How to Store XML Data

Technical Report No.: 2010/2
Dept. of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague
November 2010

Pavel Loupal[1], Irena Mlýnková[2], Martin Nečaský[2], Karel Richta[2], Michal Valenta[1]

[1] Department of Software Engineering, Faculty of Information Technology,
Czech Technical University in Prague, Czech Republic
{pavel.loupal,michal.valenta}@fit.cvut.cz
[2] Department of Software Engineering, Faculty of Mathematics and Physics,
Charles University in Prague, Czech Republic
{mlynkova,necasky,richta}@ksi.mff.cuni.cz

How to Store XML Data*

Pavel Loupal[1], Irena Mlýnková[2], Martin Nečaský[2], Karel Richta[2], Michal Valenta[1]

[1] Department of Software Engineering, Faculty of Information Technology,
Czech Technical University in Prague, Czech Republic
{pavel.loupal,michal.valenta}@fit.cvut.cz

[2] Department of Software Engineering, Faculty of Mathematics and Physics,
Charles University in Prague, Czech Republic
{mlynkova,necasky,richta}@ksi.mff.cuni.cz

Abstract. The article deals with the methods how to store XML data. We describe possibilities how to store XML data in relational databases, because relational systems are widely used. But XML data are trees, not tables, so the main attention of this article is oriented to native XML databases. We describe general properties of such sort of databases, and explain possible solutions on the two experimental native XML database management systems – ExDB and CellStore.

1 Introduction

Without any doubt the eXtensible Markup Language (XML) [14] is currently one of the most popular formats for data representation. The wide popularity naturally invoked an enormous endeavor to propose faster and more efficient methods and tools for managing and processing XML data. Soon it was possible to distinguish several different directions based on various storage strategies. The two most popular ones are methods which store and process XML data using an (object-)relational database management system ((O)RDBMSs) – we speak about so-called *XML-enabled databases* – and *native XML approaches* that use special indices, numbering schemas, and/or structures suitable particularly for tree structure of XML data. Expectably, the highest-performance techniques are 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.

Contribution. In this paper we are trying to compare different approaches to the problem of storing XML data. Either we can use existing resources and DBMSs (Data Base Management Systems), or we can create new tools. DBMSs are complex systems, so creating new XML-native systems has meaning only if we bring a new options and a new quality. For this purpose it is necessary to carry out some experiments, to be able

* This work was partially supported by the Czech Science Foundation (GAČR), grants no. 201/09/0990 and 201/09/P364, and also by the Ministry of Education, Youth and Sports under Research Program no. MSM0021620838.

to compare different approaches. There exists two projects of the native XML DBMS - ExDB and CellStore. Both projects are based on similar ideas, but use a different environment - ExDB uses Java, CellStore uses Smalltalk.

Outline. The rest of the text is structured as follows: Section 2 introduces the features and (dis)advantages of XML-enabled databases. Section 3 deals with the basic features of XML-native database management systems. Section 4 contains a description of the ExDB native XML DBMS, and section 5 contains a description of the CellStore DBMS. Finally, conclusions are attached, with possible future research directions.

2 XML-Enabled Databases

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 are 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 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 [29] 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 [21, 10] or XQuery [12], 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's encoding* [23], *interval encoding* [38], *prefix encoding* [22], *ORDPATHS* [44] or *APEX* [19]. And such indices can be also exploited in relational databases to optimize query processing.

2.1 Generic vs. Schema-Driven Methods

Generic mapping methods [25, 35] do not use (possibly) existing XML schema of stored XML documents. They are usually based on one of 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 [4], while the latter reflect its special "relational" structure.

A typical representative of generic mapping is a group of methods called *structure-centered mapping* [35]. 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,...), l is the node label, c is the node content and $n \in \{1, ..., n\}$ 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 access performance. 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 *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 (DF) 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 Figure 1).

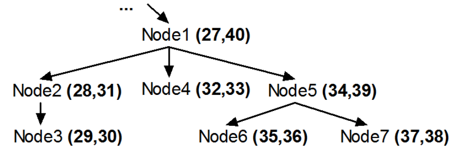

Fig. 1. An example of a generic-tree

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* [49, 48] are based on an existing schema S_1 of stored XML documents, written in DTD [14] or XML Schema [51, 11], 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 meth-

ods 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. 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. 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* [49]. 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 Figure 2).

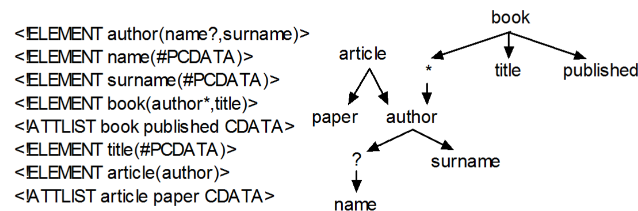

Fig. 2. An example of a generic-tree

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 [34, 13, 58, 60] 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* [13]. 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* [6] 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 [3, 1, 2] support some kind of *user-driven mapping* [8, 7, 42] 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* [24]. 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.

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.

Table 1. Annotation Attributes for ShreX

As we can see from the table, 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* [25] strategy, and mapping an element or an attribute to a CLOB column. Furthermore, it enables 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.

3 Native XML Databases

A *Native XML Database System (NXDBMS)* 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 NXDBMS in comparison to storing XML data into RDBMS are obvious. An XML document may be stored "as it is" without complicated transformation into relational tables and, therefore, may be easily retrieved from the system in its original form. Moreover, 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 some fundamental disadvantages of NXDBMS 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.

In opposite to relational queries, XML queries deal with not only data items of stored XML documents but also structural relationships between their XML nodes. Therefore, 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 will describe the most important representatives of these algorithms in this section. Before this we introduce the notion *numbering schema* which is crucial for structural joins.

3.1 Numbering schemas

Obviously, the amount of XML data might be extensive and 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*. In this section, we briefly describe the most important ones introduced in recent literature.

The simplest numbering schema is *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 base of data values of XML nodes but also on the base of structural relationships between them. The problem is that

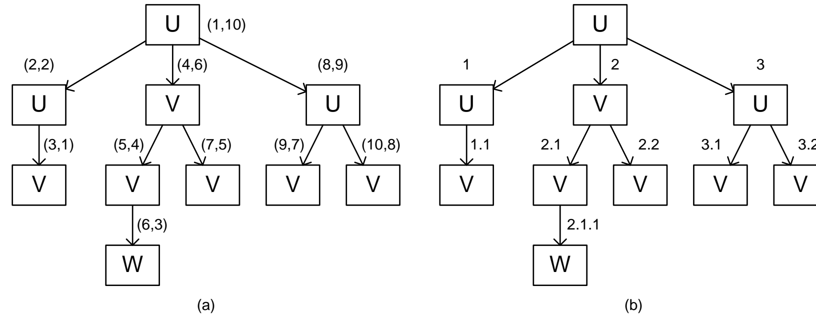

Fig. 3. An example of XML document with nodes numbered by (a) Dietz and (b) Dewey encoding

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*. It assigns each XML node v with a pair $L(v) = (pre(v), post(v))$ where $pre(v)$ returns the order of v in the pre-order traversal of the XML tree while $post(v)$ returns its order in the post-order. Having two nodes u and v, $p(L(u), L(v)) = \texttt{true}$ iff $pre(u) < pre(v)$ and $post(u) > post(v)$. 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 *interval schema*. It assigns each XML node v with an interval $L(v) = (start(v), end(v))$ such that $L(v)$ is contained in the interval $L(u)$ for each ancestor u of v and two intervals of any siblings nodes do not overlap. The intervals may be "blew up" so that there is a space for new incoming nodes without the necessity of renumbering the existing nodes.

Another partial solution to the node insertion problem provides *Dewey encoding*. 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 then in the case of

previous schemas. However, the resulting codes are longer and with a variable length. Solution to this problem may be found in [9].

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, 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, . . ., q_n is a tuple $u_1, \ldots, u_n$ 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 a 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. some of the ones introduced in the previous section.

Occurrences of a twig pattern in an XML documents 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 in A 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*.

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. In this part, we provide an overview of methods based on this idea. They are, in general, 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 the article [59]. It showed that an algorithm specially designed for twig query matching can significantly outperform classical relational join algorithms. In [5], Al-Khalifa et al. 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. The 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 read twice. Therefore, Al-Khalifa et al. 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 will 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 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 was using a B+-tree [18]. More optimal solutions are XR-tree [30] and XB-tree [15]. These indexing techniques may be used also for holistic structural join algorithms we discuss in the next part of this section.

Holistic Structural Join Algorithms All binary structural join algorithms can solve only a binary relationship which is usually 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 the selection of the appropriate order of the joins [57]. However, such a solution needs expensive statistics about the XML document.

This problem is partly solved by another family of structural join algorithms called *holistic structural join algorithms*. In general, we can identify the following phases in each holistic structural join algorithm. Note that the algorithms in this family still do not evaluate a twig pattern as a whole. There is the decomposition of the twig pattern

to root-to-leaf paths which can produce intermediate occurrences that do not contribute to the final solution. However, 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 [15], Bruno et al. 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 some 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, e.g. TSGeneric [31], TWIG-STACK-LIST [41], or TWIG-BUFFER [37]. These extensions provide various techniques which allow for evaluation of some twig patterns which contain PC edges on specific positions. However, they still cannot evaluate a general twig pattern with PC edges on arbitrary position optimally.

One-Way Structural Join Algorithms In [16], 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 algorithm TWIG-LIST [47] 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 has also appeared further optimizations in [32] or [36] which are based on combining one-way structural join algorithms with 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 NXDBMS. As we have already showed, each structural join

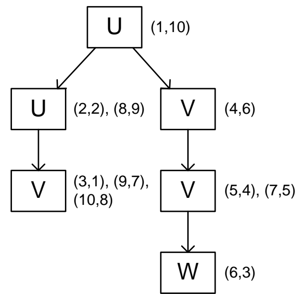

Fig. 4. An example of DataGuide

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 to retrieve them in a form of the input streams. In this section, we will discuss an index structure called *DataGuide* which is designed for this. We will also discuss some alternative indexing techniques which help in particular situations when evaluating XML queries.

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 from Fig. 3 is depicted in Fig. 4.

DataGuide indexes for each of its nodes 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 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, 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 [17], 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 [9].

Covering Indexes *Covering index* is an index which allows for evaluating queries of a particular type without accessing source data on the disc. This kind of indexes may be also found in RDBMS but there are also equivalents in NXDBMS. 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* [33]. 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 then original data. On the other hand, the authors show in [33] 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 and index them by an F&B index. This is analogical to indexing tables in RDBMS. Here, it is 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 NXDBMS therefore depends on the chosen compromise between the number of indexed paths and the size of the covering index.

Adaptive Indexes 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 some basic structural relationships and extends them in runtime according to incoming queries.

For example, the APEX index [20] primarily indexes only edges in a DataGuide of a given set of XML documents. In other words, it indexes only which pairs of XML nodes are connected by a 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 use 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.

4 ExDB Native XML DBMS

ExDB (Experimental XML DataBase) [39] is a native XML database management system being developed at the Czech Technical University in Prague by students of Faculty of Electrical Engineering and Faculty of Information Technology. The primary goal of the project is to prototype a working system based upon the XML-λ 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's Ph.D. thesis [40].

4.1 Concept and Architecture of the System

The main design goals have been set as follows: (1) design and develop a modular and configurable system, (2) target the system more as a educational project, hence take care more about system quality, 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 is shown in Figure 5. That sort of modularity allows us to distribute relatively independent assignments to particular developers; new features can be afterwards designed and programmed in parallel and ongoing integration is not complicated.

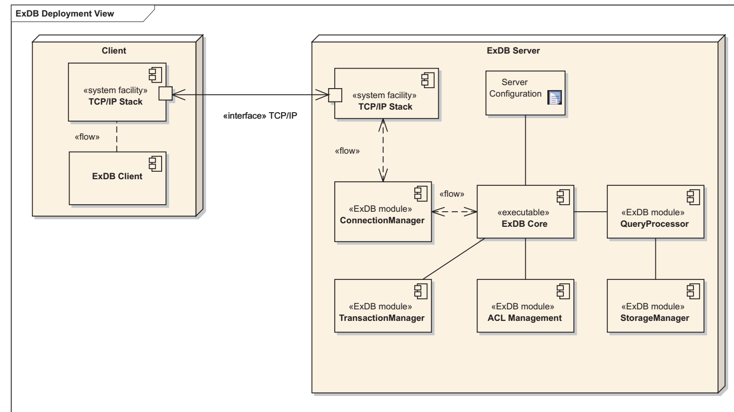

Fig. 5. The ExDB Architecture

Let us provide a brief sketch of all modules employed in current version of the system.

Connection Manager manages all client connections (Recall that 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 (users may select among a command-line, Java-based GUI or a web-based clients) but there is an existing need for additional alternatives such as web services or REST-through-servlet API.

ACL Managment 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 (an OASIS standard dealing with this field); more exactly, a SUN's implementation of this standard [43].

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

Query Processor can process XPath, XQuery and XML-λ 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-λ's form and only then evaluated in a Virtual Machine (see Section 4.4 for more).

Transaction Manager is a planned module used for solving simultaneous access to 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 startup, initial configuration and message routing among all parts of the system.

4.2 System Internals

All operations with XML data are performed through a library implementing the XML-λ functional data model. It is the most important fact that distinguishes ExDB from other systems. Strict use of the data model, its influence even into the structure of low-level paging mechanism is a thorough test of fuctional approach's features. The following text deals with the key topics.

Functional Data Model The data model utilized inside the ExDB for encoding XML data is exclusively based on the functional data model introduced by Pokorný [46] and later altered in [40]. These works describe its formal base in deep detail. For purpose of this paper, we select its main properties only and incite the reader to explore the detailes there.

In the XML-λ Framework, an XML document is modelled 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 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.

4.3 The Storage Subsystem

ExDB generally offers two storage options for XML data: (1) filesystem-based, and (2) native storage. The first choice is a (trivial) testing-only alternative not suitable for production deployment. Data is stored in available filesystem in directories and files that are named by 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 accordingly more important for us. With no doubt, its design is one of the most critical issues of the database system.

Native Storage The storage is divided into three logical parts which realize particular operations for collections / documents, indices and text content, respectively. Underneath, that high-level interface is backed by a Data Manager performing requested operations on structures designed with respect to available filesystem. This 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 disk drives. The effectivity of this operations is ensured by involving some caching mechanisms and by clustering related data into one block (if possible, of course; if not, then into multiple but adjoining blocks).

The overall schema of the storage is depicted in Figure 6.

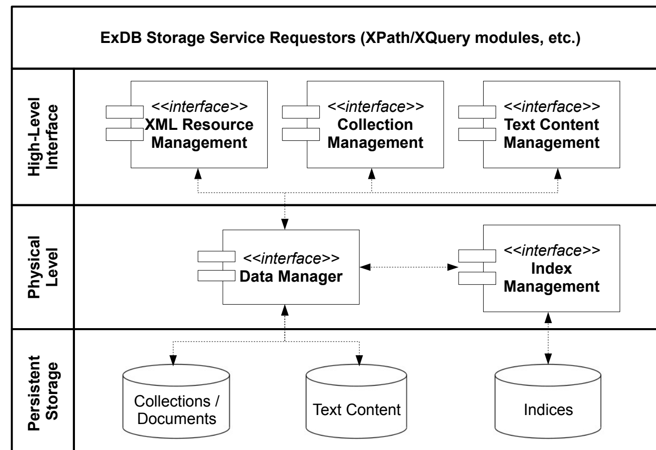

Fig. 6. Schema of the persistent storage within the ExDB

4.4 Query Processing

Let us briefly describe the approach for query processing within the system. As described in [40] there exists a query language based on the XML-λ data model and simply typed λ-calculus – XML-λ Query Language. ExDB uses this language for evaluating queries written in "conventional" query languages such as XQuery or XPath. Input queries are first transformed into a respective XML-λ form and consequently evaluated in a virtual machine (see Figure 7). We claim that this unification lets us concentrate on improving the evaluation capability of the functional approach and hence encourage futher research activities within this field.

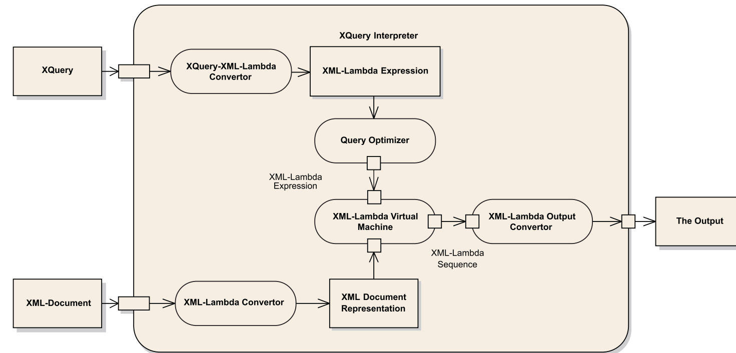

Fig. 7. Evaluation of XPath/XQuery Queries in ExDB

4.5 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 project's original goals and its management – the development team (comprising of Bc. and MSc. students) changes almost each semester and quality of the 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 research-oriented project 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 some existing standard-compliance suites.
- **Storage improvements.** The storage module can be still improved in terms of implemented indexing methods and related algorithms. We plan to redesign the module (especialy 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-λ equivalents and only then are evaluated offers a very good chance to perform optimizations within the functional machine.

In spite of existing issues, we claim that ExDB is a usable working prototype of a native XML DBMS utilizing the functional approach for XML (entitled XML-λ as described in [40]). The project fulfils its goals but still offers a pool of challenging topics to be solved in the future.

5 CellStore Native XML DBMS

We are going to present CellStore project [55] in this section. We'll concentrate on its storage subsystem to be consistent with the main topic of the report. But it is necessary shortly introduce the context of the CellStore project itself. This context information is done in the first subsection, while the main contribution – CellStore storage subsystem is the subject of subsection 5.3.

5.1 What is CellStore

The main goal of the CellStore project is to develop XML-native database engine for both educational and research purposes. It is meant rather as experimental platform then an in-box and ready-to-use database engine.

We planed such an engine because our students can easily look inside it, understand and create new components for this engine, e.g. in-built 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 subsystems functionality, 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. At the end we selected Smalltalk/X as the development platform.

Development Strategy Several grants were applied in order to establish wider user and development community. The project was reported as interesting and well-planed, but, unfortunately, non of applied grants was accepted. Hence 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 [53] is also the topic of a PhD thesis of Pavel Strand, and code-debugging framework Perseus [56] was added recently in order to approve concepts of PhD thesis of Jan Vraný. The evolution potential of the project is also occasional participation on more general projects covered by grants.

History Project was started at 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.

At 2008 a significant change in the system architecture had been done. Jan Vrany 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 slow-down CellStore efficiency.

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 embedded DBMS with partial implementation of XQuery 1.0. It had a database console and a Transaction management and monitoring tool. A comprehensive description of CellStore at this stage was published in [45].

In 2008 several new modules or subsystems were under development (web and line clients, DML module, testing tool). At the same time Jan Vrany started with Perseus implementation [54]. 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 is ongoing (master thesis) on modules for DML and indexing. Some modules (web and line clients and testing tools were not affected), and the others (namely cache management module) were not redesigned yet.

5.2 System Architecture

CellStore's architecture is depicted on figure 5.2. It can be approached through several interfaces on a different level of services. The lowest layer – low level storage – consist on several cooperating modules. Modules depicted in solid boxes are already implemented while modules in dotted boxes are not ready yet.

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 subsection 5.3 below.

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

Transaction Manager is subject of PhD work of Pavel Strnad. Some concepts and benchmarks were already published [50]. It uses non-blocking taDOM algorithms developed by Theo Härder and his research group [27].

ACL Manager is just a plan - it was not yet designed nor implemented. Thinking seriously about database engine, one cannot omit multiuser access which implies boths – transaction management and user/role subsystem with granting abilities.

Front End APIs The rest of the system architecture is marked as "Front End APIs". Individual APIs are represented by interface marks in the CellStore's architecture 5.2. They provide various additional services and abstraction layers like XPath or XQuery etc.

5.3 Storage Subsystem

We developed a new method for storing XML data. The method is based on work of Toman [52] and partially inspired by solutions used in DBMSs of Oracle and Gemstone. Structural and data parts of 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 disk space management especially in the case of documents update and also in query processing and indexing stored XML data.

Let us describe the storage model more in detail. 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 this quick view. XML data documents are parsed and placed in two different files during the storing process – the cell file and the data file. The structure of each of them is described in an individual subsection. We will illustrate the structure of files by example of the following 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>
```

Cell-File Structure This file consists of fixed-length cells. Each cell represents one DOM object (document, element, attribute, character data, etc.) or XML:DB API object (collection or resource). Remind that this API is developed by XML:DB Initiative for XML Databases [28]. Cells are organized into fixed-length block.

Database block is the smallest I/O unit of transfer between disk and low-level storage cache. Only the cells from one document can be stored in one block. The set of blocks describing the structure of the whole document is called segment. Each block starts with header with a bitmap describing the density of the block. This storing strategy is effective also in the case of repeated changes of stored documents.

Inside the cell structure internal pointers are used to represent parent-child and sibling relationships of nodes. Each cell consists of eight fields. The meaning of some fields can differ with different types of cells. The following cell types are in the system: character data, attribute, document, document type, processing instruction, comment, XML Resource, and collection. The general structure of cell is described in the following Table 2.
See figure 5.3 to grasp the idea how the cell storage looks for the sample XML document mentioned above.

Text File Structure This file contains all text data (i.e. contents of DOM's text elements and attributes). The data is organized into blocks too. One block belongs just

Name	Content	Reason
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 (possibly to text file or tag file) depending to the type of cell.

Table 2. CellStore cell structure

to one document. The set of data blocks belonging to one document is called again a segment. Text pointer is a pointer to text file. It consists from the text block# and the record#. Each text block contains a translation table which accepts the record number and returns the offset and the length of the data block. This is effective for storing data changes. The translation table grows from the end of block, while data grows from the beginning. The translation table contains the number of actual records for these purposes. The header of a text block contains also the pointer to the root of its cell node. It is useful for fulltext searching - for the case we need documents containing some patterns. The example content of text file structure is shown on fig. 5.3.

Low-level subsystem was fully implemented. Its stability had been tested on INEX data set. INEX is the set of articles from IEEE; see [26] for an overview. It contains approximately 12 000 individual XML documents (without figures), total size of the set is about 500MB.

The newer version of low-level subsystem implementation allows 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.

Storage discussion Our storage strategy has obvious drawback – necessity to divide XML data into text and structure parts during the storing process and their repeat 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 the case of frequent changes of parts of stored data. Moreover, some obvious improvements like using convenient compress algorithms on text space are evident, although they are not approved by experiments yet.

We believe that our storage method can also provide a 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.

Separation structural and text information may also allow to apply special indexing algorithms. Unfortunately, all this notions are still on the level of hypothesis.

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

5.4 CellStore's Future

CellStore project is already running more then 6 years with a 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, on the other hand the development strategy described above can be hardly changed under the same circumstances.

6 Conclusions

In this report, attention is paid to methods how to store XML data. Initially, we try to employ commonly used relational systems. Various methods are presented, how it is possible to store XML data in relational systems. Then we try to formulate principles of so called native XML DBMS and we presented methods how to store XML data in native systems. Finally, the most original artifacts are ExDB and CellStore native XML database management systems, that we realize as prototypes of native systems, and we have presented main principles of these two proposals.

Most important for any kind of DBMS is querying and its effectivity. The standard of the query language for XML data is at the present time the XQuery language. The future research can be oriented to the way how to compare all possible representations presented in this report in the effectivity of evaluation of queries written in the XQuery language.

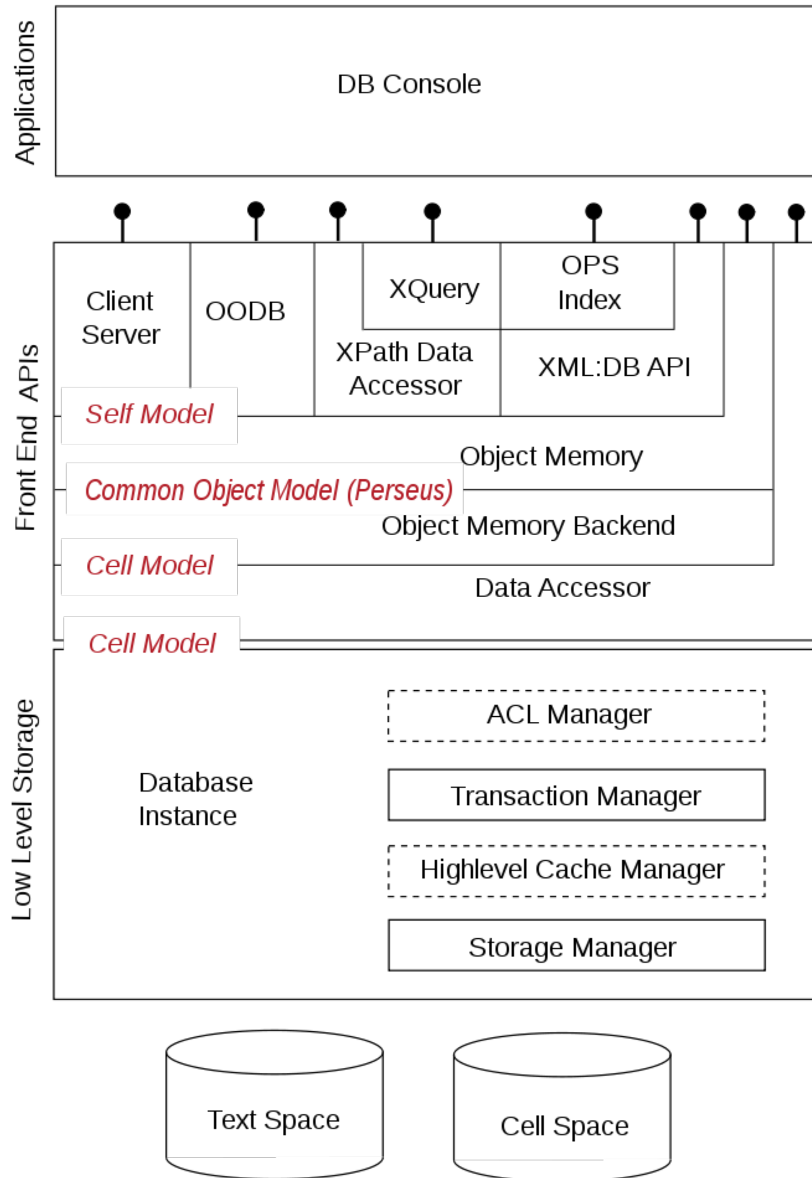

Fig. 8. CellStore Architecture

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	001122:02				<!-- 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 cellfile structure

Text Block # 001122

Offset						
0x0000	Document 112233:01	Table size 6	Prev Block nil	Next Block nil	Segment # 0x001122	Padding
0x0020	1 FF6 00A	2 FE7 00F	3 FDD 00A	4 FCD 00F	5 FC7 006	6 FC1 006
0x0040						
0x...						
0x0F80	Free Space					
0x0FA0						
0x0FC0	value2	value1	First text	pro		
0x0FE0	cess me	First comment	simple.dtd			
0x1000						

Fig. 10. CellStore textfile structure

Bibliography

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

[2] *Microsoft SQL Server 2008*. Microsoft Corporation. http://www.microsoft.com/sqlserver/2008/.

[3] *Oracle Database 11g*. Oracle Corporation. http://www.oracle.com/technology/products/database/oracle11g/.

[4] *Document Object Model (DOM)*. W3C, 2005.

[5] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 141–152, 2002.

[6] S. Amer-Yahia. *Storage Techniques and Mapping Schemas for XML*. Technical Report TD-5P4L7B, AT&T Labs-Research, 2003.

[7] S. Amer-Yahia, F. Du, and J. Freire. A Comprehensive Solution to the XML-to-Relational Mapping Problem. In *WIDM'04*, pages 31–38, New York, NY, USA, 2004. ACM.

[8] A. Balmin and Y. Papakonstantinou. Storing and Querying XML Data Using Denormalized Relational Databases. *The VLDB Journal*, 14(1):30–49, 2005.

[9] Radim Bača, Jiří Walder, Martin Pawlas, and Michal Krátký. Benchmarking the Compression of XML Node Streams. In *Proceedings of the BenchmarX 2010 International Workshop, DASFAA*. Springer-Verlag, 2010.

[10] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernï£¡ndez, M. Kay, J. Robie, and J. Simeon. *XML Path Language (XPath) 2.0*. W3C, January 2007. http://www.w3.org/TR/xpath20/.

[11] P. V. Biron and A. Malhotra. *XML Schema Part 2: Datatypes (Second Edition)*. W3C, 2004. http://www.w3.org/TR/xmlschema-2/.

[12] S. Boag, D. Chamberlin, M. F. Fernï£¡ndez, D. Florescu, J. Robie, and J. Simeon. *XQuery 1.0: An XML Query Language*. W3C, January 2007. http://www.w3.org/TR/xquery/.

[13] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML Schema to Relations: A Cost-based Approach to XML Storage. In *ICDE'02*, pages 64–75, Washington, DC, USA, 2002. IEEE.

[14] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. W3C, 2006.

[15] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 310–321, 2002.

[16] Songting Chen, Hua-Gang Li, Jun'ichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig^{2}Stack: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 283–294. VLDB endowment, 2006.

[17] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On Boosting Holism in XML Twig Pattern Matching using Structural Indexing Techniques. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 455–466, 2005.
[18] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB)*, pages 263–274, 2002.
[19] C.-W. Chung, J.-K. Min, and K. Shim. APEX: an Adaptive Path Index for XML Data. In *SIGMOD'02*, pages 121–132, New York, NY, USA, 2002. ACM.
[20] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. Apex: an adaptive path index for xml data. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 121–132, New York, NY, USA, 2002. ACM.
[21] J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0*. W3C, November 1999. http://www.w3.org/TR/xpath.
[22] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In *PODS'02*, pages 271–281, New York, NY, USA, 2002. ACM.
[23] P. F. Dietz. Maintaining Order in a Linked List. In *STOC'82*, pages 122–127, New York, NY, USA, 1982. ACM.
[24] F. Du, S. Amer-Yahia, and J. Freire. ShreX: Managing XML Documents in Relational Databases. In *VLDB'04*, pages 1297–1300, Toronto, ON, Canada, 2004. Morgan Kaufmann Publishers Inc.
[25] D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.
[26] N. Govert and G. Kazai. Overview of the initiative for the evaluation of xml retrieval (inex) 2002. In *Proc. of the first Workshop of the INitiative for the Evaluation of XML Retrieval (INEX)*, pages 1–17, 2002.
[27] M. Haustein and T. Harder. tadom: A tailored synchronization concept with tunable lock granularity for the dom api. In *Proc. ADBIS'03*, pages 88–102. Springer-Verlag LNCS 2798, 2003.
[28] XML:DB Initiative. XML:DB, 2003.
[29] ISO/IEC 9075-14:2003. *Part 14: XML-Related Specifications (SQL/XML)*. Int. Organization for Standardization, 2006.
[30] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 253–263, 2003.
[31] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic Twig Joins on Indexed XML Documents. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB)*, pages 273–284, 2003.
[32] Zhewei Jiang, Cheng Luo, Wen-Chi Hou, Qiang Zhu, and Dunren Che. Efficient processing of xml twig pattern: A novel one-phase holistic solution. In *Database and Expert Systems Applications (DEXA)*, pages 87–97, 2007.
[33] Raghav Kaushik, Philip Bohannon, Jeffrey F Naughton, and Henry F Korth. Covering indexes for branching path queries. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 133–144, New York, NY, USA, 2002. ACM.

[34] M. Klettke and H. Meyer. XML and Object-Relational Database Systems – Enhancing Structural Mappings Based on Statistics. In *Selected papers from the 3rd Int. Workshop WebDB'00 on The World Wide Web and Databases*, pages 151–170, London, UK, 2001. Springer.

[35] A. Kuckelberg and R. Krieger. Efficient Structure Oriented Storage of XML Documents Using ORDBMS. In *VLDB'02 Workshop EEXTT and CAiSE'02 Workshop DTWeb*, pages 131–143, London, UK, 2003. Springer.

[36] Jiang Li and Junhu Wang. Fast matching of twig patterns. In *Database and Expert Systems Applications (DEXA)*, pages 523–536, 2008.

[37] Jiang Li and Junhu Wang. Twigbuffer: Avoiding useless intermediate solutions completely in twig joins. In *Database Systems for Advanced Applications, 10th International Conference (DASFAA)*, pages 554–561, 2008.

[38] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *VLDB'01*, pages 361–370, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers, Inc.

[39] Pavel Loupal. Experimental DataBase (ExDB) Project Homepage. http://exdb.fit.cvut.cz.

[40] Pavel Loupal. *XML-λ : A functional framework for XML*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, February 2010.

[41] Jiaheng Lu, Ting Chen, and Tok Wang Ling. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In *Proceedings of the 13th ACM international conference on Information and knowledge management (CIKM)*, pages 533–542, 2004.

[42] I. Mlynkova. A Journey towards More Efficient Processing of XML Data in (O)RDBMS. In *CIT'07*, pages 23–28, Los Alamitos, CA, USA, 2007. IEEE.

[43] OASIS. Xacml 2.0. http://sunxacml.sourceforge.net/.

[44] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD'04*, pages 903–908, New York, NY, USA, 2004. ACM.

[45] J. Pokorný, K. Richta, and M. Valenta. Cellstore: Educational and experimental xml-native dbms. In *Information Systems Development 2007*, pages 1–11, Galway, 2007. National University of Ireland.

[46] Jaroslav Pokorný. XML-λ: an extendible framework for manipulating XML data. In *Proceedings of BIS 2002*, pages 160–168, Poznan, 2002.

[47] Lu Qin, Jeffrey Xu Yu, and Bolin Ding. *TwigList* : Make twig pattern matching fast. In *Database Systems for Advanced Applications, 10th International Conference (DASFAA)*, pages 850–862, 2007.

[48] K. Runapongsa and J. M. Patel. Storing and Querying XML Data in Object-Relational DBMSs. In *EDBT'02*, pages 266–285, London, UK, 2002. Springer.

[49] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB'99*, pages 302–314, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[50] P. Strnad and M. Valenta. On benchmarking transaction managers. In *Database Systems for Advanced Applications DASFAA 2009 International Workshops:*

BenchmarX, MCIS, WDPP, PPDA, MBC, PhD, Brisbane, Australia, April 20 - 23, 2009, pages 79–92, Berlin, 2009. Springer.

[51] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures (Second Edition)*. W3C, 2004. http://www.w3.org/TR/xmlschema-1/.

[52] K. Toman. Storing xml data in a native repository. In *Proc. of DATESO 2004*, pages 51 – 62. CEUR Workshop Proceedings, Vol. 98, 2004.

[53] M. Valenta and P. Strnad. Object-oriented implementation of transaction manager in cellstore project. In *Objekty 2006*, pages 273–282, Ostrava, 2006. Technická universita Ostrava - Vysoká škola báňská.

[54] J. Vraný and Bergel A. Perseus framework, 2008.

[55] J. Vraný, P. Strnad, and M. Valenta. Cellstore, 2008.

[56] Vraný, J. and Bergel A. The debuggable interpreter design pattern. In *Proceedings of the International Conference on Software and Data Technologies (ICSOFT 2007)*, pages 1–17, 2007.

[57] Y. Wu, J. M. Patel, and H. Jagadish. Structural Join Order Selection for XML Query Optimization. In *Proceedings of ICDE 2003*, pages 443 – 454. IEEE CS, 2003.

[58] W. Xiao-ling, L. Jin-feng, and D. Yi-sheng. An Adaptable and Adjustable Mapping from XML Data to Tables in RDB. In *VLDB'02 Workshop EEXTT and CAiSE'02 Workshop DTWeb*, pages 117–130, London, UK, 2003. Springer.

[59] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 425–436. ACM Press, 2001.

[60] S. Zheng, J. Wen, and H. Lu. Cost-Driven Storage Schema Selection for XML. In *DASFAA'03*, pages 337–344, Kyoto, Japan, 2003. IEEE.