarly to `PATH-STACK`. When an XML node corresponding to a leaf node of the twig pattern is found, occurrences containing this XML node are reconstructed by combining XML nodes on the stacks.

`PATH-STACK` is optimal when evaluating a twig pattern without branching nodes (i.e., a twig pattern which has a form of path). `TWIG-STACK` introduces an optimization. When an XML node corresponding to a twig pattern node $q$ is found, it is not directly put on $S_q$. Instead, `TWIG-STACK` checks whether it is in AD relationship with all current nodes in the input streams corresponding to twig pattern child nodes of $q$. This prevents from processing XML nodes which cannot participate in any occurrence. `TWIG-STACK` is optimal when evaluating twig patterns with AD edges only.

Later there appeared extensions to `TWIG-STACK` which optimize it such as, e.g., `TSGeneric` (Jiang *et al.*, 2003a), `TWIG-STACK-LIST` (Lu *et al.*, 2004), or `TWIG-BUFFER` (Li and Wang, 2008b). These extensions provide various techniques which allow for evaluation of twig patterns which contain PC edges on specific positions. However, they still cannot evaluate a general twig pattern with PC edges at an arbitrary position optimally.

### 3.2.3. *One-Way Structural Join Algorithms*

In Chen *et al.* (2006), another family of structural join algorithms was introduced. It overcomes the main drawback of holistic structural join algorithms – the necessity to decompose a given twig pattern to root-to-leaf paths and merging their intermediate results. More specifically, the work introduces algorithm TWIG$^2$STACK which extends the idea of stacks by so-called *hierarchical stacks*. It is then able to store a partial occurrence of a twig pattern as a whole on the hierarchical stacks without decomposition to root-to-leaf paths. The advantage is clear – the merging phase is reduced. However, it might be necessary to hold the whole XML document in the hierarchical stacks.

Later, an algorithm called TWIG-LIST (Qin *et al.*, 2007) was introduced. It optimizes TWIG$^2$STACK by replacing hierarchical stacks with direct pointers to input streams of XML nodes. This reduces the space complexity and allows for easier management. There have also appeared further optimizations in Jiang *et al.* (2007) or Li and Wang (2008a) which are based on combining one-way structural join algorithms with the holistic ones.

## 3.3. *Indexing Structures*

The effectiveness of any structural join algorithm depends on the way how the data is stored and indexed by an NXDBMS. As we have already showed, each structural join algorithm requires an ordered input stream of XML nodes of a given name associated with each twig pattern node. It is therefore necessary to store the XML nodes on the disc in a way which allows us to retrieve them in a form of the input streams. In this section, we discuss an index structure called *DataGuide* which is designed for this purpose. We also discuss alternative indexing techniques which help in particular situations when evaluating XML queries.

### 3.3.1. *DataGuide*

*DataGuide* was one of the first NXDBMS-specific indexing structures. It allows for indexing structure of XML documents. More specifically, a DataGuide of an XML document is a tree. Its each node represents a single root-to-leaf path of XML node names in the XML document. Its each edge represents that XML nodes on the path represented by the parent are parents of the XML nodes on the path represented by the child. A DataGuide for the sample XML document in Fig. 3 is depicted in Fig. 4.

For each of its nodes a DataGuide indexes a sequence of XML nodes on the path represented by the node. For each indexed XML node, the DataGuide indexes the identification number assigned to the XML node by the chosen numbering schema. It then allows for providing structural join algorithms with required input streams of XML nodes. In the basic version, XML nodes with a given name are put into a common stream. However, a DataGuide allows for more advanced streaming schemas. For example, it may provide a separate stream for each of its nodes. In other words, XML nodes targeted by the same root-to-leaf path of names are put into a common stream. As shown in Chen *et al.* (2005), this improves the time complexity of structural join algorithms when evaluating twig pattern PC edges. It is also possible to reduce the space complexity by stream compression as shown in Bača *et al.* (2010).

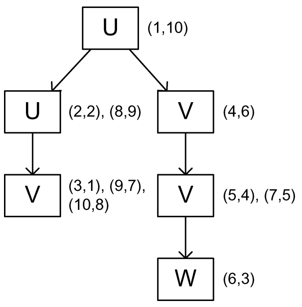


Fig. 4. An example of DataGuide.

### 3.3.2. *Covering Indices*

A *covering index* is an index which allows for evaluating queries of a particular type without accessing the source data on the disc. This kind of indices may be also found in RDBMSs but there are also equivalents in NXDBMSs. For example, a DataGuide is a covering index for queries whose twig patterns do not contain branching nodes. These queries may be evaluated directly by searching for respective node in the DataGuide and returning the associated XML node stream.

Having a query whose twig pattern contains branching nodes, we can separate the twig pattern to root-to-leaf paths, evaluate them using the DataGuide and then join them using a structural join algorithm. Another possibility is to exploit a stronger index which covers not only twig patterns in a form of paths but twig patterns in general (i.e., with branching nodes). This index is called *F&B index* (Kaushik *et al.*, 2002). F&B index is a DataGuide constructed over source XML documents complemented with reversed edges of the original source edges.

In practice, F&B index may be, however, extremely large – even larger than the original data. On the other hand, the authors show in Kaushik *et al.* (2002) that F&B is the smallest possible index covering queries whose twig patterns contain branching nodes. The problem with the size of F&B index may be, therefore, solved only by restricting the index to cover only specific kinds of queries. For example, we might index only selected paths in the XML documents by an F&B index. This is analogical to indexing tables in an RDBMS, where it is a common practice to index only selected table columns instead all columns and their combinations. It is up to the database administrator to decide what paths should be indexed. The overall performance of NXDBMSs, therefore, depends on the chosen compromise between the number of indexed paths and the size of the covering index.

### 3.3.3. *Adaptive Indices*

Another solution to the problem of optimization of evaluating queries with twig patterns with branching nodes which moreover decreases the size of the resulting index is so-called *adaptive indexation*. An adaptive method indexes primarily basic structural relationships and extends them in runtime according to incoming queries.

For example, the *APEX index* (Chung *et al.*, 2002) primarily indexes only edges in a DataGuide of a given set of XML documents. In other words, it indexes only pairs of XML nodes which are connected by an edge corresponding to an edge of the DataGuide. Then, it extends the index according to evaluated queries by concatenating the edges to longer paths. A path is indexed by the APEX index only when the ratio of all user queries containing the path to all user queries exceeds a given threshold.

An advantage is that the system automatically reflects actual user queries. On the other hand, when users start pushing different kind of queries, the index cannot be used. The best way, therefore, is to combine the presented approaches, i.e., to index a set of paths using a fixed F&B index and then also use an adaptive index for other, not indexed paths.

## 3.4. *Comparison of Native XML Databases*

At the end of this section, we compare three selected native XML databases – open source (eXist Native XML Database) and (Oracle Berkeley XML DB) and commercial (webMethods Tamino). Table 3 provides an overview of the XML-related functions that are supported in the three systems.

Let us discuss in more detail some observations resulting from the table. Firstly, many native XML database systems still use a relational database as an option of their internal data storage. Even though the relational database is completely hidden from the applications built on top of the native XML database, there are some important consequences for the applications. For example, transactions, security or full-text search functions can be easily built on top of relational mechanism and directly offered to the applications as functions of the native XML database.

Secondly, regarding XML data querying, all systems support at least the XPath language. Most of them support the full XQuery and XSLT languages as well. This is natural

Table 3
Overview and comparison of key XML features of selected native XML databases

| Feature | *eXist* | *Berkeley XML DB* | *Tamino* |
|---|---|---|---|
| Data store | Native in paged files and B+-trees | Relational database | Native in paged files; relational database |
| Query languages | XQuery, XSTL | XQuery | XQuery |
| Indexing | Element and attribute names, Dewey encoding, DataGuide | Element and attribute names and values, path indices | Indices covering user-defined XPath expressions |
| Query evaluation | Binary structural join | Binary structural join | n/a |
| Other functions | XInclude, XPointer, full-text search, security, transactions (only for crash recovery) | Full-text search, transactions, security | Full-text search, transaction, security |
| Updating | XQuery Update Facility | XQuery Update Facility | Own XQuery extension |

for native XML databases. No combination with the SQL language as we could see in the case of XML enabled databases is supported and it is not even necessary. Regarding XML data updates, usually XQuery Update Facility is supported. The commercial *Tamino* system supports own extension to XQuery.

All systems support indices to speed up XML querying. All of them support basic indices for indexing XML element and XML attribute names and values. The name indices are very important for structural joins as we described in Section 3.2 (they allow for retrieving a sequence of XML nodes with a given name which is the input for structural join algorithms). Both open-source and commercial systems support some more advanced kinds of indices. The *eXist* system indexes the full structure of elements and attributes in the XML documents with a structure based on DataGuide that was described in Section 3.3. The *Berkeley XML DB* system uses a kind of indices which cover all paths going from the root to elements or attributes with a specified name. The *Tamino* server uses indices which cover user defined XPath expressions. Additionally to XML-specific indices, most native XML database systems (and all the three compared in the ta ble) support a kind of XML full-text search, requiring respective indices as well. The table also shows that binary structural join algorithms are supported by the open-source tools.

Thirdly, an important feature of each database system are security and transactional features. While these are sufficiently supported by current relational database systems their support in native XML database systems must be built from scratch when relational features are not used. As can be seen, current versions of the native systems support these features which is a great step towards applying native XML database systems in practice.

In general, current open-source as well as commercial native XML database systems support recent research results in the area of indexing XML data and evaluating path expressions only partly. For example, more advanced covering indices are not supported. Also holistic structural join algorithms are not supported as well. Therefore, current systems could be further improved by adding support of these advanced techniques. This is also demonstrated by the authors of the research results who usually implement their proposed techniques as extensions to current open-source systems, e.g., *Berkeley XML DB*.

In the second part of this paper we will describe two NXDBMSs we have proposed and implemented for educational purposes at our department – *ExDB* and *CellStore*. We will show their key features and compare them mutually as well as with the current state of the art.

## 4. ExDB Native XML DBMS

*ExDB* (Experimental XML DataBase; Loupal, 2006) is a native XML database management system being developed at the Czech Technical University in Prague by students of the Faculty of Electrical Engineering and the Faculty of Information Technology. The primary goal of the project is to prototype a working database environment based upon the XML-$\lambda$ Framework – a functional framework for XML – and thus confirm its suitability for such use case. The framework and related research activities are described in detail in Loupal (2010).

## 4.1. *System Concept*

There are two key ideas that distinguish the *ExDB* system from the other competitors:

(1) the employment of the functional data model for all in-memory data structures related to XML, and

(2) method of evaluating XPath/XQuery queries via their XML-$\lambda$ alternative.

These two properties influence to certain extent the internals of the system. Let us briefly outline these two ideas.

### 4.1.1. *Functional Data Model*

All operations with XML data are performed through a library implementing the XML-$\lambda$ functional data model. It is the most important fact that distinguishes *ExDB* from other systems. Strict use of the data model, its influence even on the structure of low-level paging mechanism is a thorough test of features of the functional approach. The following text points out its key attributes.

The data model utilized inside the *ExDB* for encoding XML data is exclusively based on the functional data model introduced by Pokorný (2002) and later altered in Loupal (2010). These works describe its formal basis in detail. For the purpose of this paper, we select its main properties only and incite the reader to explore the details there.

In the XML-$\lambda$ Framework, an XML document is modeled as a triple $D = \langle \mathbf{E}, \mathbf{T}, \mathbf{S} \rangle$, where $\mathbf{E}$ denotes a set of *abstract elements*, i.e., unique entities corresponding to elements from a particular XML document, $\mathbf{S}$ denotes a set of all strings (either element or attribute content), and finally $\mathbf{T}$ denotes a set of functions that encode relations between abstract elements and strings; informally, we can say that these functions describe the parent-child relationship for all elements within the document. For consistency reasons, attributes are treated in the same way as elements.

### 4.1.2. *Query Processing Approach*

Our approach for query processing within the system is closely related to aforementioned data model. As described in Loupal (2010) we have adopted and employed a query language based on the XML-$\lambda$ data model and simply typed $\lambda$-calculus – the XML-$\lambda$ Query Language. *ExDB* uses this language for evaluating queries written in "conventional" query languages such as XQuery or XPath. Input queries are at first transformed into a respective XML-$\lambda$ form and consequently evaluated in a virtual machine (see Fig. 5). We claim that this unification lets us concentrate on improving the evaluation capability of the functional approach and hence encourage further research activities within this field.

## 4.2. *System Architecture*

Having the functional data model and the associated query language in mind, we have furthermore set two main design goals for the system:

- design and develop a modular and configurable system,

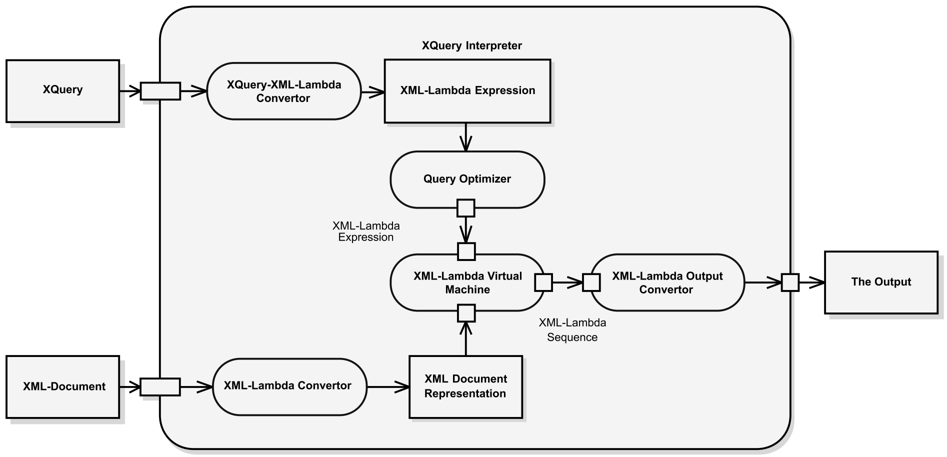


Fig. 5. Evaluation of XPath/XQuery queries in *ExDB*.

- target the system more as an educational project, hence to take more care about system design quality, its stability and code readability instead of chasing for superior performance results.

We claim that such approach ensures long-term maintainability of the code base but, nevertheless, can bring up an efficient and stable database system.

The modular design of *ExDB* is shown in Fig. 6. The modularity allows us to distribute relatively independent assignments to particular developers; new features can be afterwards designed and programmed in parallel and consequent integration is not too complicated. A brief sketch of all modules employed in the current version of the system follows.

#### 4.2.1. *Connection Manager*

manages all client connections (*ExDB* is designed as a client/server system only; we do not plan to support its embedded clone). Nowadays, we offer only a TCP/IP-based proprietary communication protocol (nevertheless, users may select among a command line, Java-based GUI or web-based clients), but there is an existing need for additional alternatives such as web services or a REST-through-servlet API.

#### 4.2.2. *ACL Management*

Each DBMS must obviously support user authentication and corresponding authorization covering all activities being performed. Within *ExDB* we have designed a module that utilizes XACML 2.0 (a general-purpose access control policy language standardized by OASIS); more precisely, SUN's implementation of this standard (OASIS).

#### 4.2.3. *Storage Manager*

aims to persistently save XML data and provide efficient access to it. As the main topic of this paper, we discuss it in detail later in Section 4.3.

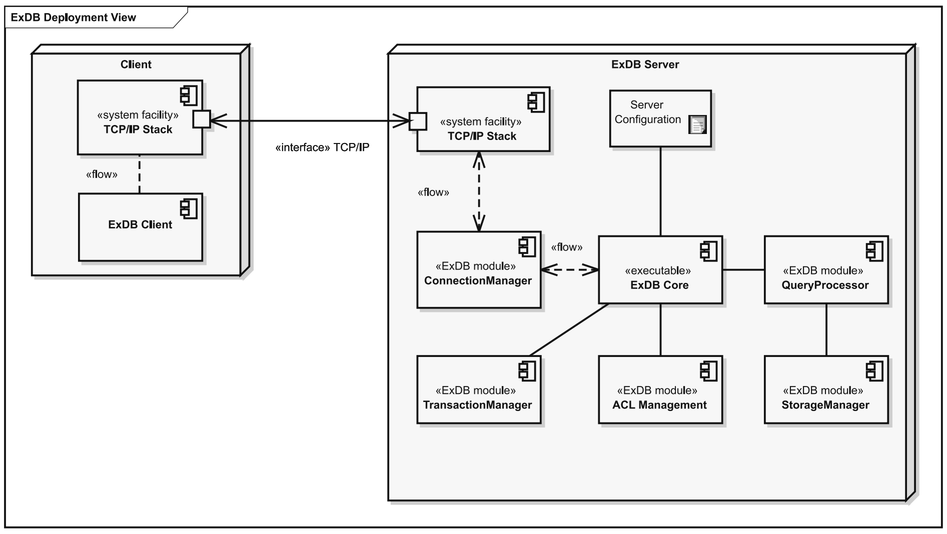


Fig. 6. The *ExDB* architecture.

4.2.4. *Query Processor*
can process XPath, XQuery and XML-$\lambda$ (see Section 4.1.1) queries. It is basically the main "customer" of the Storage module in the system and the ability of these two modules to effectively communicate determines the overall both functional and performance outcomes of the database system. What might be seen as a distinct feature is the fact that all queries are first converted into their XML-$\lambda$ form and only then evaluated in a Virtual Machine (see Section 4.1.2).

4.2.5. *Transaction Manager*
is a planned module used for solving simultaneous access to the stored data. Due to its relative complexity, we have not been able to design a solution in sufficient quality yet.

All these components are controlled by the *Core module* responsible for start-up, initial configuration and message routing among all parts of the system.

## 4.3. *The Storage Subsystem*

*ExDB* generally offers two storage options for XML data: (1) filesystem-based, and (2) native storage. The first alternative is a testing-only option not suitable for production deployment. Data is stored in a filesystem within directories and files that respect a straightforward one-to-one collection/directory and XML document/file mapping. Due to its simplicity this alternative is not worth mentioning in detail here. The latter option, native storage, is obviously more efficient and configurable solution and is hence more important for us. With no doubt, its design is one of the most critical challenges appearing inside the database system.

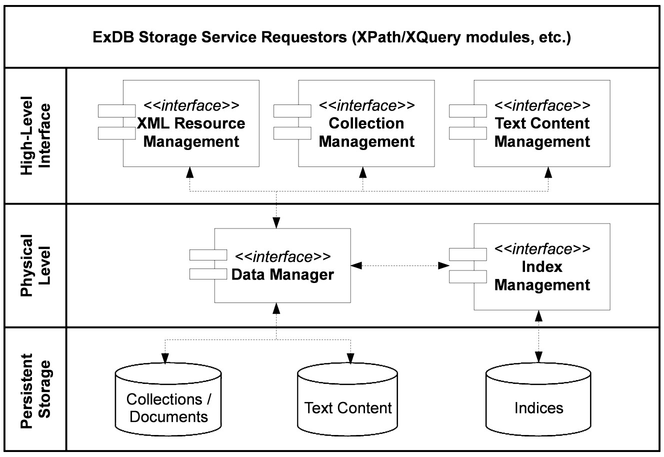

ExDB Storage Service Requestors (XPath/XQuery modules, etc.)
High-Level Interface
<<interface>> XML Resource Management
<<interface>> Collection Management
<<interface>> Text Content Management
Physical Level
<<interface>> Data Manager
<<interface>> Index Management
Persistent Storage
Collections / Documents
Text Content
Indices

Fig. 7. Schema of the persistent storage within the *ExDB*.

Our approach for storing XML data is based on existing methods, primarily on B+ trees. The storage is divided into three logical parts which realize particular operations for *collections* and *documents*, *text content* and *indices*, respectively. Underneath, the high-level interface is backed by a Data Manager performing requested operations on structures designed with respect to the available filesystem.

The low-level persistence layer does not principally differ from existing solutions; it uses fixed-size blocks as fundamental elements of data and provides exchange of these blocks between operational memory and disc drives. Each block contains references to respective parent/sibling fellows and its payload. If possible, the storing algorithm clusters neighboring data entities into one block (if it is not possible, then into multiple but adjoining blocks). This approach is especially suitable for storing the mapping part of the data model instances (denoted as $\mathbf{T}$ or $T$-objects; Loupal, 2010). This structure is in the current implementation represented as a nested hashmap and thus such a tree-like structure is a natural way of storing it persistently.

The efficiency of I/O operations is improved by involving of a memory-cache that realizes (for the present only) the *Least Recently Used* algorithm. The overall schema of the storage is depicted in Fig. 7.

## 4.4. *Issues and Future Work*

*ExDB* is a software project still under active development (lasts more than four years up to now). The relatively slow progress in introduction of new features can be expected with respect to its original goals and its management – the development team (comprising of Bc and MSc students) changes almost each semester and the quality of particular deliverables varies. However, we managed to design and develop multiple working releases of both client and server parts with particular features. Such "long-term" approach to software development cannot be obviously applied for a commercial project but for a research-oriented project it is acceptable. The vision of the "final state" version can be closely defined and the deliverables can be polished by continuous development.

Nowadays, we can describe the following topics as the most important areas for further research and development:

- **Experimental results.** We need to perform both functional and performance benchmarking of the system. So far we have executed only a few particular experiments but still have not performed complex tests such as comparison with other systems or execution of existing standard-compliance suites.
- **Storage improvements.** The storage module can be still improved in terms of implemented indexing methods and related algorithms (see Section 3). We plan to redesign the module (especially the structure of its interface) to gain more efficient access to data.
- **Query optimization.** XPath and XQuery query languages are complex enough to allow us to yield various optimizations. It is a wide topic we plan to address in our future research work – we claim that the model where XPath/XQuery queries are converted into their XML-$\lambda$ equivalents and only then are evaluated offers a very good chance to perform optimizations within the functional machine.

In spite of existing issues, *ExDB* is a usable working prototype of an NXDBMS utilizing the functional approach for XML. The project fulfills its goals but still offers a pool of challenging topics and potential improvements to be solved in the future.

## 5. *CellStore* Native XML DBMS

The main goal of project *CellStore* (Vraný *et al.*, 2008) is to develop an NXDBMS for both educational and research purposes. It is meant rather as an experimental platform than an in-box and ready-to-use database engine. We planed such an engine because the students can easily look inside it, understand and create new components for this engine such as, e.g., a built-in XSLT engine, a query optimizer, an index engine, an event-condition-action (ECA) processing, etc.

According to this goal the development platform had been chosen. Especially:

- it should be easy to change of functionality of subsystems,
- it should be purely object-oriented for development and design,
- it must enable component reusing, test-driven development and trace & log facilities for both debugging and educational purposes.

In the end we selected Smalltalk/X as the development platform.

### 5.1. *Development Strategy*

*CellStore* development is managed incrementally mostly by master thesis of individual participants. There are 8 already successfully finished and 1 ongoing master thesis on the project. Its transaction subsystem (Valenta and Strnad, 2006) is also the topic of a PhD thesis of Pavel Strand, and code-debugging framework *Perseus* (Vraný and Bergel, 2007) was added recently in order to approve concepts of PhD thesis of Jan Vraný. The evolution potential of the project is also an occasional participation on more general projects covered by various national grants.

### 5.2. *History*

The project was started in 2004 with the first implementation of storage subsystem. Implementation of part of XQuery functionality (2007) was the next step. Then implementation of modules for simple-indexing, DML, transactional processing, cache management, web-based approach, remote client, and test setting and evaluation environment followed from 2007 to 2009.

In 2008 a significant change in the system architecture had been done. Jan Vraný included Perseus framework into *CellStore*'s architecture. It brought really illustrative code debugger based on event mechanism. But, on the other hand, it also requires partial redesign of several already done subsystems and slightly slows down *CellStore* efficiency.

### 5.3. CellStore*'s State of The Art*

There are two stages in *CellStore* history – before and after Perseus incorporation. The first – pre-Perseus stage – provided several relatively well integrated modules. *CellStore* worked as an embedded DBMS with partial implementation of XQuery 1.0. It had a database console, a transaction management and a monitoring tool. A comprehensive description of *CellStore* at this stage was published in Pokorný (2007).

In 2008 several new modules and subsystems were under development (e.g., web and line clients, DML module, testing tool etc.). At the same time, Jan Vraný started with Perseus implementation (Vraný and Bergel, 2008). His work implied the necessity of partial redesign of several already developed modules as well as modules just under development. The redesign process was successfully done on new XQuery interpreter, partially on transaction manager, and continues (within master theses) on modules for DML and indexing. Some modules (web and line clients and testing tools) were not affected, others (namely cache management module) were not redesigned yet.

### 5.4. *System Architecture*

*CellStore*'s architecture is depicted in Fig. 8. It can be approached through several interfaces at different levels of services. The lowest layer – low level storage – consists of several cooperating modules. Modules depicted in solid boxes are already implemented, whereas modules in dotted boxes are not ready yet.

#### 5.4.1. *Storage Manager*

Storage Manager is responsible for I/O operations. It operates on physical data layer, it uses both persistent storages – cell space and text space. It also uses its own low-level cache subsystem. Physical structure of both storages is described in detail in Section 5.5.

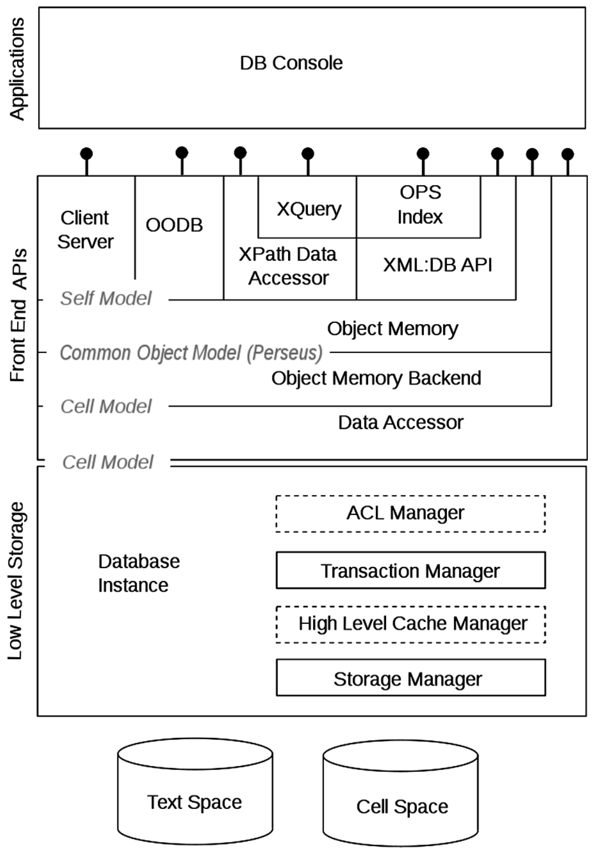


Fig. 8. *CellStore* architecture.

### 5.4.2. *Higher Level Cache Manager*

Higher Level Cache Manager was designed and partially implemented and tested as a master thesis of Karel Příhoda in 2008. It is meant as a “database buffer cache”.

### 5.4.3. *Transaction Manager*

Transaction Manager is a subject of PhD thesis of Pavel Strnad. Some concepts and benchmarks were already published in Strnad and Valenta (2009). It uses non-blocking *taDOM* algorithms developed by Theo Härder and his research group (Haustein and Harder, 2003).

### 5.4.4. *ACL Manager*

ACL Manager is in the planing phase – it was neither designed, nor implemented yet. Thinking seriously about database engine, one cannot omit multiuser access which implies both – transaction management and user/role subsystem with granting abilities.

### 5.4.5. *Front End APIs*

The rest of the system architecture is denoted as “Front End APIs”. Individual APIs are represented by interface marks in the *CellStore*’s architecture (see Fig. 8). They provide various additional services and abstraction layers like XPath or XQuery etc.

## 5.5. *Storage Subsystem*

We developed a new method for storing XML data. The method is based on work of Toman (2004) and partially inspired by solutions used in DBMSs Oracle[1] and Gemstone[2]. Structural and data parts of an XML document are stored separately. Of course, it increases necessary time to store and reconstruct documents. But, on the other hand, it provides a great benefit in disc space management especially in case of document update, query processing and indexing of the stored XML data.

Let us describe the storage model in more detail. Note that the description is based on the first implementation version, because it is more illustrative. There exist improvements in the newer versions of *CellStore*, but they are not so important for a quick view. XML data documents are parsed and placed in two different files during the storing process – *cell file* and *data file*. We illustrate the structure of both the files using the following sample XML document:

```
<?xml version="1.0"?>
<!DOCTYPE simple PUBLIC
  "-//CVUT//Simple Example DTD 1.0//EN" SYSTEM simple.dtd">
<simple>
<!– First comment –>
<?forsomeone process me?>
  <element xmlns="namespace1">
    First text
  <ns2:element xmlns:ns2="namespace2"
      attribute1="value1" ns2:attribute2="value2">
  </ns2:element>
  <empty/>
  </element>
</simple>
```

### 5.5.1. *Cell File Structure*

A *cell file* consists of fixed-length cells. Each *cell* represents a single DOM object (document, element, attribute, character data, etc.) or XML:DB API object (collection or resource). Note that this API is developed by XML:DB Initiative for XML Databases (XML:DB, 2003). Cells are organized into fixed-length blocks.

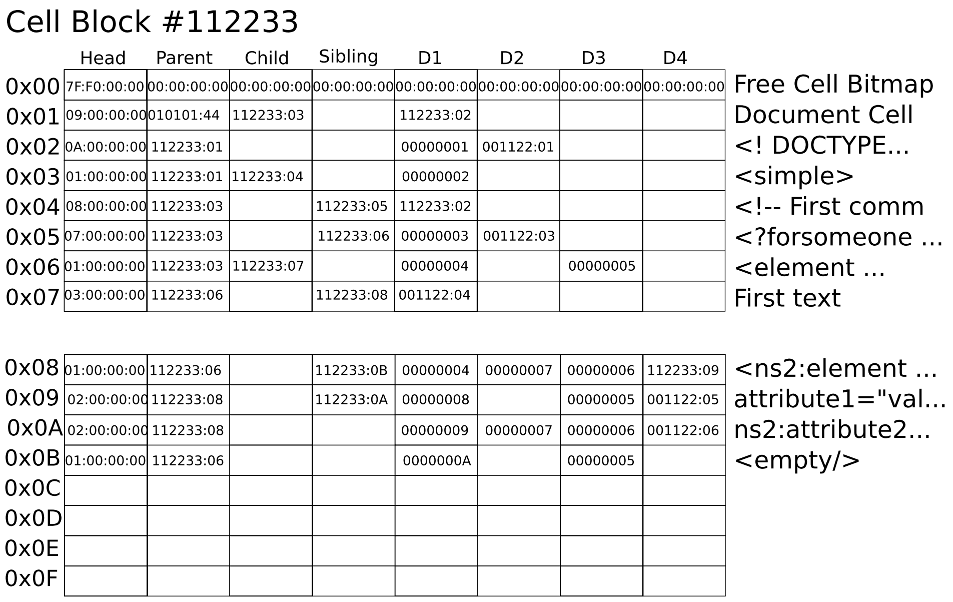
A database *block* is the smallest I/O unit of transfer between disc and low-level storage cache. Only cells from one document can be stored in one block. The set of blocks describing the structure of the whole document is called a *segment*. Each block starts with header with a bitmap describing the density of the block.

[1]`http://www.oracle.com/us/products/database/index.html`.
[2]`http://www.gemstone.com/products/gemstone`.

Table 4

*CellStore* cell structure

| Name | Content | Meaning |
|---|---|---|
| Head | 1 byte | The type of cell. |
| Parent | Cell pointer | Pointer to parent cell. |
| Child | Cell pointer | Pointer to the first child. |
| Sibling | Cell pointer | Pointer to the next cell brother (NIL if there is no one). |
| D1, D2, D3, D4 | Depends on type | Contain either data or pointers (to a text file or a tag file) depending on the type of cell. |



Cell Block #112233

| | Head | Parent | Child | Sibling | D1 | D2 | D3 | D4 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x00 | 7F:F0:00:00 | 00:00:00:00 | 00:00:00:00 | 00:00:00:00 | 00:00:00:00 | 00:00:00:00 | 00:00:00:00 | 00:00:00:00 | Free Cell Bitmap |
| 0x01 | 09:00:00:00 | 010101:44 | 112233:03 | | 112233:02 | | | | Document Cell |
| 0x02 | 0A:00:00:00 | 112233:01 | | | 00000001 | 001122:01 | | | <! DOCTYPE... |
| 0x03 | 01:00:00:00 | 112233:01 | 112233:04 | | 00000002 | | | | <simple> |
| 0x04 | 08:00:00:00 | 112233:03 | | 112233:05 | 112233:02 | | | | <!-- First comm |
| 0x05 | 07:00:00:00 | 112233:03 | | 112233:06 | 00000003 | 001122:03 | | | <?forsomeone ... |
| 0x06 | 01:00:00:00 | 112233:03 | 112233:07 | | 00000004 | | 00000005 | | <element ... |
| 0x07 | 03:00:00:00 | 112233:06 | | 112233:08 | 001122:04 | | | | First text |
| 0x08 | 01:00:00:00 | 112233:06 | | 112233:0B | 00000004 | 00000007 | 00000006 | 112233:09 | <ns2:element ... |
| 0x09 | 02:00:00:00 | 112233:08 | | 112233:0A | 00000008 | | 00000005 | 001122:05 | attribute1="val... |
| 0x0A | 02:00:00:00 | 112233:08 | | | 00000009 | 00000007 | 00000006 | 001122:06 | ns2:attribute2... |
| 0x0B | 01:00:00:00 | 112233:06 | | | 0000000A | | 00000005 | | <empty/> |
| 0x0C | | | | | | | | | |
| 0x0D | | | | | | | | | |
| 0x0E | | | | | | | | | |
| 0x0F | | | | | | | | | |

Fig. 9. *CellStore* cell file structure.

Inside the cell structure internal pointers are used to represent parent-child and sibling relationships of nodes. Each cell consists of eight fields, whereas their meaning can differ with different types of cells. The following cell types are supported in the system: character data, attribute, document, document type, processing instruction, comment, XML Resource, and collection. The general structure of cell is described in Table 4.

See Fig. 9 to grasp the idea how the cell storage looks for the sample XML document mentioned above.

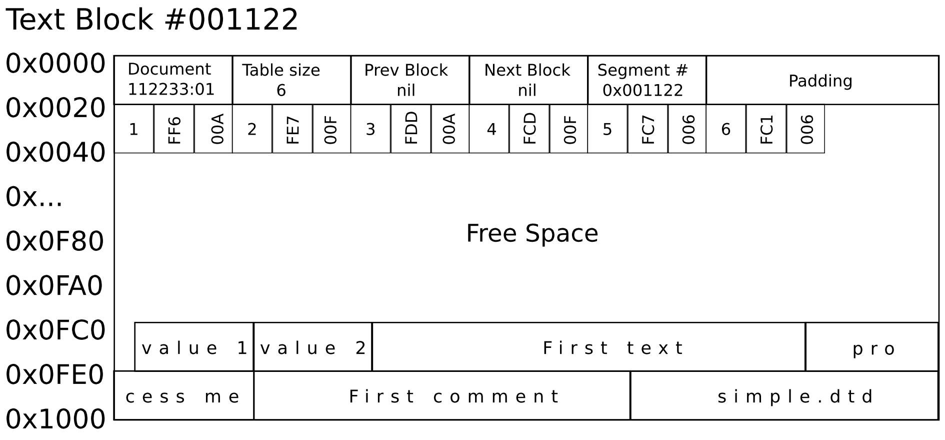


Fig. 10. *CellStore* text file structure.

## 5.5.2. *Text File Structure*

A *text file* contains all text data (i.e., contents of DOM text elements and attributes). The data is organized into *blocks* too, whereas one block belongs just to one document. The set of data blocks belonging to one document is called again a *segment*. A *text pointer* is a pointer to a text file. It consists of a *text block* and a *record*. Each text block contains a translation table which accepts a record number and returns the offset and the length of the data block. This strategy ensures efficiency in case of data changes. The translation table grows from the end of block, while data grow from the beginning. For these purposes the translation table contains the number of actual records. The header of a text block contains also a pointer to the root of its cell node necessary for full-text searching. A sample content of text file structure is shown on Fig. 10.

The low-level subsystem was fully implemented and its stability was tested on INEX data set. INEX (Govert and Kazai, 2002) is the set of articles from IEEE which contains approximately 12,000 individual XML documents (without figures) with total size of about 500 MB.

The newer version of low-level subsystem implementation allows for individual setting of cell, cell-pointer, and block sizes. All these parameters can be used to optimize low-level storage according to specific data needs[3]. Unfortunately, we did not provide enough experiments yet to be able to approve efficiency of such low-level customization.

## 5.5.3. *Storage Discussion*

Our storage strategy has an obvious drawback – necessity to divide XML data into text and structure parts during the storing process and their joining during the document reconstruction. On the other hand, it was experimentally shown, that the space requirement of our storage method is acceptable even in case of frequent changes of parts of stored

[3]Similarly, in Oracle DBMS a `BLOCK_ SIZE`, `PCT_FREE`, and extent-allocation parameters can be used to optimize storage.

data. Moreover, selected obvious improvements like using convenient compress algorithms for text space are evident, although they are not approved by experiments yet.

We believe that our storage method can also provide significant benefits in XQuery processing. Of course, it requires well designed and complex (XQuery) optimizer, which is able to guess and decide when to prefer text and when structure selection criteria. And, separation of structural and text information may also allow us to apply special indexing algorithms. However, all these notions are still at the level of hypothesis and future work.

### 5.6. *Issues and Future Work*

The *CellStore* project is currently running more than 6 years with very alternate development activities. The main idea of educational and research platform is still vital and attractive. Actually, a lot of design and programming work had been done, but, on the other hand, the development strategy described above can be hardly changed under the same circumstances, i.e., combination of theses and grants.

## 6. Experimental Results

In this section we provide a performance comparison of *CellStore*, *ExDB* and *eXist* – one of the most commonly used NXDBMSs – on the basis of efficiency of storing documents. We have measured time needed to store an XML document into the database. But, for the sake of completeness and possibility to compare the systems with those mentioned in Tables 2 and 3, we firstly provide an overview and comparison of main features of all the three benchmarked systems in Table 5.

Table 5
Main characteristics of all systems submitted for the benchmark

| Feature | *ExDB* | *CellStore* | *eXist* |
|---|---|---|---|
| Started in | 2005 | 2004 | 2001 |
| Platform | Java | Smalltalk/X | Java |
| Query languages | XPath, XQuery, XML-$\lambda$ | XPath, XQuery | XPath, XQuery, XSLT |
| Query optimization | – | – | – |
| Data model | XML-$\lambda$ functional model | DOM-based | DOM-based |
| Storage strategy | Filesystem-based, native | Cell file, text file | Native |
| Transaction support | – | taDOM | – |
| Access control | XACML 2.0 | – | XACML 1.0, 1.1 |

### 6.1. *Benchmarking Environment*

The benchmarking environment consists of the XMark data files, all NXDBMSs mentioned above, and a bash script that controls the execution. All parts are subsequently described.

### 6.1.1. *Hardware and Software*

For all parts of the benchmark we have utilized a personal computer with the following system configuration:

- Single-core AMD Athlon 64 3000+ CPU, 2 GB RAM,
- Seagate Barracuda 5400 rpm (SATA),
- Ubuntu 9.10 running 2.6.31-17 (32-bit) linux kernel,
- Sun JRE 1.6.0_17,
- Smalltalk/X 5.4.6.

During the benchmarking, all the daemon processes were shut down to minimize external interference. In order to get realistic results, swap file usage was disabled, so that final results reflect rather the NXDBMS storage algorithm efficiency and not the virtual memory manager paging behavior.

### 6.1.2. *Benchmark Configuration*

6.1.2.1. *XML data.* The `xmlgen` tool, developed at CWI as a part of the XML Benchmark Project (Schmit *et al.*, 2001), was used to generate a set of input XML files. Setting the XMark factor parameter values to 0.01, 0.02, 0.05, 0.1 and 0.2 in successive steps, the test set containing 1.12 MB, 2.27 MB, 5.6 MB, 11.3 MB, 22.8 MB, 56 MB and 112 MB XML files was obtained.

6.1.2.2. *Benchmark Realization.* Within the benchmark we have to deal with two technologies – Java and Smalltalk. Each one requires different setup and behaves in a specific manner.

For Java databases we developed a benchmarking suite (in principle, a simple Java application that acts as an Adapter for all databases) that is encapsulated by various operating system scripts (using fundamental Linux tools, such as `grep` or `awk`).

Both Java and Smalltalk databases output their results in a common shared (textual) format. These files are processed by additional Ruby scripts that prepare input for the GNUPlot drawing utility.

## 6.2. *Results*

The obtained results are shown in Table 6. Figure 11 depicts corresponding chart. The *CellStore* has a linear time complexity of the storing algorithm according to the size of a document. Its storing algorithm is approximately 4-times slower than the *eXist* but the time complexity of the import operation is the same. The *ExDB* implements storing algorithm in another way. Storing of a document is done in two phases. In the first phase the document is parsed and stored into the memory. The document is stored into the disc in the second phase. Unfortunately, as shown in Fig. 12 this solution does not scale well and it has to be redesigned. Moreover, the parsing time complexity seems to be unacceptable.

Table 6

Document storing experimental results

| Dataset | Filename | XMark factor | Size (kB) | Document Storing Time (ms) | | |
|---|---|---|---|---|---|---|
| | | | | *CellStore* | *ExDB* | *eXist* |
| 1 | db001.xml | 0.01 | 1 154 | 3 222 | 2 798 | 651 |
| 2 | db002.xml | 0.02 | 2 291 | 6 472 | 5 423 | 1 084 |
| 5 | db005.xml | 0.05 | 5 735 | 15 542 | 17 992 | 2 770 |
| 10 | db01.xml | 0.10 | 11 596 | 31 032 | 56 614 | 6 995 |
| 20 | db02.xml | 0.20 | 23 364 | 63 040 | 192 334 | 12 925 |
| 50 | db05.xml | 0.50 | 56 647 | 157 603 | 781 239 | 32 896 |
| 100 | db10.xml | 1.00 | 113 787 | 305 510 | 2 356 872 | 93 696 |

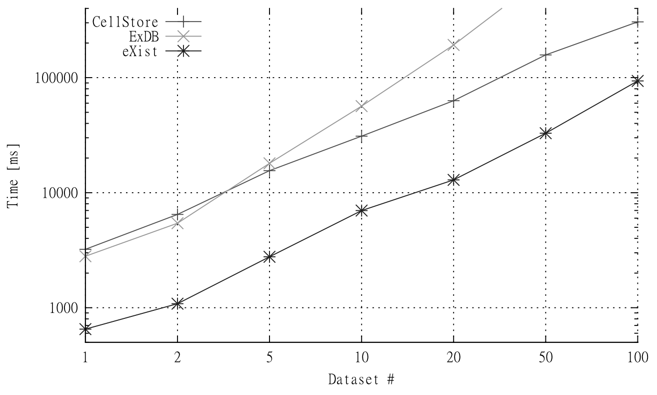


Fig. 11. Document storing experiment results.

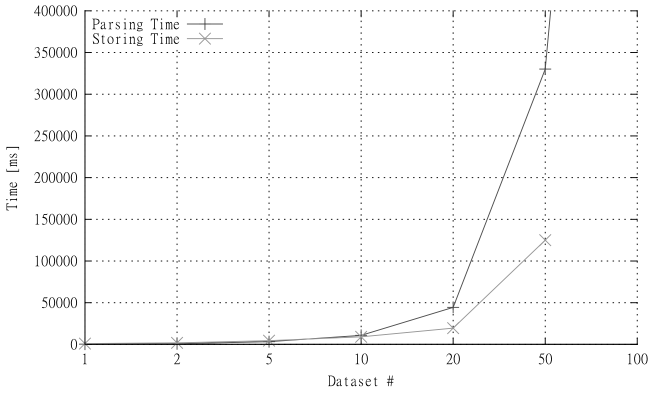


Fig. 12. *ExDB* parsing and storing times.

## 7. Conclusions

The main aim of this paper was to provide an overview of current approaches towards efficient storage and management of XML data. In general, the paper consists of two parts. Firstly, we have provided a general overview of two current key approaches – XML-enabled databases which are based on usage of verified relational database management systems and native XML databases which are designed particularly for the hierarchical tree structures of XML data.

In the second part of our paper we have introduced and compared two native XML databases – *ExDB* (Loupal, 2006) and *CellStore* (Vraný *et al.*, 2008) – that have been proposed, implemented and optimized in our research groups in recent years. Both the systems were planned for the educational purposes, can be freely downloaded and tested and enable anyone to study and analyze the native approaches in detail. The reader should get an idea how to design and develop an XML database.

Our aim was to provide a general description of the state of the art of XML storage strategies and to put into this context our two systems. The reader may easily get acquainted with their features in comparison to the other implementations and decide whether and for what purposes they can be used.

As we have indicated in the text, both the systems have several open issues to be proposed, implemented and tested. The key aspect of an XML database is naturally efficient evaluation of XML queries. Hence, the amount of existing approaches is enormous and we have focused on this aspect primarily as well. However, a reasonable DBMS must support also "side" functionality such as multi-user access, transactions, versioning etc. So, in our future work, we will focus mainly on two parallel aims – query optimization strategies and implementation of full functionality of a classical database management system. On the other hand, our key target still remains the educational aspect of the systems, not the performance results which form the key aim of other related, partly of fully commercial projects.

**Acknowlegments.** This work has been supported by the Czech Science Foundation (GAČR), grants No. 201/09/0990 (M. Nečaský, K. Richta), 201/09/P364 (I. Mlýnková) and by the Czech Technical University in Prague, grants No. SGS10/201/OHK3/2T/13 (P. Strnad), SGS10/226/OHK3/2T/18 (P. Loupal).

## References

Al-Khalifa, S., Jagadish, H.V., Patel, J.M., Wu, Y., Koudas, N., Srivastava, D. (2002). Structural joins: a primitive for efficient xml query pattern matching. In: *Proceedings of ICDE'02*. IEEE Computer Society, Washington, pp. 141–152.

Amer-Yahia, S. (2003). *Storage techniques and mapping schemas for XML*. Technical report TD-5P4L7B, ATT Labs-Research.

Amer-Yahia, S., Du, F., Freire, J. (2004). A comprehensive solution to the XML-to-relational mapping problem. In: *Proceedings of WIDM'04*. ACM Press, New York, NY, pp. 31–38.

Balmin, A., Papakonstantinou, Y. (2005). Storing and querying XML data using denormalized relational databases. *The VLDB Journal*, 14(1), 30–49.

Bača, R., Walder, J., Pawlas, M., Krátký, M. (2010). Benchmarking the compression of XML node streams. In: *Proceedings of BenchmarX'10 International Workshop, DASFAA*. Springer, Berlin, LNCS, Vol. 6193, pp. 179–190.

Berglund, A., Boag, S., Chamberlin, D., Fernandez, M.F., Kay, M., Robie, J., Simeon, J. (2007). *XML Path Language (XPath) 2.0*. W3C. `http://www.w3.org/TR/xpath20/`.

Biron, P.V., Malhotra, A. (2004). *XML Schema Part 2: Datatypes*, 2nd edn. W3C. `http://www.w3.org/TR/xmlschema-2/`.

Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J., Simeon, J. (2007). *XQuery 1.0: An XML Query Language*. W3C. `http://www.w3.org/TR/xquery/`.

Bohannon, P., Freire, J., Roy, P., Simeon, J. (2002). From XML Schema to relations: a cost-based approach to XML storage. In: *Proceedings of ICDE'02*. IEEE Computer Society, Washington, pp. 64–75.

Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F. (2006). *Extensible Markup Language (XML) 1.0*, 4th edn. W3C.

Bruno, N., Koudas, N., Srivastava, D. (2002). Holistic twig joins: optimal XML pattern matching. In: *Proceedings of ACM SIGMOD'02*. ACM Press, New York, pp. 310–321.

Chen, T., Lu, J., Ling, T.W. (2005). On boosting holism in XML twig pattern matching using structural indexing techniques. In: *Proceedings of ACM SIGMOD'05*. ACM Press, New York, pp. 455–466.

Chen, S., Li, H.-G., Tatemura, J., Hsiung, W.-P., Agrawal, D., Candan, K.S. (2006). Twig$^2$stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In: *Proceedings of VLDB'06*. VLDB endowment, pp. 283–294.

Chien, S.-Y., Vagena, Z., Zhang, D., Tsotras, V.J., Zaniolo, C. (2002). Efficient structural joins on indexed XML documents. In: *Proceedings of VLDB'02*. VLDB endowment, pp. 263–274.

Chung, C.-W., Min, J.-K., Shim, K. (2002). APEX: an adaptive path index for XML data. In: *Proceedings of ACM SIGMOD'02*. ACM Press, New York, pp. 121–132.

Clark, J., DeRose, S. (1999). *XML Path Language (XPath) Version 1.0*. W3C. `http://www.w3.org/TR/xpath`.

Cohen, E., Kaplan, H., Milo, T. (2002). Labeling dynamic XML trees. In: *Proceedings of PODS'02*. ACM Press, New York, pp. 271–281.

*DB2 Product Family*. IBM. `http://www-01.ibm.com/software/data/db2/`.

Dietz, P.F. (1982). Maintaining order in a linked list. In: *Proceedings of STOC'82*. ACM Press, New York, pp. 122–127.

*Document Object Model (DOM)*. W3C. `http://www.w3.org/DOM/`.

Du, F., Amer-Yahia, S., Freire, J. (2004). ShreX: managing XML documents in relational databases. In: *Proceedings of VLDB'04*. Morgan Kaufmann Publishers Inc., Toronto, pp. 1297–1300.

*eXist Native XML Database*. `http://exist.sourceforge.net/`.

Florescu, D., Kossmann, D. (1999). Storing and querying XML data using an RDMBS. *IEEE Computer Society Data Eng. Bull.*, 22(3), 27–34.

Govert, N., Kazai, G. (2002). Overview of the initiative for the evaluation of XML retrieval (INEX) (2002). In: *Proceedings of INEX'02*. INEX, Dagstuhl, pp. 1–17.

Haustein, M., Harder, T. (2003). taDOM: A tailored synchronization concept with tunable lock granularity for the DOM API. In: *Proceedings of ADBIS'03*. Springer, Berlin, LNCS, Vol. 2798, pp. 88–102.

ISO/IEC 9075-14:2003 (2003). *Part 14: XML-Related Specifications (SQL/XML)*. Int. Organization for Standardization.

Jiang, H., Wang, W., Lu, H., Yu, J.X. (2003a). Holistic twig joins on indexed XML documents. In: *Proceedings of VLDB'03*. VLDB endowment, pp. 273–284.

Jiang, H., Lu, H., Wang, W., Ooi, B.C. (2003b). XR-tree: indexing XML data for efficient structural joins. In: *Proceedings of ICDE'03*. IEEE Computer Society, Washington, pp. 253–263.

Jiang, Z., Luo, C., Hou, W.-C., Zhu, Q., Che, D. (2007). Efficient processing of XML twig pattern: a novel one-phase holistic solution. In: *Proceedings of DEXA'07*. Springer, Berlin, LNCS, Vol. 4653, pp. 87–97.

Kaushik, R., Bohannon, P., Naughton, J.F., Korth, H.F. (2002). Covering indexes for branching path queries. In: *Poceedings of ACM SIGMOD'02*. ACM Press, New York, pp. 133–144.

Klettke, M., Meyer, H. (2001). XML and object-relational database systems – enhancing structural mappings based on statistics. In: *Selected Papers from WebDB'00*. Springer, London, pp. 151–170.

Kuckelberg, A., Krieger, R. (2003). Efficient structure oriented storage of XML documents using ORDBMS. In: *Proceedings of VLDB'02 Workshop EEXTT and CAiSE'02 Workshop DTWeb*. Springer, London, pp. 131–143.

Li, J., Wang, J. (2008a). Fast matching of twig patterns. In: *Proceedings of DEXA'08*. Springer, Berlin, LNCS, Vol. 5181, pp. 523–536.

Li, J., Wang, J. (2008b). TwigBuffer: avoiding useless intermediate solutions completely in twig joins. In: *Proceedings of DASFAA'08*. Springer, Berlin, LNCS, Vol. 4947, pp. 554–561.

Li, Q., Moon, B. (2001). Indexing and querying XML data for regular path expressions. In: *Proceedings of VLDB'01*. Morgan Kaufmann Publishers, Inc., San Francisco, pp. 361–370.

Loupal, P. (2006). *Experimental DataBase (ExDB) Project Homepage*. `http://exdb.fit.cvut.cz`.

Loupal, P. (2010). *XML-$\lambda$: a functional framework for XML*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

Lu, J., Chen, T., Ling, T.W. (2004). Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In: *Proceedings of CIKM'04*. ACM Press, New York, pp. 533–542.

*Microsoft SQL Server 2008*. Microsoft Corporation. `http://www.microsoft.com/`.

Mlýnková, I. (2007). A journey towards more efficient processing of XML data in (O)RDBMS. In: *Proceedings of CIT'07*. IEEE Computer Society, Los Alamitos, pp. 23–28.

OASIS. *XACML 2.0*. `http://sunxacml.sourceforge.net/`.

O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N. (2004). ORDPATHs: insert-friendly XML node labels. In: *Proceedngs of SIGMOD'04*. ACM Press, New York, pp. 903–908.

*Oracle Berkeley XML DB*. `http://www.oracle.com/technetwork/database/berkeleydb/overview/ index.html`.

*Oracle Database 11g*. Oracle Corporation. `http://www.oracle.com/`.

Pokorný, J. (2002). XML-$\lambda$: an extendible framework for manipulating XML data. In: *Proceedings of BIS 2002*. Poznan University of Economics, Poznan, pp. 160–168.

Pokorný, J., Richta, K., Valenta, M. (2007). CellStore: educational and experimental XML-native DBMS. In: *Proceedings of ISD'07*. Springer, Berlin, pp. 989-1000.

Qin, L., Yu, J.X., Ding, B. (2007). *TwigList*: make twig pattern matching fast. In: *Proceedings of DASFA'07*. Springer, Berlin, LNCS, Vol. 5181, pp. 850–862.

Runapongsa, K., Patel, J.M. (2002). Storing and querying XML data in object-relational DBMSs. In: *Proceedings of EDBT'02*. Springer, London, pp. 266–285.

Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D.J., Naughton, J.F. (1999). Relational databases for querying XML documents: limitations and opportunities. In: *Proceedings of VLDB'99*. Morgan Kaufmann, San Francisco, pp. 302–314.

Schmidt, A. R., Waas F., Kersten, M. L., Florescu, D., Manolescu, I., Carey, M. J., Busse, R. (2001). *The XML Benchmark Project*. `http://xml-benchmark.org/`.

Strnad, P., Valenta, M. (2009). On benchmarking transaction managers. In: *Proceedings of BenchmarX'09 International Workshop, DASFAA*. Springer, Berlin, LNCS, Vol. 5667, pp. 79–92.

Tatarinov, I., Viglas, S. D., Beyer, K., Shanmugasundaram J., Shekita, E. (2002). Storing and querying ordered XML using a relational database system. In: *Proceedings of ACM SIGMOD'02*. ACM Press, New York, pp. 204–215.

Thompson, H.S., Beech, D., Maloney, M., Mendelsohn, N. (2004). *XML Schema Part 1: Structures*, 2nd edn. W3C. `http://www.w3.org/TR/xmlschema-1/`.

Toman, K. (2004). Storing XML data in a native repository. In: *Proceedings of DATESO'04*. CEUR Workshop Proceedings, Vol. 98, pp. 51–62.

Valenta, M., Strnad, P. (2006). Object-oriented implementation of transaction manager in CellStore project. In: *Objekty 2006*. Technická Universita Ostrava, Vysoká Škola Báňská, Ostrava, pp. 273–282.

Vraný, J., Bergel, A. (2008). *Perseus Framework*.
`http://swing.fit.cvut.cz/projects/perseus`.

Vraný, J., Bergel A. (2007). The Debuggable Interpreter Design Pattern. In: *Proceedings of ICSOFT'07*. Springer, Berlin, pp. 1–17.

Vraný, J., Strnad, P., Valenta, M. (2008). *CellStore*. `http://cellstore.felk.cvut.cz/`.

*webMethods Tamino*.
`http://www.softwareag.com/Corporate/products/wm/tamino/default.asp`.

Wu, Y., Patel, J.M., Jagadish, H. (2003). Structural Join Order Selection for XML Query Optimization. In: *Proceedings of ICDE 2003*. IEEE Computer Society, Washington, pp. 443–454.

Xiao-Ling, W., Jin-Feng, L., Yi-Sheng, D. (2003). An adaptable and adjustable mapping from XML data to tables in RDB. In: *VLDB'02 Workshop EEXTT and CAiSE'02 Workshop DTWeb*. Springer, Berlin, LNCS, Vol. 2590, pp. 117–130.

XML:DB (2003). *XML:DB – Application Programming Interface for XML Databases*. The XML:DB Initiative. `http://xmldb-org.sourceforge.net/xapi/`.

Zhang, C., Naughton, J., DeWitt, D., Luo, Q., Lohman, G. (2001). On supporting containment queries in relational database management systems. In: *Proceedings of ACM SIGMOD'01*. ACM Press, New York, pp. 425–436.

Zheng, S., Wen, J., Lu, H. (2003). Cost-driven storage schema selection for XML. In: *Proceedings of DASFAA'03*. IEEE Computer Society, Washington, pp. 337–344.

**P. Loupal** received his PhD degree in computer science in 2010 from the Czech Technical University in Prague, Czech Republic (CTU). He is an assistant professor at the Department of Software Engineering of the Faculty of Information Technology at CTU. His research interests cover particularly the area of native XML database systems, related XML technologies, and formal methods of data processing based on functional approach. Currently, he acts as the lead developer of the ExDB DBMS. He has published more than 20 papers.

**I. Mlýnková** received her PhD degree in computer science in 2007 from the Charles University in Prague, Czech Republic. She is an assistant professor at the Department of Software Engineering of the Charles University and an external member of the Department of Computer Science and Engineering of the Czech Technical University. She has published more than 50 publications, 4 gained the Best Paper Awards. She is a PC member or reviewer of 15 international events and co-organizer of 3 international workshops (X-Schemas@ADBIS, MoViX@DEXA, BenchmarX@DASFAA, all since 2009).

**M. Nečaský** received his PhD degree in computer science in 2008 from the Charles University in Prague, Czech Republic, where he currently works in the Department of Software Engineering as an assistant professor. He is an external member of the Department of Computer Science and Engineering of the Faculty of Electrical Engineering, Czech Technical University in Prague. His research areas involve XML data design, integration and evolution. He is an organizer or PC chair of three international workshops. He has published 15 refereed conference papers (two received Best Paper Award). He has published 3 book chapters and a book.

**K. Richta** received his PhD degree in computer science in 1982 from the Czech Technical University in Prague, Czech Republic. He is an associate professor at the Department of Software Engineering of the Charles University and an external member of the Department of Computer Science and Engineering of the Czech Technical University. He has published more than 100 publications, including 5 books. He is the president of Czech ACM Chapter.

**P. Strnad** is a PhD student at the Department of Computer Science and Engineering of the Faculty of Electrical Engineering of the Czech Technical University in Prague. His research areas involve transaction processing in native XML databases, component benchmarking, XML benchmarking and GPU processing in databases. He has published 5 conference papers.

## XML duomenų saugojimas: ExDB ir CellStore sprendimai kitų esamų sprendimų kontekste

Pavel LOUPAL, Irena MLÝNKOVÁ, Martin NEČASKÝ,
Karel RICHTA, Pavel STRNAD

Straipsnyje nagrinėjama, kaip saugoti XML duomenis, kad po to juos būtų patogu apdoroti. Vienas iš populiarių būdų yra patalpinti juos kokioje nors jau esamoje duomenų bazėje, pavyzdžiui, reliacinėje arba reliacinėje objektinėje bazėje. Straipsnyje nagrinėjami įvairūs XML duomenų vaizdavimo reliacinėse duomenų bazėse būdai. Parodyta, kad, nepaisant plačiai paplitusio reliacinių duomenų bazių valdymo sistemų naudojamos patiems įvairiausiems tikslams, taip pat ir XML duomenims tvarkyti, tokie būdai nėra perspektyvūs, nes originalūs XML duomenys turi ne lentelių, bet medžių pavidalą. Todėl pagrindinis dėmesys straipsnyje skirtas originalioms XML duomenų bazėms. Nagrinėjamos XML duomenų bazių valdymo sistemų realizavimo problemos, aprašomi siūlomų tų problemų sprendimo būdų eksperimentinių tyrimų rezultatai, atlikti panaudojant autorių ir jų bendradarbių suprojektuotas, realizuotas ir optimizuotas eksperimentines XML duomenų bazių valdymo sistemas ExDB ir CellStore.