The Simplest XML Storage Manager Ever*

Avinash Vyas
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974
vyas@lucent.com

Mary Fernández
AT&T Labs - Research
180 Park Ave.
Florham Park, NJ 07932
mff@research.att.com

Jérôme Siméon
IBM Watson Research Center
19 Skyline Drive,
Hawthorne, NY 10532
simeon@us.ibm.com

ABSTRACT

After more than four years of incubation within the W3C, XQuery 1.0 is close to completion. Even before its official release, development of numerous XQuery implementations is underway. However, those implementations have either focused on completeness or performance at the expense of the other. In this paper, we report on our experience building Jungle, a secondary storage manager for Galax. We designed Jungle to be the "simplest XML storage manager ever" that supports the complete XQuery 1.0 language, and we show it can scale to documents up to several hundred megabytes. Interestingly, the process of developing Jungle lead us to revisit several key features of currently proposed XML indexes. As a result, we propose alternative indices that provide more efficient support for the `child` axis and for XML serialization, two critical operations on the XQuery data model. Jungle is fully operational and now being deployed in production applications.

1. INTRODUCTION

After more than four years of incubation within the W3C, XQuery is emerging as the de facto query language for XML. Its success and impact are due, in part, to XQuery's unwavering commitment to being a "good XML citizen". In particular, XQuery supports or is compatible with many related and pre-existing XML standards. XQuery's data model handles namespaces and multiple character sets, preserves the inherent order in XML documents, and supports all six kinds of XML nodes. XQuery 1.0 includes XPath 2.0 as a proper sub-language, making it compatible with other XML-aware languages that depend on XPath 2.0, such as XSLT 2.0 or XPointer. Lastly, XQuery's type system is based on XML Schema, making it possible to query and produce typed XML documents. Being a good XML citizen, however, has a cost and building a complete implementation for XQuery is a significant undertaking. Implementors in both industry and research are adapting decades of database and programming-language results to their XQuery implementations and are crafting new techniques for implementing XQuery efficiently. Still, there is currently no implementation that is both complete and provide reasonable performances for large scale applications. The objective of this paper is to describe Jungle, an XML storage manager that is integrated with Galax to create a complete-yet-performant XQuery 1.0 implementation.

*(Produces the permission block, copyright information and page numbering). For use with ACM_PROC_ARTICLE-SP.CLS V2.6SP. Supported by ACM.

Galax [5] is our own implementation of the family of XQuery 1.0 specifications [2] and is designed to be complete and conformant with respect to the W3C standard. Being one of the very few publicly available XQuery implementations and one of the most complete, Galax has gathered a small user community and is the implementation of choice in many university database classes. More recently, Galax has been used to support constraint checking within Lucent's UMTS wireless infrastructure which is now on trial at several customers sites. This application requires support for many of the advanced XQuery features notably node identity, recursive user-defined functions, and type conversions. Moreover it has to routinely process documents of several hundred megabytes, which is beyond the scope of all public XQuery implementations we are aware of.

Jungle

The goal of the Jungle project is to develop the simplest XML storage manager that can support the following requirements.

Completeness. The storage manager must support all of the operations of the XQuery data model [16]. This requirement is sufficient to ensure full XQuery 1.0 support. Galax implements an abstract XQuery 1.0 data model interface which facilitates the use of a different physical representation of the document, and is used on top of Jungle.

Scalability. The storage manager must support efficient representation of XML documents, and scale to documents of the order of a few hundred megabytes. The response time for typical UMTS constraints must be of the order of a few seconds per constraint. In Section 4, we will show performance results for query evaluation time on document stored in Jungle.

Reliability. Jungle must be reliable enough to be deployed in industrial applications. This requirement was the main driver behind our decision to favor simplicity and the use of well-established technology. In particular, Jungle is based on standard B-Tree indexes, for which there exist good public implementations[1], rather than to develop novel XML indexes. Our experience throughout the Galax's development has been that the simplest solutions are often best: this results in code which is more robust, easier to maintain, as well as easier to extend in the future.

In this paper, we describe the design and implementation of Jungle, and provide preliminary performance results that demonstrate the effectiveness of the proposed approach. Although not officially released yet, the jungle engine can be downloaded from the Galax CVS development server[2].

Related work

Numerous strategies have been proposed for storing and indexing XML documents [11, 3, 8, 4, 7, 10]. Most of those indexes [11, 3, 8, 4] act as secondary indexes, allowing to speed up evaluation for specific subset of the query, but do not support all of the XQuery data model. Other approaches, like Natix [10] require the development of new specific kind of XML indexes, and therefore do not fit our requirement of being based on standard B-Trees.

The approach presented by Grust in [7] is appealing for several reasons. First, it can be used to support the XQuery data model in its entirety, including node identy, and all of the XPath axis. Second, it is based on tables and as a result can easily be implemented using B-Trees. In the rest of the paper, we will refer to that approach as *Accelerated XPath*. Jungle implements an indexing approach which is inspired from Accelerated XPath. However, a direct implementation of Accelerated XPath revels that two key operations: the child axis and serialization, are prohibitively slow. In Section 2 we will considered two alternative designs that we considered for Jungle. The first includes an index on a node's children and one on its attributes. The second includes one index on a node's first child, one on its next sibling, and one on its attributes. We are still investigating these two alternatives. The former substantially improves the performance of the child axis, whereas the latter improves performance of serialization.

Organization

In Section 2, we briefly recall the Accelerated XPath approach, then present two alternatives indexing approaches that are used in Jungle. In Section 3 we describe Jungle's implementation. And in Section 4, we present experimental results, notably comparing the performances of Jungle against the original Accelerated XPath approach. Section 5 reviews alternative indexing strategies, and Section 6 concludes the paper with some discussion about future work, notably about support for XML updates.

[1]Jungle is currently developed on top of BerkeleyDB [1].
[2]`ncc.research.bell-labs.com:8081/cgi-bin/cvsweb`

Table/Index	Key	Content
Main	pre	par-pre, kind, id (, *post*)
Text	id	text
QName	id	QName
QName-ID	QName	id
Prefix-URI	prefix	URI

Table 1: Core storage stuctures

```
<ex:a xmlns:ex="http://ximep/example.xsd">
 <ex:b><ex:c d="d"><ex:e/></ex:c></ex:b>
 <ex:f><ex:g/><ex:h i="i">j</ex:h><ex:k/></ex:f>
</ex:a>
```

Figure 1: Example Document

2. JUNGLE INDEXES

We first recall the Accelerated XPath approach before presenting the two alternative *children*, and *sibling* indexes implemented in Jungle. All three approaches share of the same *core* storage structures, which are necessary and sufficient to implement the XQuery data model. These core structures contain all of the information needed to reconstruct the original XML document. Each approach then has additional *auxiliary* structures, in order to support efficient access to particular parts of the document's structure or content, notably for specific XPath axis.

The Accelerated XPath approach

Table 1 describes the core structures for all three techniques and are similar to the main table of Accelerated XPath. The main table contains one record per node, keyed on the node's pre-order number, which is obtained during document loading. Each record contains the pre-order number of the node's parent node, the node kind (element, attribute, etc.), the identifier of its QName in the QName table, and an overloaded field whose meaning depends on the node kind. For a text node, for example, the overloaded field contains the identifier of the text node's value in the Text table. As an example, Figure 2 contains the core structures for the document fragment given in Figure 1.

Because there are many redundant QNames in documents, the QName and Prefix-URI tables serve as compact "string pools". The QName table maps concise ids to pairs of a namespace prefix and a local name, and the Prefix-URI table maps concise prefixes to long URIs. Note that the Accelerated XPath design proposed in [7] also stores the post-order for each node, as indicated in italics on Figure 1. In fact, we do not use post-order in Jungle. This is one of the main departure from [7], and will be explained later.

In this approach, a node's pre-order number also serves as its node identifier, which makes comparison of two nodes' relative order very cheap. In the presence of updates, however, maintaining pre-order numbers, in the worst case, is $O(N)$, where N is the size of the document. We return to the issue of node identity and node-numbering schemes in Section 6.

In the Accelerated XPath approach, all axes are evaluated

Pre	Par-Pre	Kind	QName-id	Value-id
1	-	elem	1	-
2	1	elem	2	-
. . .				

ID	QName
1	("ex","a")
2	("ex","b")
3	("ex","c")
. . .	

ID	Text-Value
1	"d"
2	"i"
3	"j"

Prefix	URI
"ex"	"http://ximep/example.xsd"

Figure 2: Core tables for example document

by scanning records in the main table. The number of records scanned is determined by applying a two-dimensional "window" to the main table that filters the nodes for a given axis. One dimension of that window consists of a pre-order range and the other of a post-order range. A window reduces the total number of records that must be scanned to compute the axis. For example, the descendant-axis window is:

```
<(pre(v), post(v) + height(v)],
 [pre(v)-height(v), post(v)), *, elem, *>
```

This window is applied to the main table as follows. To compute the a node v's descendants, select those records in the main table with pre-order number in the range pre-order of v to post-order of v plus the height of v. Similarly, the records must satisfy the remaining constraints, which are on the node's post-order value, its parent pre-order, its kind, and overloaded value. The * wildcard denotes any value is permissible.

Limitations

We started by implementing the Accelerated XPath approach as is. However, a direct implementation exhibits two important limitations. The first limitation has to do with the evaluation of the child axis. In fact, the child axis has the same complexity as evaluating the descendant axis, and requires a large scan over the main record structure. To illustrate this point, here is the child-axis window:

```
<(pre(v), post(v) + height(v)],
 [pre(v)-height(v), post(v)), pre(v), elem, *>
```

The only difference between the child and descendant windows is that the pre-order number of a candidate node's parent must equal v's pre-order. As a result, `child::` turns out to be as expensive as `descendant::` which is not acceptable for a lot of queries.

The second limitation has to do with the efficiency of serialization. Serializing the result of a query (or similarly exporting the original document back into XML) requires accessing a node's descendants in document order and is implemented as a recursive walk of each node's children. This recursive walk step by step turns out to be significantly inefficient using the window access described above. For a single node, its children are accessed (with cost equivalent to accessing its descendants), these children are pushed on a stack, and the procedure is applied recursively to each child on the top of the stack. For example, serializing node `f` in Figure 1 requires scanning the following nodes:

Node	Children	Node-records scanned
f	(g,h,k)	(g,h,i,j,k)
g	()	(h,i,j,k)
h	(i,j)	(i,j,k)
i	()	(j,k)
j	()	(k)
k	()	()

Obviously, this is not optimal, because as successive windows are are used to scan many nodes' records multiple times. Our very first implementation of Jungle, which was based solely on the Accelerated XPath approach, exhibited significant overhead during serialization.

The children index

The first alternative design that we considered was the addition of a specific auxiliary index on the children of a node one on its attributes. The child index maps the pre-order number of each element node to the sequence of pre-order numbers of the node's children. Figure 3 contains a fragment of the children and attribute indices for the document in Figure 1.

Key	Children
1	(2,6)
2	(3)
. . .	

Key	Attributes
1	()
3	(4)
. . .	

Figure 3: Children and Attribute Indices

The obvious advantage of this strategy is that accessing a node's children and its attributes requires constant time. In this strategy, the descendant axis can be implemented by a generic recursive walk of nodes' children, which unlike the window technique, does not require accessing the main table nor requires comparisons on main-table fields. Serialization is similar to descendant: a node's children are pushed on a stack, and the procedure is applied recursively to each of its children, in document order. Using that approach, each node in the main record structure must only be accessed once. As we will see in the experiments in Section 4, the children index substantially improve evaluation of the child and descendant axes, as well as serialization.

The sibling index

Our second alternative design replaces the children index with two indices: one on a node's first child and one on its next sibling. This strategy corresponds to the decomposition of unranked trees into binary trees. Figure 4 contains a fragment of these indices for the example document.

One of the motivation for this design is to be more closely aligned with accesses to nodes sequences as cursors (or streams) of items at the data model level. The original Galax evaluation engine implemented item sequences as main-memory

Key	First-Child
1	2
2	3
. . .	

Key	Next-Sibling
2	6
4	5
. . .	

Figure 4: First-child and Next-sibling Indices

lists in the physical data model, and the physical operators only accepted materialized lists of items. Currently, Galax's physical data model includes both materialized sequences and streamed sequences as well as materialized sequences or streamed sequences of tuples. The children index naturally correspond to materialized sequences, whereas the first-child and next-sibling indices naturally correspond to streamed sequences. Their inherent benefit is that both the child and descendant axes and serialization never require materialization of intermediate sequences.

However, the sibling index is somewhat less compact that the children index. As a result, we are still investigating the trade offs between the two approaches. In general, it appears that the children approach speeds up operations that require access to nodes in a bread-first fashion (e.g., `child::`), while the sibling approach speeds up operations that require access to nodes in a depth-first fashion `descendant::` or serialization).

About loading, post-order, and clustering

In the Accelerated XPath technique, the post-order number, which is required to compute an axis' window, is stored in the main table. In our two alternative design the post-order number is not required in the main table anymore. We currently exclude the post-order number from the main table and instead build an auxiliary index from pre-order to post-order numbers. This permits us to write records to tables in document order as soon as the pre-order number is known. If the post-order number is included in the main record, then during document loading, a record cannot be written into the table until after the end-element event, or the record must be updated once the post-order number is known. We found that updating the main record with the post-order number substantially slowed loading, whereas writing the records in reverse document order slowed down the window operations, which can benefit from having records clustered in document order.

3. JUNGLE IMPLEMENTATION

Galax's Data Model

Although we focus here on the Jungle secondary storage manager, we note that Galax can operate over any implementation of its abstract data model, of which Jungle is just one implementation. In addition to Jungle, there are several other implementations of Galax's abstract data model: a main-memory implementation, similar to the DOM, and an implementation that provides an XML view of a proprietary streaming data format [6]. Galax's abstract data model requires that an implementation support the parent, children, and attribute axes natively. Galax provides generic implementations of the other axes, such as the descendant, ancestor, and the sibling axes, but permits an implementation to provide native implementations of these axes if they so choose. For example, the generic implementation of descendant is implemented as a depth-first, recursive traversal of the child axis. This design permits Galax to operate on many kinds of data sources and permits implementations to add native axis support incrementally.

```
doc("jungle:///tmp/Example#test")

test-main.db       Record
test-Namespace.db  BTree prefix -> URL
test-Text.db Hash table from IDs to text nodes (strings)
test-AttrIndex.db  BTree ID -> ID
test-ChildIndex.db BTree ID -> ID
(string pools:)
test-QName2QNameID.db Hash tables
test-QNameID2QName.db Hash tables

accel XPath: no AttrIndex or Childindex, only:
test-PrePost.db

first/sibling:
test-AttrIndex.db  BTree ID -> ID
test-FirstChildIndex.db BTree ID -> ID
test-NextSibIndex.db BTree ID -> ID

jungle-load -store_dir /tmp/Example -store_name test foo.xm
```

NB: Interface

NB: Architecture/Implementation

We implemented the indices using BerkeleyDB [5] as a storage manager/index engine. BerkeleyDB can store key-data pairs only.

Physical Indexes

Jungle uses the Berkeley DB database product [1] to implement the indexes described in Section 2. Berkeley DB provides tables of key/value pairs and B-Tree indices over tables. Although some design choices were made with Berkeley DB in mind, most of the choices should apply to any generic relational database.

4. EXPERIMENTS

We assess the three indexing scheme in terms of loading time, storage size, average children access time, average descendant access time and serialization. For the purpose of this comparitive study we implemented the three indexing scheme using BerkeleyDB-4.2.1.

We decided to use XMark xml benchmarking tool in our experiments as it allows to generate large XML documents of different sizes, conforming to a standard schema. Since jungle don't handle typed documents we don't use the xmark schema in our experiments.

We carried our experiments on a 2.6 GHz Intel Pentium 4 processor machine with hyperthreading with 1 GB of main memory and running Linux 2.6.4. In order to reduce the inconsistencies we performed each experiment 5 times, discarded the highest and the lowest number and took the median of the remaining.

NB: Loading and Footprint Loading time is refered to the time taken to store a XML document using a selected indexing scheme and Footprint is refered to the total space occuppied by all the indices used in a selected indexing scheme.

Table 4 gives the loading time (seconds) and footprint (megabytes) for xml documents of different sizes under the three indexing schemes. Loading time is reasonable and approximately the same for the three indexing schemes. Footprints under the three indexing scheme are also reasonable and comparable to that of typical main memory dom implementations. For e.g. main memory dom materialized by Xerces is in general 5 times the size of the xml document.

Although Children technique and Next Sibling technique stores the same kind of information, use of fixed size data in the later approach yields a compact representation and slightly better loading time.

In accelerated XPath technique we need to create fewer index structures but creating postorder index during parsing the document in document order results in continous rebalancing of the binary tree which stores the postorder. This results in same loading time and a 12 percent higher footprint in comparison to the children approach.

Table 2: Loading time and Footprint

Document	Children		Next Sibling		Accel. XPath	
Size (mb)	time (sec)	size (mb)	time (sec)	size (mb)	time (sec)	size (mb)
1	2	4.72	2	4.26	2	5.22
5	15	20.96	12	20.18	15	23.44
10	31	42.09	25	40.48	30	47.21
15	47	54.21	37	61.38	45	61.67
20	66	86.34	53	84.29	64	96.71
25	83	97.86	67	105.59	79	110.93
30	101	145.14	84	127.14	95	161.12

NB: Child Axis

As mentioned earlier access to children is a basic and important datamodel operation. We refer the time required to access all the children of a node using a selected indexing technique as children access time. Table 4 gives the average children access time for randomly selected node and nodes accessed in document order.

To access children for a node, accelerating XPath technique accesses all the descendant in comparison to children index technique which requires a single lookup to find all the children and hence later is faster by a factor of approx. 10. The overhead of multiple lookup in case of next sibling approach reflects in the 40 percent increased children access time.

NB: Descendant Axis Descendant access time is refered to the time required to access all the descendant of a node in a selected indexing technique. In Children index technique and first sibling technique we evaluate descendant

Table 3: Children Access Time

Document	Children		Next Sibling		Accel. XPath	
Size	Random	Serial	Random	Serial	Random	Serial
1	8.88	8.58	12.69	11.88	51.92	56.58
5	10.02	9.12	14.21	12.58	61.23	62.98
10	10.31	9.27	14.63	12.71	66.19	63.34
15	10.51	9.35	14.77	12.81	58.18	63.65
20	10.62	9.37	15.03	12.90	49.89	63.76
25	10.73	9.36	15.36	13.02	68.70	64.18
30	10.78	9.38	15.35	12.51	50.46	64.39

using recursive calls to children, where as accelerated xpath technique uses the descendant window.

As shown in Table 4 the descendant access time in children index technique is faster by a factor of 3 in comparison to accelerated XPath technique and by a factor of 2 in comparison to next sibling approach. Although the children axis and accelerated XPath techniques access the same number of records the overhead in later is result of window comparison.

Table 4: Descendant Access Time

Document	Children	Next Sibling	Accel. XPath
Size	Avg	Avg	Avg
1MB	22.94	45.46	108.34
5MB	56.72	108.62	183.65
10MB	70.53	122.79	206.81
15MB	62.66	110.44	193.02
20MB	68.88	125.39	208.40
25MB	73.12	131.41	217.17
30MB	83.73	146.85	222.30

NB: Cursor Based Child Access

Table 5: Children Access Time

Document	Children	Next Sibling	Accel. XPath
Size	Random	Random	Random
1	9.22	12.83	52.56
5	10.56	14.57	62.64
10	10.73	14.77	67.34
15	10.58	14.68	58.63
20	10.81	14.93	50.05
25	10.98	15.27	69.67
30	11.04	15.42	50.83

NB: Serialization

5. RELATED WORK

Indices in recently proposed indexing schemes can be roughly categorized into *Structure* indices, *Value* indices and *Path* indices. Example of a structure index is the children index, that of value index are the element name index and the attribute value index. Path index as the name suggest, could be considered as the hybrid of the structure and value index as it encodes information about structure as well as value.

Timber [9] (maybe) complete implementation of XQuery. Uses Shore as storage manager. (Pre, Post, Level) label-

Table 6: Serialization

Document	Children	Next Sibling
Size	Avg	Avg
1MB	3	3
5MB	15	16
10MB	30	29
15MB	46	46
20MB	64	64
25MB	79	80
30MB	95	96

ing for efficient ancestor/descendant access. Same problem as accelerated-XPath technique – inefficient child axis and serialization. Closest to our work vis a vis scope.

Storage and indexing techniques for XML is an active research area.

Structure: Grust [7]

Value:

Cooper et al [3] : Path index, cannot recover original structure of document, use Patricia tries to encode paths to values in documents. Cannot recover relative document/sibling order of internal nodes in XML document.

Li and Moon [11] : contains both structure and value indices. For each tag name, can access extent of all nodes with that tag name. First of techniques that use numbering to support efficient implementation of axes (ancestor-descendant). Decomposition of regular path expressions into small path expressions, whose results can be joined.

Tatarinov et al [14] : hybrid of structure and path techniques. Core is Edge storage, instead of storing tag name, store path of tags.

Path: ViST [8] (pre-order representation of tree, each node represented by a symbol, document is string of pre-order walk of tree. Permits efficient implementation of path expressions. Unclear how serialization of results is handled, also document-order queries)

It is understood that no single indexing scheme is best for all types of queries. For example path indices are better than structure indices in evaluation of simple path queries, but they do not address the needs of branching queries and queries with predicates. Without additional information path index cannot ensure the correct document order in the query result, while document order is inherent in many structure indices.

The key factor in evaluation of indexing schemes is the "Access Time" i.e. the time it takes to find a particular data item. Equally important factors in evaluation of an indexing scheme are insertion time, deletion time and space overhead. Most of the current work mainly considers queries i.e. the access time, and few [12, 13] focus on reducing the index overhead. There is no emphasis on the cost involved in maintaining those indices in the presence of updates. Indexes are always difficult to maintain in presence of updates and so is clustering of the data.

6. CONCLUSION & FUTURE WORK

NB: Updates

We realize that clustering schemes and their maintainance is important for query performance, in this paper we focus mainly on former.

Issues in maintaining indices in presence of updates in relational databases are well known. XML imposes new requirement of maintaining document order in presence of updates. Inserts in XML could be postional which necessitates reordering of the data in addition to indices.

Typically node identifiers [16] apart from uniquely identifying the nodes in the tree representation of the document, have been used for ordering nodes and encoding containment relationships among the nodes. The main implication of supporting updates is thus support the renumbering or relabelling of these node identifiers.

Recent work in indices which support updates [15, 9, 8], try to delay renumbering of the nodes by leaving space between nodeids allocated to two successive nodes. A typical example is to use of floating point number for nodeids. Due to localized insertions or insertion of large subtrees, the space at a insert location may run out thus node renumbering becomes unavoidable.

To the best of our knowledge, no indexing scheme facilitates node renumbering or proposes an incremental node renumbering algorithm.

6.0.1 Problem with updates

In presence of updates, one cannot avoid node renumbering completely. Most of the recent work [9, 15] focuses on node numbering schemes to delay the process of renumbering the nodes as much as possible. When node renumbering becomes unavoidable they do renumbering of the complete document tree.

We identify node renumbering as a major issue affecting choice and design of XML indexes supporting updates. Also any additional indexes created to improve the query preformance needs to be updated.

For example in this scheme in addition to updating the main records, we also need to reflect the changes of node renumbering in child index.

If we want to insert a subtree shown in Fig 5. as a child of "c" between "d" and "e". If we were to do node renumbering at this point, we need to renumber every node which follows node "d" in pre-order traversal of the original tree. In the Fig 2 (again) such nodes are marked with "-¿"

The cost and complexity for updating a Child index such as above would be high. Infact a complete scan of the index is required to update it.

7. REFERENCES

[1] Berkeley DB product. http://www.sleepycat.com/.

[2] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0 An XML Query Language, W3C Working Draft, Nov 2003. http://www.w3.org/TR/xquery.

[3] B. Cooper, N. Sample, M. Franklin, and M. Shadmon. A fast index for semistructured data. In *VLDB*, September 2001.

[4] K. Deschler and E. Rundensteiner. MASS: A multi-axis storage structure for large XML documents. In *CIKM*, 2003.

[5] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax Experience. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 1077–1080, Berlin, Germany, Sept. 2003.

[6] K. Fisher and R. Gruber. Pads : Processing arbitrary data streams. In *Workshop on Management and Processing of Data Streams*, June 2003.

[7] T. Grust. Accelerated XPath location steps. *ACM SIGMOD*, 15(5):795–825, June 2002.

[8] W. F. Haixun Wang, Sanghyun Park and P. S. Yu. ViST : A dynamic index method for querying XML data by tree structures. In *ACM SIGMOD*, pages 110–121, June 2003.

[9] H. Jagdish et al. Timber: A native XML database. *ACM SIGMOD*, page 672, June 2003.

[10] C. C. Kanne and G. Moerkotte. Efficient storage of XML data. 2000.

[11] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *VLDB*, September 2001.

[12] H. K. S. Abiteboul and T. Milo. Compact labelling schemes for ancestor queries. In *ACM-SIAM Symposium on Descrete Algorithms (SODA)*, January 2001.

[13] S.Alstrup and T. Rahue. Improved labeling scheme for ancestor queries. In *ACM-SIAM Symposium on Descrete Algorithms (SODA)*, January 2002.

[14] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. *ACM SIGMOD*, pages 204–215, June 2002.

[15] M. Y. Toshiyuki Amagasa and S. Uemura. QRS : A robust numbering scheme for xml documents. *ICDE*, pages 291–301, June 2003.

[16] W3C. XQuery 1.0 and XPath 2.0 data model. *W3C*, 15(5):795–825, June 2002.