

Performance Evaluation and Enhancement of XML store for Database Applications

Thomas Mørup Helmudt

Klaus van Berkel

University of Copenhagen

19 May 2004

Abstract

Relational databases has existed for decades, and has been fine tuned for performance and reliability, this can not be said of the existing XML storage facilities. Most of them have problems with performance and are still suffering from child diseases, which cause them to behave unpredictably when handling data. Experimental tests shows how the chosen storage strategy has a compound effect on performance; and that handling of large amounts of data is not supported very well in the commercial XML storage systems.

XMLStore from the Plan-X project is an application configurable, distributed, mobile persistence layer for storing semistructured data (in particular XML documents) based on a value-oriented document model and interface. Together with XPath as query language operating on its data, it can be considered a native XML database system.

The goal of this project is to evaluate how the XMLStore from the Plan-X project performs when executing database operations on XML, compared to other XML storage solutions. To be more specific the insights gained from a comparison of the XMLStore with other approaches both on the conceptual level and from empirically knowledge will be used in the evaluation. If the insights gained provide enough information on where the XMLStore could be changed to improve performance this will either be implemented or described for future implementation.

Overall we feel that the XMLStore is a solid and stable application, we did not experience any problems at all using it. But to be competitive with commercial systems we suggest that further development is needed on enhancing performance of querying, since querying is not efficient enough compared to other XML storage technologies. It is especially interesting that the best querying engine was a freeware implementation querying XML stored in flat text files.

1	BACKGROUND AND GOAL	5
<hr/>		
1.1	METHOD	5
1.1.1	SOME TERMINOLOGY	6
2	STORAGE STRATEGIES	6
<hr/>		
2.1	THE APPROACHES FROM [TDCZ02]	7
2.1.1	THE TEXT APPROACH	8
2.1.2	THE RELATIONAL DTD APPROACH	10
2.1.3	THE EDGE APPROACH	11
2.1.4	THE ATTRIBUTE APPROACH	12
2.1.5	THE OBJECT APPROACH	13
2.2	XML-STORE	14
2.2.1	XML-STORE COMPARED TO THE DIFFERENT APPROACHES	16
3	PERFORMANCE EVALUATION	19
<hr/>		
3.1	EXPERIMENTAL TESTING	19
3.2	PHYSICAL AND LOGICAL CONFINES	20
3.2.1	POSSIBLE SOURCES OF ERRORS	20
3.3	DATABASE	21
3.3.1	STORING DATA	22
3.3.2	QUERIES	22
3.4	STANDARD XML QUERYING COMPONENTS	23
3.4.1	COLD START	23
3.4.2	SERVER	24
3.4.3	QUERYING	24
3.5	TAMINO AND ORACLE TWO COMMERCIAL SYSTEMS	25
3.5.1	TAMINO V. 4.1 DB	25
3.5.2	ORACLE V.9.2 DB	26
3.6	XMLSTORE	26
3.6.1	XMLSTORE DECORATORS	26
3.6.2	STORING DATA	28
3.6.3	QUERYING DATA	30
3.6.4	PROFILING AND OPTIMIZATION	31
3.6.5	COMPARING OPTIMIZED XMLSTORE WITH OTHER IMPLEMENTATIONS	34
4	ENHANCEMENTS OF THE XMLSTORE	36
<hr/>		
4.1	USING SCHEMAS TO OPTIMISE THE QUERY STRATEGY	36
4.2	OPTIMIZE LOADING	36
4.3	REDUCING OBJECTS WITH THE FLYWEIGHT PATTERN	36
4.4	INDEX	36
5	STATE-OF-THE-ART XML STORAGE TECHNOLOGIES	38
<hr/>		

LITERATURE	39
-------------------	-----------

APPENDIX A	40
-------------------	-----------

Acknowledgements

We would like to thank our supervisor Fritz Henglein for invaluable ideas and motivation. We would also like to thank Henning Niss for answering questions about the XMLStore and helping with the implementation of performance improvements. And at last we would like to thank Thomas Stilling Ambus for answering questions about the XPath query engine.

1 Background and Goal

XML has gained more and more focus over the last years, as it has become a standard way of exchanging data. One of the areas where XML can be used is business to business communication; this can be achieved by service oriented architectures, which externalise public business services. These services can be activated remotely by other companies, using XML as the data carrier of request and response.

To make the services of one business publicly known for others to find they can be registered in a central service repository. The processes describing inter-business communication can be stored and retrieved from XML repositories; the services made available by one business can be registered in these central databases, for others to subscribe to.

The above described scenario is just one of the reasons why we believe that XML will gain even more focus in the future. Reliability and fast storage facilities for storing large amounts of semistructured XML data is of paramount importance, if an XML Registry and Repository should become a reality.

Relational databases has existed for decades, and has been fine tuned for performance and reliability, as [GB03] and [TDCZ02] shows this can not be said of the existing XML storage facilities. Most of them have problems with performance and are still suffering from child diseases, which cause them to behave unpredictably when handling the data. [TDCZ02] shows how the chosen storage strategy has a compound effect on performance; and [GB03] shows that handling of large amounts of data is not supported very well in the tested commercial XML storage systems. Fritz Henglein describes an alternative XML storage facility: XMLStore of the Plan-X Project.

XMLStore is an application configurable, distributed, mobile persistence layer for storing semistructured data (in particular XML documents) based on a value -oriented document model and interface. Together with XPath as query language operating on its data, it can be considered a native XML database system.

The goal of this project is to evaluate how the XMLStore from the Plan-X project performs when executing database operations on XML, compared to other XML storage solutions. To be more specific the insights gained from a comparison of the XMLStore with other approaches both on the conceptual level and from empirically knowledge will be used in the evaluation. If the insights gained provide enough information on where the XMLStore could be changed to improve performance this will either be implemented or described for future implementation.

1.1 Method

The method we use to reach this goal is as follows. First we will describe several approaches on storing XML on the conceptual level including the concepts of the XMLStore; then we will discuss similarities and differences between the different approaches. This description and comparison can be found in section **2 Storage strategies**.

Secondly we will perform an empirical evaluation of the XMLStore by comparing experimental test results with the results of [GB03]; where two commercial XML database implementations were tested, namely a native XML database and a XML enabled RDBMS. We will also perform some tests on XML querying components working on XML stored in flat text files; this is the simplest approach of storing and querying XML and will be used as a benchmark. All results of the tests will be compared and discussed in section **3 Performance evaluation**.

Based on the theoretical and empirical evaluations, we will discuss possible performance improvements of the XMLStore, in regard to database functionality. This can be found in section 4 **Enhancements of the XMLStore**.

In section 5 **State-of-the-art XML Storage Technologies** we will recollect all the insights we have gathered on state of the art XML storage technologies.

1.1.1 Some terminology

Before we start we will introduce some terminology that we use throughout the whole report, in each subsequent section we will define specific terms only used there.

We use the term XML to denote semi-structured data encoded in XML syntax¹. The data is structured in a set-containment relationship between an element and its child elements, represented by a rooted directed acyclic graph (DAG). We will refer to such a rooted DAG as an XML document.

An XML document is a textual representation of data; where the basic component is an element, i.e. some text enclosed by matching tags such as `<person> ... </person>`, or an empty element like `<children/>`. A tag can have associated information in the form of (name, value) pairs called attributes; an attribute's value is interpreted as CDATA². The content inside an element can either be other elements, PCDATA³ or a mixture of the two. If the latter case is dominating the XML document is said to be text centric (since the interspersed elements are usually textual formatting). The opposite of text centric documents is data centric documents, in such documents the structure is usually stricter, in the sense of conforming to some sort of schema.

A tag in an XML document will always be referenced by the convention name-tag for example person-tag referencing `<person/>`. When referring to the value of an element we mean whatever is between two matching tags.

When referring to schemas describing the structure of XML documents in the following, we will not restrict ourselves to some specific implementation (e.g. DTD or XML Schema etc.); we will refer to a schema as “*a description of data in terms of a data model*” .

2 Storage strategies

This section describes the basics of the five different approaches to store XML documents based on the work of [TDCZ02], and the XMLStore described in [BT02].

The presentation of each approach will use the following structure. We will start with describing the approach in general and give a representation of how data is structured and will be discussing advantages and disadvantages of each approach; including querying/updating data and concurrency control. At the end of this section we will compare the described approaches with XMLStore describing pros and cons. The [TDCZ02] does not mention how locking/concurrency control strategies are implemented throughout the different approach but we will discuss this for each approach based on our own knowledge.

The XMLStore is based on value-oriented programming. To describe value-oriented programming in one sentence, it is “*programming with value s and value references*” [BT02]. The concept in value oriented programming is that values are immutable entities, in object oriented

¹ Fore more information on XML syntax see [XML].

² CDATA sections are used to escape blocks of text that would otherwise be interpreted as data.

³ Parsed Character Data is the part of the document not enclosed in angle brackets [ABSK99].

lingo it would be an object that can not change its state. An example from Java is the String API where all modifying methods will create a new string instead of changing the original object.

The expression “attribute” is defined in many of the technical areas discussed in the following among others: object oriented programming and database theory. We will however when referring to attributes use the definition of XML attributes as described in section 1.1.1 unless otherwise noted.

2.1 The Approaches from [TDCZ02]

When describing the approaches in this section, we will supply a simple example of each approach, and for that we use the sample data shown in Figure 1.

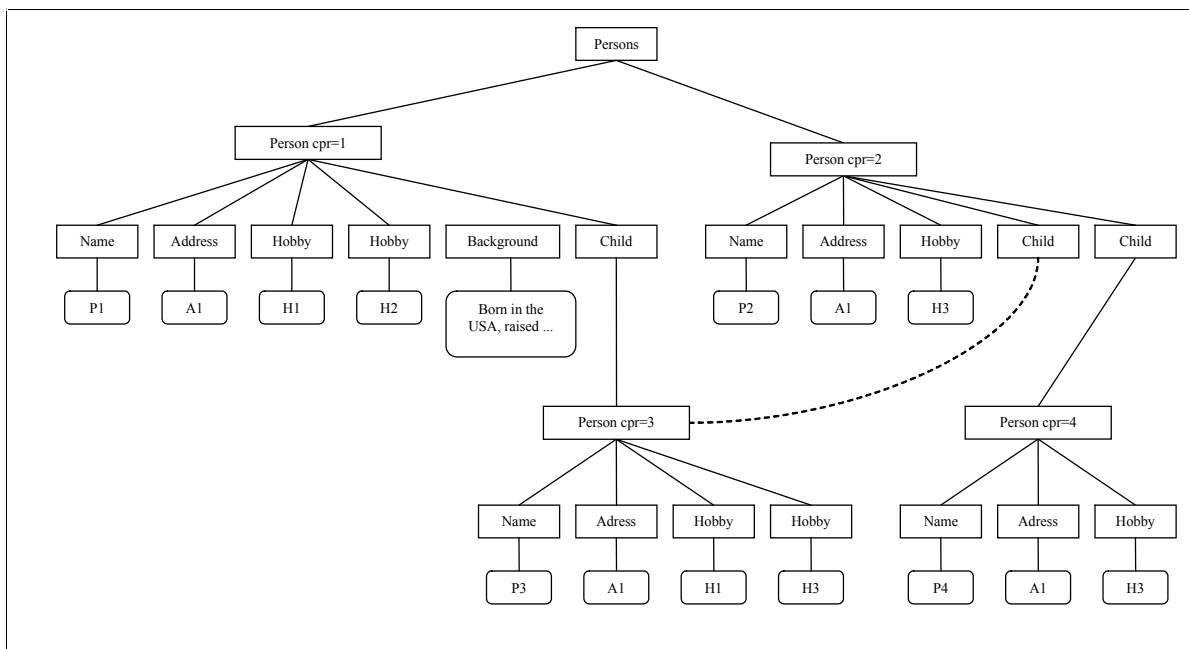


Figure 1: Sample data in tree structured form.

The interesting part about this data representation is that a person element can be within another person element, and that person elements can be referenced across other objects; this reference is purely logical information not part of the XML definition of [XML], which means that some other logic than an XML parser should be applied to use this information. The XML document of the tree representation can be viewed in Figure 2.

```

<persons>
  <person cpr=1>
    <name>P1</name>
    <address>A1</address>
    <hobby>H1</hobby>
    <hobby>H2</hobby>
    <background>
      Born in USA,
      raised in Iowa,
      moved to NY
    </background>
    <child>
      <person cpr=3>
        <name>P3</name>
        <address>A1</address>
        <hobby>H1</hobby>
        <hobby>H3</hobby>
      </person>
    </child>
  </person>
  <person cpr=2>
    <name>P2</name>
    <address>A1</address>
    <hobby>H3</hobby>
    <child>
      <person cpr=4>
        <name>P4</name>
        <address>A1</address>
        <hobby>H3</hobby>
      </person>
    </child>
    <child id=3/>
  </person>
</persons>

```

Figure 2: Sample XML document.

2.1.1 The Text Approach

The first strategy is to store XML data in a text file in the file system. What makes this approach interesting is its simplicity; accessing data do not require the use of a full fledged database system, which highly simplifies configuration (of course all the benefits, like persistence and security, of such a system are not available either).

One way to implement the text approach is to parse the XML file into a memory resident tree structure, and then perform queries on this structure; although this approach requires the whole structure to reside in memory, while any of the nodes are still needed by the query. The time required for parsing the XML file, dominates query processing, which makes this approach unacceptably slow. Therefore [TDCZ02] has implemented another strategy where indexing is used, which makes the text approach more competitive.

Building an index on (*parent-offset, tag*) to *child-offset*, and a reverse index mapping children to its parents: *child-offset* to *parent-offset*; these indices makes it possible to traverse the document without parsing. To reduce parsing time even further indices on either (*tag-name, value*) or (*attribute-name, attribute-value*) to *element-offset* can be build. Table 1, Table 2 and Table 3 below show the indices that would be build over our example data.

Parent Offset	Tag	Child Offset
R1	<person>	R2,R3,R4,R5,R6,R7
R2	<name>	Nil
R3	<address>	Nil
R4	<hobby>	Nil
R5	<hobby>	Nil
R6	<background>	Nil
R7	<child>	R8
R8	<person>	R9...
R9	...	

Table 1: (parent offset, tag) to child offset representation

Child Offset	Parent Offset
R1	Nil
R2	R1
R3	R1
R4	R1
R5	R1
R6	R1
R7	R1
R8	R7
R9	...

Table 2: Child Offset to Parent Offset representation.

Attribute Name	Attribute Value	Element Offset
Cpr	1	R1
Cpr	3	R8
...		

Table 3: (attribute name, attribute value) to element offset representation

The query engine will benefit from the indices, by pinpointing where in the XML file to look for data, and only parse the needed tags⁴. Building indices over element values is not as easy as attributes, since attributes can only contain CDATA, and attributes are normally data-centric (though there is no restriction on this). Element values on the other hand, can contain other elements as well as large chunks of PCDATA (like the background tag in the persons file above), both of which are not ideal to build indices on, because they are hard to compare and the sheer size of values will make the size of the index grow as well. If the content is data-centric though, and the tags being indexed only contain PCDATA, it will work as a usable index. In fact, it will work like the attribute index.

An example query for data could be: *“select the hobbies of the person with cpr = 1”*. Assuming that data has been parsed and the indices are build the following happens: The attribute name *“cpr”* with value *“1”* is found in the *“(attribute name, attribute value table) to element -offset”* index. The element-offset is found and used to look up the children of the parent in the *“(parent offset, tag) to child offset”* index. The references to the children are iterated and where the tag is matching *“hobby”* the reference is used to access the text file to locate data. All children references have to be iterated due to the possibility of the tag *“hobby”* existing more than once within the person element. Reconstructing the original XML document on the other hand, executes in constant time by just returning the on-disk representation.

The main disadvantage of this approach is that whenever the XML document is updated, the element offset of preceding tags are also changed, which invalidates the indices and they have to be rebuilt. Regarding concurrency control it is necessary to lock both the XML document and the matching indices when some thread access data (reading/writing) due to data consistency. When a one thread is reading other threads can read as well, but when some thread is updating other threads can not read or update the whole document since it can not be considered consistent. The worst case is of course if new threads continue to access the document for reading, then it will not be possible to update any part of the document, unless some sort of prioritizing algorithm is implemented (and updates are given higher priority, of course this could lock out reads).

⁴ This is not entirely true, since the query engine has to parse the whole XML document at least once to create the indices; when the indices have been built they can be used for subsequent queries as well. A SAX parser can be used efficiently so the whole document does not have to reside en memory at once.

2.1.2 The Relational DTD Approach

This strategy stores data in relational tables. It requires the existence of a schema and uses this information to create the tables (hence the name of the approach). One relational table per unique tag is created, and all attributes of the tag are stored within separate columns in the table. If a tag has no attributes, and its parent is restricted to contain only one such tag, it will be treated as an attribute. The parentID and the ID columns constitute the set containment relationship between the parent element and the child element. If a cycle is represent within the schema, a separate relational table has to be created to break the cycle. This table will be used to contain the set-containment relationship between an element and its children with the same tag.

ParentID	ID	Cpr	Name	Address	Background
1	2	1	P1	A1	Born in USA, raised in Iowa, moved to NY
5	6	3	P3	A1	
1	9	2	P2	A1	
12	13	4	P4	A1	

Table 4: person

ParentID	ID	TEXT
2	3	H1
2	4	H2
6	7	H1
6	8	H3
9	10	H3
13	14	H3

Table 5: hobby

ParentID	ID	id
2	5	
9	11	3
9	12	

Table 6: child

This approach saves space on disk since tag and attribute names are only written once; the longer the tag name, more space is saved. If the data representation from Figure 2 is presented through the Relational DTD approach, storage will be reduced because the data representation will be limited to the ParentID, the ID and the values. The worst case XML document for this approach is consisting of tags stored in pairs with short tag names, example can be viewed in Figure 3.:

```

<person>
  <a>1</a>
  <a>2</a>
  <b>1</b>
  <b>2</b>
</person>

```

Figure 3: Worst case XML document for The Relational DTD approach.

In such a document the set-containment relationship stored in the tables will decrease the savings, and if the ids are longer than the tag names the overhead will be the difference.

[TDCZ02] suggest building indices on the ParentID and ID columns, as well as on each column containing XML data values. The latter index is however not very useful if data is text centric and data consist of long strings of characters. How our example XML Document would look stored in relational tables using this approach can be found in Table 4, Table 5 and Table 6. Querying is done through the optimized database query engine using indexes as performance enhancers. Again a query as “*select the hobbies of person where cpr = 1*” combines the person and hobby table through a join and uses the given index to enhance searches for the specific values. If data is not represented in the indices, a lookup in the specified table is needed (tables are specified in the query).

How the locking strategy is implemented depends on the specific implementation. A locking strategy could be based on locking the smallest size database object possible, reducing the

likelihood that two transactions needs the same lock, or just to reduce the timeframe where the database object is locked. Locking strategies however takes a pessimistic approach where as a concurrency strategy can take a more optimistic approach, e.g. through a timestamp where an action ai of transaction Ti conflicts with another action aj of transaction Tj occurs if $\text{Timestamp}(Ti) < \text{Timestamp}(Tj)$. If this ordering is violated the transaction is aborted and restarted [RG03].

When reconstructing the XML document from this approach it is necessary to know how to build the document in terms of layout etc. Whether it is a partial or a full reconstruction does not matter because the work is the same, only when it is partial it is necessary to make specifications about which part one wishes to reconstruct. There is though a problem of recreating whitespace outside contents because this information is lost when the XML document is uploaded to the database.

2.1.3 The Edge Approach

The idea of this strategy is to represent data within one table. The strategy uses the XML document to create a DAG which is stored in a *single edge table* .

Each tuple in the table corresponds to one edge in the XML document. The sourceID and targetID are used for navigation throughout the tree structure. The two elements represent the edge between a parent and child or the parent and attribute data. If the value of the targetID is 0 then it is tuple represents an attribute. The sourceID also represents at which depth level a certain tag is located. The ordinal represents the order of the children of a specific element. Combined with the sourceID and the targetID the three elements represent the tree structure including the parents, children and sibling order. Hence there is no need for a schema to describe data. When a node contains only one text child the text is inlined with the given tuple within the table. Example on the edge table can be viewed in Table 7.

SourceID	Tag	Ordinal	TargetID	Data	SourceID	Tag	Ordinal	TargetID	Data
1	Persons	0	2	NULL	15	Name	1	16	"P2"
2	Person	0	3	"cpr=1"	15	Address	2	17	"A1"
3	Name	1	4	"P1"	15	Hobby	3	18	"H3"
3	Address	2	5	"A1"	15	Child	4	19	NULL
3	Hobby	3	6	"H1"	19	Person	0	20	"cpr=4"
3	Hobby	4	7	"H2"	20	Name	1	21	"P4"
3	Back-ground	5	8	"Born in .."	20	Address	2	22	"A1"
3	Child	6	9	NULL	20	Hobby	3	23	"H3"
9	Person	0	10	"cpr=3"	15	Child	5	24	NULL
10	Name	1	11	"P3"	24	Child	0	25	"Id=3"
10	Address	2	12	"A1"					
10	Hobby	3	13	"H1"					
10	Hobby	4	14	"H3"					
2	Person	0	15	"cpr=2"					

Table 7: The edge approach

Clustering the edge table has a significant impact on query performance depending on the clustering strategy. [TDCZ02] propose a strategy clustering the tag field which results in attributes with the same name are stored together making it easier to locate specific tags; another strategy is to cluster on the sourceID. The latter results in easier tree traversal, e.g. when locating all children to a given node. With certain queries such as "select all persons whose hobby is H3" the second strategy can give overhead in I/Os because the elements are not clustered together. Further if two predicates are needed for queries, i.e. "select all persons starting with A and has hobby H3". Here it is necessary to make a self-join on the edge table. Inserting new nodes into the database table is straightforward. If the node contains more attributes, one row is generated for each attribute.

[FK99] propose an index strategy based on the sourceID column which will speed up traversals such as reconstruction a specific node given its node id. An index on $(tag, target)$ will be useful for backwards traversal within the tree.

Concurrency control can be implemented on different levels. As in the Relational DTD approach it can be done through locking of a database object, here row or field level (if based on table locking no updates can be made throughout the whole document if just one tuple is read and all updating threads will be locked until all reading has terminated) or through a optimistic concurrency strategy as described in the section of the Relational DTD approach.

When reconstructing the XML document there is no difference in making a partial or full reconstruction as it is the same operation that makes final document. Whitespaces outside content though are lost due to the way data is represented in the edge table as no layout structure of the original XML document is saved.

2.1.4 The Attribute Approach

The definition of the term “attribute” in this approach does not equal the general description of the term “attribute” used through out in this report. If the name of the approach should equal a term defined here, the name should be “The Tag Approach” but we will stick with the original name.

This strategy equals the “Edge Approach” but with a horizontal partitioning of the table by the use of the tag field. The attribute approach divides data into several tables, here tuples with different tag descriptions are stored in separate tables. The strategy uses the ordinal to keep track of the numeric placement of the siblings within a given element, i.e. all the children of the person element. The data structure from figure no.1 is displayed in Table 8, Table 9, Table 10, Table 11, Table 12 and Table 13. One advantage of this approach is that if the tag name consist of many characters that disk storage overhead is minimized. A drawback of the approach is that large collections of XML documents can result in a large number of tables taking up space.

To enhance queries an index for each table can be made on the sourceID which would reduce search time for values. Additional a separate index on the targetID could be build for faster reconstructions of specific elements and its children given the ID. In general queries made in this approach rely on many join operations making operations more complex. Further it is necessary to have a schema to be able to locate how elements are connected, and also to be able to reconstruct the XML document. Again whitespace outside content can not be restored under reconstruct because the layout of the original document is not saved.

SourceID	Ordinal	TargetID	Data
1	1	2	“cpr=1”
3	1	4	“cpr=3”
1	2	5	“cpr=2”
6	1	7	“cpr=4”

Table 8: Attribute Approach: Person

SourceID	Ordinal	Data
2	1	“P1”
4	1	“P3”
5	1	“P2”
7	1	“P4”

Table 9: Attribute Approach: Name

SourceID	Ordinal	Data
2	2	"A1"
4	2	"A1"
5	2	"A1"
7	2	"A1"

Table 10: Attribute Approach: Address

SourceID	Ordinal	Data
2	3	"H1"
2	4	"H2"
3	3	"H1"
3	4	"H3"
5	3	"H3"
7	3	"H3"

Table 12: Attribute Approach: Hobby

SourceID	Ordinal	TargetID	Data
2	6	3	NULL
5	4	3	"Id = 3"
5	5	6	NULL

Table 11: Attribute Approach: Child

SourceID	Ordinal	Data
2	6	"Born in USA..."

Table 13: Attribute Approach: Background

Inserting new objects depends on the type of element that is entered. In general when inserting an element where it is necessary to uphold the ordinal numbering, i.e. when inserting a person element between two existing person elements at same tree depth level ordinal result in that all person elements that has the same sourceID (here the parent node) will have to be locked. If the ordinal is not important it is only necessary to know how many siblings exist and then enter the new element at the end of this row.

When reading or updating an element it is necessary to be able to lock or use some concurrency strategy. Depending of the number of transaction it is necessary to look at which level one wishes to lock. As mentioned in the relational DTD approach an optimistic concurrency strategy can be where either looking on the timestamp or minimizing the amount of time an database object can be locked. But if transactions are low a simple locking strategy based on the table can be implemented.

2.1.5 The Object Approach

The object approach divides the XML document into objects. The basics are that each element of the XML document is created as an object. This way of representing data can be very expensive in space allocation. [TDCZ02] has presented an optimized approach which we will adapt to our data representation from Figure 1. The optimized approach is changed so that XML elements are combined and serialized into a data format that represents one object, the *lw_object* (lightweight object). Additional the approach has an object that represents the XML document, the *file_object*. The structure of the *lw_object* is represented in Figure 4. The *lw_object* attributes (not to be compared with the XML attribute definition) contains the following: a *length* attribute which resembles the total length of the object - this attribute is also used for calculating the offset of each *lw_object*; a *flag* attribute - a flag displaying whether the object *opt_child*, *opt_attr* or *opt_text* are set; a *tag* attribute - is the tag name of the XML element; the *Parent* and *Opt_child* attributes - refer to the offset of the parent or the child if this exists; the *Prev* and *Next* attributes - siblings are located by these values. Additional XML-attribute representations such as "*cpr=1*" are inlined in the *Opt_attr* attribute and PCDATA are inlined in the *Opt_text* attribute.

Length	Flag	Tag	Parent	Prev	Next	Opt child	Opt attr	Opt text
--------	------	-----	--------	------	------	-----------	----------	----------

Figure 4: A representation of a *lw_object*

The *lw_objects* are represented in the *file_object*. The offset represent each *lw_object*. A subset of Figure 1 is represented in Table 14 as the *file_object* and the sub-elements as *lw_objects*. In this representation the offset field is only a visual effect to ease the understanding.

Offset	Record
0	Length=40, 0, Persons, parent=nil,prev=nil,next=nil First_child=40,last_child=...
40	Length=40, 1, Person, parent=0,prev=nil,next=... First_child=80,last_child=..., Attr(cpr="1")
80	Length=20, 1, Name, parent=40,prev=nil,next=100 No children, No attribute, #PCDATA="P1"
100	Length=20, 1, Address, parent=40,prev=80,next=120 No children, No attribute, #PCDATA="A1"
120	Length=20, 1, Hobby, parent=40,prev=100,next=140 No children, No attribute, #PCDATA="H1"
140	Length=20, 1, Hobby, parent=40,prev=120,next=160 No children, No attribute, #PCDATA="H2"
160	Length=20, 1, Background, parent=40,prev=140,next=180 No children, no attribute, #PCDATA="Born in USA..."
180	Length=40, 0, Child, parent=40,prev=160,next=nil First_child=200, Last_child=200
220	Length=40, 1, Person, parent=180, prev=nil, next=nil First_child=260, Last_child=..., Attn(cpr="3")
...	...

Table 14: Object Approach: Representation of the *file_object*

Making queries within the object approach can be done through traversing the tree structure through the parent-sibling-child representation, i.e. a query like “*select the name of the person with cpr=1*” where the search would begin at offset 0 where the tag would be checked if it fits the “*person*” value, if not, it would go deeper into the tree structure through the *child* attribute or through the *next* (sibling) attribute. A possible solution this query could also be to run through all objects by calculating the address from the previous address and checking the tag field for each object to match it with a person. If there is a match then check the attribute object for a match.

To enhance searches [TDCZ02] describes a strategy of building indices based on B+ tree implementation. An index is build on (*tag, opt_text*) to enhance searches on PCDATA with a specific tag name; another index build on (*parent_id, tag*) to *lw_oid* (the offset) to be able to retrieve the children of a node with a given tag.

With the representation of data as in Table 14, reconstruction of the original XML document or a subset of the original document is fast due to the tree structured representation (through the offset). Just pick an object from where the document has to be reconstructed from and run through all the siblings and children. But as with the other approaches whitespace outside content is lost due to the transformation of the XML document to the object representation.

This optimized object approach is hard to perform concurrent operations on since the locking has to occur on the object representing the whole document; unless there should be build some extra concurrency control into the *lw_objects* themselves, but this would be overkill. To when locking anything in this approach means at least locking the whole XML document.

2.2 XML-Store

XML-Store is as stated above based on value-oriented programming and the Document Value Model (DVM), which is a model for representing XML documents and methods for creating, accessing, modifying and persisting documents. Within the DVM documents are represented as trees, more specific directed acyclic graphs with nodes and character data.

The structure “element with child elements” is the primary data type representation, which contains the XML elements and data. Creating new elements can be done by adding the new element to the tree or by reusing existing substructures of the tree, i.e. when a modification to the tree structure is needed, creating new nodes or reuse of existing nodes, attributes or child-nodes does this. There are no destructive update operations within the DVM; the idea is that operations

like deletion are done through garbage collection when there are no references pointing to the specific node. This also means that concurrency control is superfluous, since nodes can not be deleted, an updated node is stored as a new node, and the original will still be available as long as threads are accessing it. Of course there would be problems if two threads change the same node at the same time; if this occurs there would have to be some sort of tree merge operation, which could merge the changed nodes into one.

Within XMLStore it is possible that elements can be shared within the same document or between documents making it possible to have multiple documents representing a XML tree structure or use a peer to peer network distribution to store data.

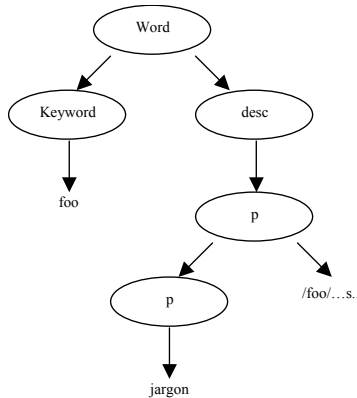


Figure 5: DVM representation of a XML document

XMLStore is a storage manager that implements the DVM and handles persistence and distribution. It has a layered architecture that consists of the DVM layer and a disk layer plus a name-server and a reference server. The disk layer handles loading and saving from physical disk and also handles the reference server, which is used when loading and saving. The reference server is basically a hash table that allow key to data mappings .The DVM layer handles the name-server, which is used for e.g. looking up symbolic names.

Documents are accessed through lazy loading. Loading a document means loading the root element and only if child elements are accessed through the API these are looked up in the reference server and then loaded into memory. When more elements and their data are loaded, memory usage will increase. The DVM module ensures that memory usage is limited through a FIFO list, which acts like a node proxy. But when an element is loaded all data contained in that specific element is also loaded. If the element consists of many child-element references or if it contains large amount of data such as text for an article, this also influences memory usage.

Each element within an XML document is saved as a separate value. This value is represented as a reference with a representation of the type of the node (<1> for elements, <2> for character data). Elements are stored together with value reference to its children. The character data is saved without additional information. An example of a subset of the data representation of Figure 2 can be found in the Table 15.

Reference	Value
R1	<1>persons <r2>...
R2	<1>person <<2>cpri=1> <r3><r5><r7>...
R3	<1>name <r4>
R4	<2>P1
R5	<1>address <r6>
R6	<2>A1
R7	<1>hobby <r8>
...	...

Table 15: Representation values in XML-store

Saving documents is done in a postorder traversal by saving one node at a time. Postorder traversal means that child nodes are accessed before the parent node. This is done to secure that the child nodes has a reference before the parent node is written. With the example in Figure 5 this method is represented in Figure 6 below:

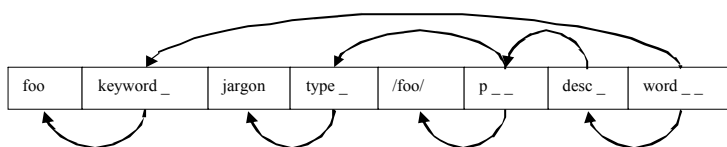


Figure 6: Sequence of value stream as they are saved to disk storage

When retrieving data from the XMLStore a reference to the specific data is needed, this reference can be retrieved from the name server. In the name server a mapping from symbolic name to reference is stored. The reference is used to load the data from the XMLStore, this is done by finding the offset on the disk where the data is stored; this offset is stored in the reference server. The XMLStore returns a new DVMNode containing the name, attributes and references to children. The subtree of the DVMNode can then be traversed through the API, and every time a node is accessed, which has not been accessed before, it will be loaded by the XMLStore. This method implies lazy loading.

2.2.1 XML-Store compared to the different approaches

When comparing XMLStore to the Text Approach there are not many areas that they have in common. Both approaches have the ability to load the XML document into a document-model representation and traverse the parsed data like a rooted DAG structure through a given API. But with the introduction of indices in the Text Approach (referred to as Improved Text Approach) there are some similarities. When looking at the representation of $((ref, tag) \text{ to } child_ref)$ index it looks like the disk representation of XMLStore but with the difference that the values (attributes, PCDATA, etc) are represented directly into the XMLStore data representation which makes access to values faster. Extra indices could off course also be build within the Improved Text Approach to speed up searches for values, but this will take up additional space and when data needs to be updated or inserted this extra index has to be maintained. The Improved Text Approach facilitates a *lazy loading* feature through the indices, where the references to the specific element values are represented. If data is not directly represented through the index structure the specific element can be access and data can be loaded. XMLStore also supports the lazy loading feature, but the difference between this and the Improved Text Approach is that XMLStore does not need the extra indices and when the specific element has been located the data is directly connected to it. Both approaches need to traverse the total tree representation when a query asks for a specific element because there is no mapping of the XML data structure, i.e. like a schema to optimize queries.

The Relational DTD Approach (referred to as DTD) is based on a relational database implementation (actually a DB2 implementation). The data representation and the way of accessing data are very different from the XMLStore. A drawback of the DTD is that it is depending on a higher level of information regarding the structure of data (through a schema definition) to be able to define table relations. Querying data in the DTD is done via joining tables through the well known table relations, where as in the XMLStore this is done through tree traversal where the structure is unknown. It is also possible to cluster data within the database tables to enhance queries even further. One of the advantages with the DTD is the minimizing of space through the structuring of tags into database tables where as the XMLStore reduces space via reusing elements or subtrees of elements, although this reduction depends on the content of the XML document. Further more the DTD has the advantage of being based on a well defined database architecture which includes various components such as portability and scalability, where as the XMLStore is in its early years regarding component development.

The Edge Approach consists of a single database table containing all data. Due to the nature of a database table it is possible to cluster the table on one of the columns, but this will only enhance certain simple queries. When making queries finding groups of elements is simple if the table is clustered accordingly, but when the query becomes more complex it is necessary to make selfjoins taking up additional storage and making querying more complex. When compared to the XMLStore this approach needs many enhancements to be able to make simple traversals through the DAG of the represented XML documents, making this approach less attractive to store XML data in, whereas it is a native feature of the XMLStore. Again this approach can be enhanced by the introduction of indices which makes it use additional storage.

As the Attribute Approach is a horizontal version of the edge approach this approach has inherited some of the downsides, i.e. the parent-child relations between XML elements are captured by join operations. Depending on the level of information needed these join operations can result in very complex queries making it difficult for the database query optimizer to produce a correct plan. As in the DTD this approach needs a schema to decide which tables that may contain sub-elements. The number of SQL queries needed to find all sub-elements equals the number of possible tags, where as in XPath used by XMLStore it is one query that traverses the tree. Again, as in the Improved Text Approach and the DTD, indices can improve performance on queries in the database.

When comparing the XMLStore to the Object Approach they both use objects as building stones. In both approaches building object for each node that exists is high priced in space consumption. The Object Approach has optimized the space consumption by building lightweight objects of several XML elements, hereby minimizing the number of normal objects with greater overhead. Compared to this, XMLStore uses the DVM that reuses subtrees when the subtree exists more than once in the DAG. This is done through a hash representation of the elements making sure that subtrees are equal. For reference to child elements the Object Approach uses a built-in reference within the object to be able to traverse through the tree structure. At present moment the XMLStore it is not possible to utilize the parent axis of the XPath standard. It is very demanding to add a new child element within the object approach (depending of the order the element has to be entered in sibling representation) due to the many reference pointers that needs to be updated, where as in the XMLStore a new node is added and the reference pointer is updated within the parent node. When accessing data the object approach needs to lock data so the node that is currently in use can not be updated from other applications. One way this could be implemented is through the use of a semaphore.

It is not only the database implementations that can be distributed; XMLStore supports peer-2-peer network solutions. XMLStore does not include a facility of handling mutable updates when two or more threads update the same subset of data⁵; the idea is to let the threads work on the node in their own copies and when data is updated the two documents could be merged together by some sort of three merging algorithm. Compared to the traditional database approach, databases need a strategy to lock the database elements if they are in use by another thread.

⁵ Henning Niss, Professor at ITU, Copenhagen; has informed that XMLStore does not contain the possibility at present moment to use tree merging

3 Performance evaluation

In the previous section we made a theoretical evaluation of the XMLStore, in this section we will make an empirical evaluation of the XMLStore. The empirical data presented in this section is collected from experimental tests on storing and querying data in the XMLStore. To evaluate how well the XMLStore performs we will compare it to the same tests performed on other XML storage implementations; more specifically those tested in [BG03], namely the Tamino Native XML database and Oracles XML enabled RDBMS. As a benchmark we have chosen to use standard XML querying components used on XML stored in flat files, since it is the simplest approach to store and query XML documents.

The foundation for the testing can be found in section **3.1 Experimental Testing**. The test environment and possible sources of errors are discussed in section **3.2 Physical and Logical confines**. The tests we perform consists of storing XML data and queries executed on the data, this is described in section **3.3 Database**. The benchmark tests can be found in **3.4 Standard XML querying components**. The tests of the XMLStore can be found in **3.6 XMLStore**.

3.1 Experimental Testing

Since the results of the performance tests carried out in this section should be comparable with the results of the black-box tests found in [GB03], we will use these tests as a baseline for our tests. If the results of the tests performed on the XMLStore indicate that performance improvements can be made, we will use profiling to gain further knowledge of where to optimize. An overview of the tests performed on the different storage systems can be found in Table 16.

		XMLStore	Text	Tamino	Oracle
Black-box	Storing	•		•	•
	Querying	•	•	•	•
Profiling	Storing	•			
	Querying	•			

Table 16: Overall test Strategy.

As in [GB03] we will use XPath for querying the database. The primary purpose of XPath is to address parts of an XML document, and to navigating through the hierarchical structure of an XML document (XPath models an XML document as a tree of nodes). An XPath expression evaluation occurs with respect to a context node; in our tests this context node is the root of the XML document, from where we will select nodes or node-sets with a location path expression. XPath is sufficient in our case since we will not be updating the XML document but only retrieve data by queries, the queries we will use can be found in section 3.3.2.

When executing tests we will be measuring system resource consumption. This includes the time it takes to execute the test, and the memory consumed during the call and any eventual cache stored in memory after the call. When timing a query, we compute the delta-time from when the query is started until it returns with a result; this way we will also collect the time of threads waiting for IO operations. Unfortunately we will also time other processes running on the operating system; we consider these problems in section 3.2.1.

We will report how much memory the query consumes, and how much of this data has been cached for future reference by the tested query engine; the Java Virtual Machine presents some problems with regard to measuring memory consumption since the memory manager of the JVM is not easily controlled. These problems are also considered in section 3.2.1.

3.2 Physical and Logical confines

When measuring system resource consumption there is a possibility that measurements will differ depending on the implementations of the Java Virtual Machine (JVM) and the Operating Systems (OS) which the JVM is running on. Taking these differences into account, we believe is outside the scope of this project, since we are testing the database functionality of the XMLStore, and not trying to come up with a recommendation on which JVM and OS to use, to gain the best performance of the XMLStore⁶. We have chosen to use the Sun J2SE 1.4.2 Java platform for our testing, which we run on Windows XP SP1. The underlying hardware is an IBM T40p notebook, fitted with a 1600 Intel Pentium M processor, and a Toshiba hard disk model MK-1016GAP (HDD2152) with 13ms average seek time and a rotational speed of 4,200rpm.

3.2.1 Possible Sources of Errors

There are various factors that can influence measurements in the tests. Among the external factors we find OS tasks taking up system resources during testing; and within the JVM we also find different tasks that can influence the results, such as garbage collection and memory allocation. These tasks influence our timing test-results because we only collect the delta from when we start the test until it completes, and not the actual time the thread is using the CPU.

The Operating System that the JVM is running on might influence the test if the JVM loses CPU time during execution of the test, while another process is running. In XP we have the options of setting the thread priority to `Time_Critical` for the JVM which should minimize interruptions by other processes and threads [SGG02]. Also the process of the current program selected in the graphical user interface receives three times as much CPU time as other processes. This will minimize interruptions by system processes taking over the CPU but it will not eliminate them.

The JVM has implemented a garbage collection algorithm that automatically removes objects within the JVM that is not referenced by any other objects. This algorithm is executed in a separate thread within the JVM with very low priority; so conceptually it should not get as much CPU time as the thread executing the test when we assign it the highest priority possible: `Thread.MAX_PRIORITY`. The scheduling of threads in the JVM is controlled by the OS, so in the end it is up to XP to assign CPU time to the threads accordingly to the assigned priority. Another way we try to make sure that a thread and/or process does not interfere too much with the result, is to run each test several times and use the average time; in our tests we perform 10 runs of every test.

Memory management of the JVM can also influence measurements. If there is not enough memory allocated to the heap to store all live objects, the JVM can allocate more memory dynamically; which will of course consume some system resources. But the problem can be overcome by supplying the JVM with parameters stating appropriate sizes of the initial and maximum size of the heap. The initial parameter allocates that amount of memory at start-up, and the maximum parameter limits the maximum amount of allocated memory [LY99]. This way we can make sure the task doing the allocation of memory does not run (or at least does not allocate memory) during testing. To counter this we have chosen the following setup of the JVM heap: maximum memory (`-Xmx`)= 600MB, initially allocated memory (`-Xms`)= 600MB⁷.

⁶ Of course we believe that if the XMLStore should be used as a widespread distributed storage facility, it should be possible to get a good reliable performance on various implementations of the underlying JVM and OS, but again it is outside the scope of this project to make such tests and optimizations.

⁷ We have chosen these values since there are 600MB of free memory when the OS is loaded.

When we are measuring the memory consumption of each test, the JVM presents some problems since we do not have the total control over the garbage collector (GC). If we want to run the GC we can only ask it to run, but it is not required to run; therefore we will pause the thread we are executing the tests from right after asking the GC to execute; this way our higher priority thread will not keep the GC from running. Also the GC can run at any time it gets the opportunity from the OS, which means that the memory used after a test might already have been exposed to some garbage collection and therefore we can not be totally sure that the memory results are accurate.

3.3 Database

In [GB03] four different types of databases based were used on XML documents generated by an XML test-database generator found at [XBENCH]. Combination of two factors, namely content composition and storage are used to create the test-databases, an overview these can be found in Table 17. The database is stored in either a single document or in multiple documents, and its composition is either text-centric or data-centric.

Database structure	Example of use.
Text-Centric Single Document (TCSD)	Dictionary.
Text-Centric Multiple Documents (TCMD)	Articles.
Data-Centric Single Document (DCSD)	Catalogue.
Data-Centric Multiple Documents (DCMD)	Transactional data; used for data exchange.

Table 17: Databases created by XBench.

The features of the dictionary test-database are one text-dominated XML document with repeated similar entries and deep nesting, containing unique data. The article test-database consists of multiple relatively small text-centric XML documents, with sizes ranging from several kilobytes to several hundred kilobytes. The catalogue test-database is similar to the TCSD, in terms of structure but with less text content; also the structure of this database is more strict when it comes to conforming to a schema, in the sense that there exists less irregularity in DCSD than in TCSD (since the XML documents in DCSD are normally translated directly from relations [XBENCH]). The features of the transactional test-database consists of multiple XML documents composed of more descriptive tags and less text content; the structure is text and restricted by schemas.

It is very time-consuming to perform our tests because we have to clear the OS cache between each test, therefore we have chosen to limit ourselves to just one database type, namely the TCSD database; in the following we will briefly justify this decision. Using XPath on multiple documents make no sense since XPath can only work on one context node as described in section 3.1. For example executing a query like `"/doc1/tag1[@id=doc2/tag2@ref]"` where doc1 and doc2 are different documents, would require the query engine to split the query up into two queries like this: `result1="/doc2/tag2@ref"` and `result2="/doc1/tag1[@id=result1]"`. But if the two documents were assembled into one document under a tag docs, then the query could be written in one XPath expression like this: `"/docs/doc1/tag1[@id=/docs/doc2/tag2@ref]"`. Merging multiple documents into a single document would make it possible to query the database with XPath, but it would be no different from just querying a single document database, which is why we limit our tests to single document databases.

With the exclusion of multiple document databases we will look at the differences between the data-centric and text-centric databases created by XBench. As described above there is no difference in structure, but the data-centric database contains less content, and the database is stricter when it comes to conforming to a schema. The DCSD and TCSD databases generated by XBench are very similar, with the only difference of being less irregular in the data-centric version. We have chosen the text-centric approach because it contains both the structure of data-centric

databases and two tags containing text-centric data, which is what we believe to be one of the forces of XML documents, contrary to the data-centric databases normally generated from the RDBMS tables.

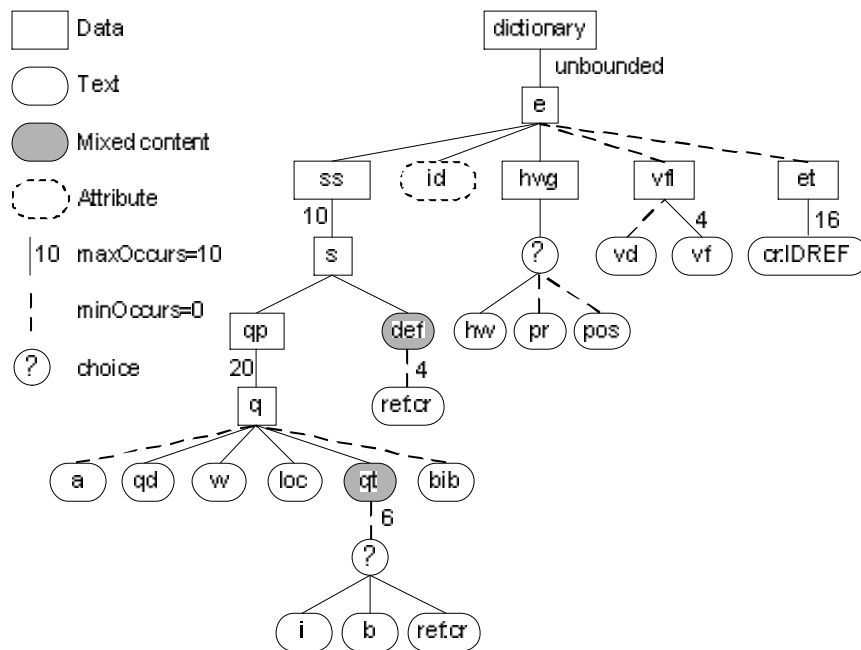


Figure 7: Structure of the TCSD test database.

The structure of the TCSD database we are testing is illustrated in Figure 7. This structure is not a one to one mapping of the schema from which the data in the database is created; the whole schema can be found in Appendix A. The unbounded maxOccurs assignment to the e-tag makes this structure very wide at the root; and with focus on the q-tag and its children, since the maxOccurs equals 10 for the s-tag and 20 for the q-tag. Since the database created is an article database the qt-tag containing the article body is by far the most space consuming, being a child of the frequently occurring q-tag.

3.3.1 Storing data

We will test storing data, by storing a complete XML database at once and then see how much system resources were consumed during and after (in the form of space consumption on disk). The text approach will not be tested for storing data since this approach loads and parses the XML document into memory directly from the raw XML text file.

3.3.2 Queries

When testing the different approaches we are using the queries defined in [GB03], so it is possible to compare our results with theirs. If we find that some extra queries are needed to show some specific characteristic not tested by their queries, we will introduce these new queries when that situation occurs. The queries we will use for our tests can be found in Table 18; in the next sections we will refer to them by their number.

Nr.	XPath	Description
1	/	retrieve the root node.
2	/e/hwg[hw="short"]	match low nested element.
3	/e/ss/s/qp/q/qt[i="long"]	match deep nested element.
4	/e[@id="beginning"]	match element in the beginning of the document.
5	/e[@id="middle"]	match element in the middle of the document.
6	/e[@id="end"]	match element at the end of the document.
7	/e/ss/s/qp/*/qt[i="one_unknown"]	match expression with one unknown element.
8	/e/ss//qt[i="many_unknown"]	match expression with many unknown elements.
9	//qt[i="anywhere"]	match expression with unknown path.

Table 18: XPath expressions used for testing.

The queries we are using, can be divided into three groups, the first group (1,2,3) retrieves data from various depths of the XML tree representation, the second group (4,5,6) retrieves data at the same depth but at various positions of the document (e.g. beginning, middle, end), and the third group (7,8,9) retrieves data with location paths that contain one or more unknown elements.

3.4 Standard XML querying components

XML querying components that can be used 'of the shelf' on XML stored in text files has been developed for many programming languages. Because of the components availability and ease of use, we believe many application developers tend to employ these instead of commercial storage solutions, take for example the XML handling framework made available by the Apache Group, free of charge.

Testing standard XML querying components, just because of their widespread use seems like a good idea, but more importantly it provides a baseline to compare with the more complex storage solutions. The interesting question is: why use a complex system, if the simple text file approach performs just as well?

We will use the Xalan for Java framework developed by The Apache Software Foundation [APACHE] for our tests in this section. When comparing test results we have to be sure what we are comparing; when we want to compare the XML data representation of the Apache framework, which is DOM, with the DVM representation of the XMLStore, we have to be sure that the query engine does not influence the results. Therefore we use the query engine of [TA03], since it works on both DVM and DOM data representations. But when testing the Apache framework as a whole we will use the query engine shipped with the framework.

We will implement simple 'cold started' program, where we load the XML file into a memory resident data representation, perform a query, and then shut down the program. We will also implement a server version, which caches XML data previously parsed, by keeping the data representation in memory, so subsequent queries to the same file can be processed without the loading and parsing otherwise required.

3.4.1 Cold Start

The cold started approach is much like the basic text approach without indices outlined in 2.1.1; we load and parse the whole XML document into a memory resident DOM⁸ tree every time we need to query it, and when querying is done we unload the DOM tree again. No indexes are used in this approach.

⁸ The DOM tree is the Apache implementation of a Document Object Model.

3.4.2 Server

The server approach works as the cold start, except that it does not unload the memory resident DOM tree when querying is finished, instead a reference to the tree is stored in memory and the next time the same XML document is queried the memory resident DOM tree is queried again, this saves the loading and parsing of the XML document, but on the other hand the approach requires more memory, if several documents is loaded at the same time. We have not implemented the second approach of 2.1.1, the text approach with indices, so we will be loading the whole document into memory, instead of using indices to decide which parts of the document to load into memory.

3.4.3 Querying

We will first test the two approaches and see how big the difference is between the two, when one has to load and parse the document between every query and the other does not. In this test we only retrieve the node of the query for further processing, and not the XML serialized form of the query. The test results are shown in Table 19; for each approach there are three results per query, the milliseconds it took to execute the query, the MB in memory after the query has executed and the MB in memory after the garbage collector has been run.

Query nr.	Cold Start			Server		
	ms	MB	MB(gc)	ms	MB	MB(gc)
1	3639	80	14	1	90	75
2	3444	80	14	3	78	75
3	3469	80	14	2	78	75
4	3451	80	14	1	78	75
5	3477	80	14	1	78	75
6	3451	80	14	2	78	75
7	3474	80	14	3	78	75
8	5989	338	14	2031	336	97
9	3475	81	14	1	99	97

Table 19: Executing XPath queries from Table 18 on text files, returning nodes.

The results in Table 19 clearly show the difference between the cold start and the server approaches, in the cold start approach there is an overhead for parsing the document every time, in the server approach this overhead is eliminated because the document has already been loaded into memory. In Table 19 query 8 is interesting since it executes in 2 seconds in the server approach where the other queries only takes about 2 milliseconds to execute. The query has a “broken” path with many unknown elements from the root node to a deeply nested element. If we look at the memory usage, it indicates that some further information is loaded into memory. If this is the case then there should be some sort of index that is sufficient to answer the other queries, but not query 8. We need some additional information to draw any conclusions and to discover if we are right about the index, hence we ask for some more information from the queries. We now test how the approach reacts when it has to return the serialized result of the query. The results of this test can be found in Table 20.

Reconstructing the result node			
Query nr.	ms	MB	MB(gc)
1	1	90	75
2	3	78	75
3	8	79	75
4	4	80	75
5	670	140	107
6	1441	364	123
7	4	127	123
8	2383	377	97
9	10	103	97

Table 20: Executing XPath queries from Table 18 on text files, returning text.

The most obvious difference from the previous test is query 5 and 6, the queries retrieve a node in the middle and at the end of the document, but at the same depth, query 4 retrieves the same level as 5 and 6 but in the beginning of the document. The memory consumption also grows for query 5 and 6, and after the query terminates, the data stored in memory are garbage collected (i.e. it is not used as a cache for further querying).

Since query 8 has the same slow execution time in both tests, it indicates that the indices build are not sufficient to perform the query, this can be shown in another simple test: if we put an extra tag in the path of query 9 like this: `//q/qt[i="anywhere"]` the time to execute suddenly expands from 10ms to 2158ms, and the memory consumption from 103MB to 336MB. This indicates that since no index can be used, the whole XML document has to be searched to find the answer. When we look at the executions times of query 4, 5 and 6 it seems like the query engine searches from the beginning of the document to the end, since the execution time and memory consumption of the queries grows the further we get from the beginning of the document; query 8 has the longest execution time because the unknown tags forces the query engine to search the whole document, when no indices can be used.

3.5 Tamino and Oracle two Commercial Systems

Two XML storage systems has been tested and evaluated in [GB03]. The Oracle v.9.2 XML enabled relational database and the Tamino v.4.1 Native XML database. We will briefly describe the databases here and then use the tests results from [GB03] for comparisons in section 3.6 XMLStore. The physical confines of the tests performed in [GB03], was a machine with 1GH CPU, 384MB ram and 20 GB hard disk. The tests were run on Microsoft Windows 2000 Server.

3.5.1 Tamino v. 4.1 DB

The Tamino Native XML Database uses two components to handle data. One is *Data Map*, the second is *Native XML Data Store*. The *Data Map* contains maps based on schemas represented as a B+ tree, and additional information about where the data is stored; this means that data can be located in different stores. Through the use of *Data Map* it is possible to query data stored in multiple documents or other data sources.

Tamino uses DOM to access data. The DOM representation is stored binary on disk, and large files are compressed. Tamino supports round-tripping which makes it possible to restore the XML document exactly as it was stored. Data is stored in collections that can be used to limit documents searched when performing queries, by specifying a specific collection. Indices over the data are not created automatically.

3.5.2 Oracle v.9.2 DB

Oracle introduces the XMLType which is an abstraction because data can be represented in different ways. One way is to store XML documents in a Character Large Object (CLOB); another is to map XML documents directly to objects in automatically generated tables. If an XML document is saved as a CLOB round-tripping is possible. If data is represented through the object approach whitespaces between elements are lost. The XMLTypes can be accessed through Oracle's SQL statements.

If the XMLType is CLOB then Oracle saves data as raw text. Additionally if the XML document is to be evaluated Oracle needs to parse the content through DOM. To enhance the queries it is possible to create indices but these can only be used for selection of documents. If data from an XML document is needed the entire document has to be parsed. To update data where data is based on CLOB is inefficient. The CLOB has to be parsed and the update has to be performed in DOM. Finally the DOM tree needs to be written to the CLOB as an XML document. This approach is similar to the text approach described above.

When the object approach is chosen an XML schema is needed. Oracle automatically creates tables and objects from the schema definition. Accessing data is done through lazy loading. Oracle v. 9.2 uses more disk space than the original files to store XML documents, and documents above 4 kB have to be imported as a CLOB before they can be related/changed to XMLType columns and tables.

Oracle v. 9.2 uses XPath, but only within a limited scope, because of this some of the queries have not been tested due to errors. Important to note is that schema based documents perform slower than documents without schemas. [GB03] has no answer to why this is so, but a suggestion is that Oracle validates the documents that have been created.

Conclusion is that Oracle v.9.2 performs at its best when dataset is CLOB. Storing and querying documents larger than 10 MB results in unacceptable bad performance [GB03].

3.6 XMLStore

In this section we will test the XMLStore and compare the results with those of [GB03] and the tests from section 3.4 Standard XML querying components. But first we will describe the XMLStores decorators that we will be testing in this section. Then we will test storing followed by querying. At the end of this section we will perform some profiling of the querying of the XMLStore, and we will show what optimizations could be implemented, and at last show a comparison of the running times of the compared XMLStore with the "old" version.

3.6.1 XMLStore Decorators

The XMLStore is dynamically assembled from various components like a Decorator pattern⁹, which makes the XMLStore highly configurable; we will not test all configurations of decorators that apply to the XMLStore, but only those that make sense in our setup. Since we do not test the distributed features of the XMLStore, the decorators implementing these features will not be tested.

Each of the pluggable XMLStore modules can work alone, or be assembled in a number of ways. The module at the lowest level has direct access to the XMLStore disk implementation, and the modules in between forwards requests from the module above or to the module below and returns the result. The individual module can modify the request or the result, when it has the control.

⁹ The decorator pattern makes it possible to add responsibilities to individual objects dynamically and transparently; it is further described in [GHJV94].

Unlike the decorator pattern, an XMLStore module can decide not to forward the request, if the module has enough information to respond to the request.

3.6.1.1 Caching (Read/Write)

The caching module stores already retrieved (value, value reference) pairs in a local cache, and return the value of requested value references already stored, instead of reading it again from disk (or lower modules). The read module stores previously read values, and the write module stores values previously written to disk, for later retrieval.

The read cache module can be very useful when processing queries, since values will only be read once from disk, and therefore IO and parsing on-disk representation to memory representation will only occur once. The downside is if the value is only used this once by the query processor, it will just take up space in the memory, and when memory is sparse the cache has to be deleted again.

Our test can also use the write cache module, by utilizing that after we have written the whole XML document to disk, we will also have a memory resident representation, and therefore IO will be saved when querying.

3.6.1.2 Buffering (Write)

The write buffer stores write requests until enough is stored for efficient execution of the request. If a read requests a value not yet stored in disk, it is collected from the buffer and returned. No read buffer has been implemented yet.

When we test how fast we can store a XML document on disk, this module will be tested as well; although we expect it only to be effective over a network.

3.6.1.3 Asynchronous Requests (Optimistic Read/Write)

Asynchronous requests implemented at this moment are the Optimistic Read and Write modules. The optimistic read module returns a locked shared variable immediately when it receives a request, and another thread is started up and processes the read request. When the value is retrieved the module saves the result in the variable and unlocks it. If the requesting thread tries to access the value before it has been stored in the variable, it will be blocked and forced to wait until the variable is unlocked again. But if the requesting thread does not access the value it can continue using the CPU, instead of waiting for the IO to complete. The optimistic write module works in a similar way, only it returns a locked value reference to the saved value.

This is a useful feature for the query processor, because it can perform other tasks while the requested node is retrieved, not having to wait actively for the IO operations to complete. Of course if the query processor has to use the value of the requested node, in its computations it will have to wait for the XMLStore to return the value before continuing. This module should conceptually improve performance of queries.

The optimistic write could improve our performance tests, since we need to parse an XML file in order to save it in the XMLStore, and while waiting for the IO to perform there might be time for the CPU to parse some more of the file; although the overhead of changing threads may take longer than waiting for the IO operation to finish. If however the values should be stored on a network node, then the idle time would probably be greater compared to disk IO.

3.6.2 Storing data

In the XMLStore there are implemented three different approaches for storing data, with very different memory consumption. The first method, visualized in Figure 8, shows how the XML file is parsed with SAX into an in-memory DOM representation, which is again transformed into an in-memory DVM representation, which is then saved to disk.

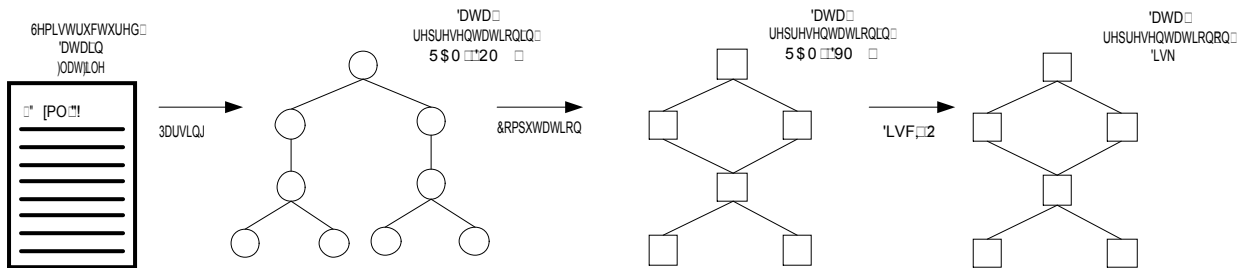


Figure 8: Storing data using DOM.

Of course this procedure is very memory consuming, and why not leave out one of the in-memory data representations, namely the DOM-representation. This is done in the second procedure, where the XML file is parsed into an in-memory DVM representation, which is then stored on disk, this is shown in Figure 9.

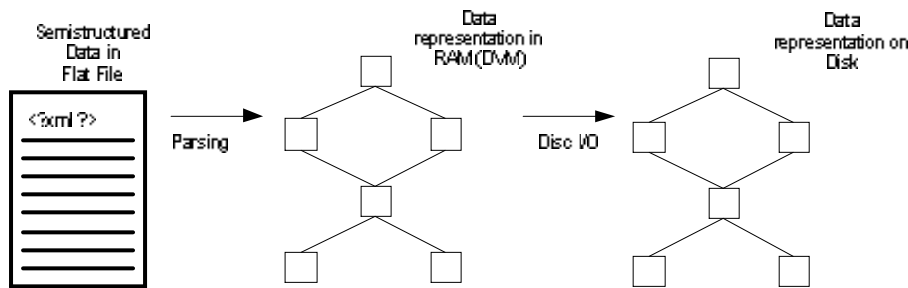


Figure 9: Storing data using in-memory DVM representation.

But again why keep the data representation in memory at all, when it is to be saved on disk anyway. Well this is done in the third procedure, where the XML file is saved to disk, in a DVM representation while it is parsed with the SAX parser; this is shown in Figure 10.

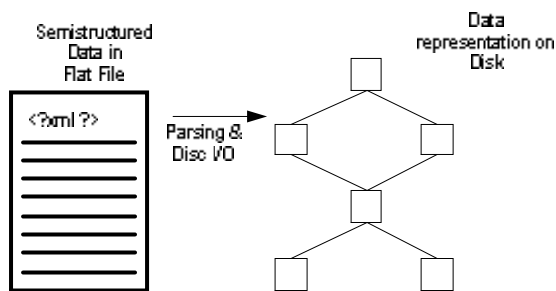


Figure 10: Parsing data directly to disk

An argument to keep the parsed XML file in memory is if the data should be pre-processed immediately after the parsing. But this can be done directly in the XMLStore, by using a WriteCacheXMLStore which stores the whole file in-memory (of course this does not utilize the lazy loading feature of the XMLStore). Timing and memory consumption of the storage of XML data in the XMLStore, using the approaches described above can be found in Table 21.

XMLStore type	ms	MB	MB (GC)
RawXMLStore (DOM)	16875	305	53
RawXMLStore (SAX in Memory DVM)	15662	203	43
RawXMLStore (SAX to disk)	14441	76	40

Table 21: Storing 10 MB XML TCSD data.

As the results tells us, the execution time only varies by one second, but the memory consumption on the other hand varies from 76MB to 305MB, where of course the direct SAX parsing to disk is the best solution, we will use this parsing method when testing the different ‘write’ decorators of the XMLStore.

Now we will test how the different decorators of the XMLStore perform when saving data to disk. As discussed in section 3.6.1 we will test write-buffering XMLStore as well as the optimistic-write XMLStore. The caching XMLStore will also be tested, but this is only to see how large the overhead is for caching which should be used in subsequent reads. The results of the test can be found in Table 22.

XMLStore type	ms	MB	MB (GC)
OptimisticWriteXMLStore	76209	83	40
WriteCacheXMLStore	18762	243	62
WriteBufferXMLStore (buffer at 32768)	14423	76	40

Table 22: Storing 10 MB XML TCSD data.

As we can see the OptimisticWriteXMLStore has much worse running time when storing documents; we had anticipated that the performance would be improved by this approach. The bad running time of this decorator might be because we are parsing and writing the whole document at once. This means that there are created many threads before each IO is completed, this causes the CPU to spend a lot of its time on creating and switching between threads. When comparing the WriteBufferXMLStore with the RawXMLStore there are no difference, so as we suspected the improvement is probably only when storing data over a network. The WriteCacheXMLStore does not as we anticipated keep the tree-representation in memory after the document as been stored, as we can see from the garbage collected memory. Although the memory used to store the XML document indicates that the tree-representation has been stored in memory at some point while storing the document. This is probably because there is a limit on how much data is kept in memory.

We can not compare the storing of data in the XMLStore with the text approach, since it uses the original document; but we can compare it to the Tamino and Oracle databases, a comparison is shown in Table 23.

Database – Document Size - Schema	Space on disk	Time
Tamino - 10MB TCSD - with schema	4,63MB	66sec
Tamino - 10MB TCSD - without schema	20,17MB	13hours
Oracle - 10MB TCSD - with schema	12,95MB	407sec
Oracle - 10MB TCSD - without schema	12,95MB	403sec
RawXMLStore - 10MB TCSD - without schema	24,54MB	14,4sec
RawXMLStore - 100MB TCSD - without schema	233,32MB	242sec

Table 23: Comparison with Tamino and Oracle.

When comparing the storing of data with Tamino and Oracle it is clear that the XMLStore performs best when it comes to the time it takes to store documents. But this is also very fair, because both of the other approaches are using DOM, and Oracle are transforming the data several times when it has to store more than 4 kB. Also when the Tamino has to store the database without a schema, it uses 13 hours, which leads us to believe that it is doing some sort of data-mining to

generate its own schema from the data, to store in the *DataMap*. It also has to be said that the setup in [GB03] only used 384 MB of memory where we used 600MB for the JVM alone, this could mean that the whole XML database could not be stored in memory when using DOM as Tamino and Oracle do, and therefore the swapping to hard disk is what takes the extra time.

In the space on disk column we are adding the space consumed by the reference server as well as the disk, as the Tamino and Oracle space consumption is adding the data itself and any schemas generated while storing the data. The XMLStore uses somewhat more than the Tamino and Oracle databases; this is probably because the value references for all the elements are stored in the reference server and in the disk. The test database we are using are also not utilizing all of the potential of the XMLStore, since all nodes in the test database is unique. If some of the nodes had been duplicates, they would only occur once because of the DVM model, and then the XMLStore would consume less space on disk.

3.6.3 Querying Data

When querying data in the XMLStore we first run all the queries from Table 18 consecutively on the different XMLStore decorators, this is done only to see how the XMLStore performs on the different queries. It could be discussed that the test of the ReadCacheXMLStore decorator would give different results depending on the execution order of the queries, but again this test is only to give an overview; later we will go into more details. The test results can be found in Table 24.

Query nr.	RawXMLStore			OptimisticReadXMLStore			ReadCacheXMLStore		
	ms	MB	MB(gc)	ms	MB	MB(gc)	ms	MB	MB(gc)
1	0	40	40	0	40	40	0	40	40
2	234	56	40	235	56	40	244	56	44
3	13828	238	40	13856	238	40	13716	233	203
4	40	48	40	41	48	40	0	212	203
5	40	44	40	42	44	40	0	204	203
6	41	44	40	41	44	40	0	204	203
7	13772	232	40	13798	232	40	576	223	209
8	22254	314	40	22218	314	40	8467	334	255
9	22999	314	40	22974	314	40	3014	340	258

Table 24: Executing XPath queries from Table 18 on different XMLStores.

The RawXMLStore is the most basic test, it reloads all data between queries and from this we can see how the lazy loading is working, in the queries with shallow depth (1,4,5,6) the memory consumption is not very big, and in the queries with broken paths (7,8,9), where wildcards are inserted most of the XML structure is loaded into memory. The interesting query is nr 3, where it seems like most of the structure is loaded into memory; this is because of the structure of the XML which can be seen in Figure 7. Here it is shown how the tag referenced by query 3 is one of the most frequently occurring tags, and there is no specific selection in the query that restricts the search, so query 3 has to traverse all the qt-tags to find the one containing the element `<i>long</i>`.

The OptimisticReadXMLStore is not as optimistic as we thought it would be, in fact it performs pretty much the same as the RawXMLStore. This might be because the query engine tries to access the values right away and is then locked until the started thread returns with the value, but we can not tell for sure if this is the fact.

The ReadCacheXMLStore has better performance in query 7,8 and 9 than the other two stores, this is due to the caching done in previous queries, more specifically query 3 where a large amount of the tree structure is loaded into the cache since the query potentially matches most of the tree, since the most specific part of the query matches the leaf of one of the most frequently occurring tags, all other qt-tags has to be searched as well.

The overall results from the first tests are surprisingly bad compared to the text approach, but looking at the results of the ReadCacheXMLStore, we see that query 9 performs a lot better than the two other stores; this is because most of the data is loaded into memory at the time of execution as we can see in the memory after garbage collection column for query 8. If we test the queries with the whole XML document loaded into memory we will be able to tell if it is the loading of data into memory or querying the data that should be optimized; the results are shown in Table 25.

Query nr.	ReadCacheXMLStore			Text-Server approach		
	Ms	MB	MB(gc)	ms	MB	MB(gc)
1	0	261	246	1	90	75
2	10	247	246	3	78	75
3	596	266	246	8	79	75
4	0	253	246	4	80	75
5	0	247	246	670	140	107
6	0	247	246	1441	364	123
7	589	266	246	4	127	123
8	2403	339	246	2383	377	97
9	2563	303	246	10	103	97
[10] //q/qt[i="\`anywhere\`"]	2870	323	246	2099	344	97

Table 25: Querying XMLStore and Text with all data cached in memory.

When we compare the ReadCacheXMLStore with the text-server approach from section 0, we see that the timings are not that much different on some of the queries and even better for the XMLStore on query 4, 5 and 6. Query 3 performs worse in the XMLStore because of the structure of the XML document which can be found in Figure 7; query 3 selects a path which matches most of the document since the qt-tag is one of the five most frequently occurring tags in the database. So even if the query selects a specific element, the path to the element is not specific enough and therefore no subtrees can be deselected by the query engine. If on the other hand a specific e-tag is selected like in: “/e[@id="E2"]/ss/s/qp/q/qt[i="long"]” it only takes 3ms to execute the query, this is because of the e-tag is unbounded which creates a wide structure at the root element (see Figure 7). The text approach has a better execution time on query 3 because it can use indices. Looking at the wildcard queries (7, 8 and 9, and the extra 10'th query), in 7 and 9 the text approach benefits on its indices, but in 8 and 10, where it can not use its indices the two approaches are pretty close (in query 10 the XMLStore is somewhat slower, but not with the same degree as with query 7 or 9).

3.6.4 Profiling and Optimization

What we learned from the tests of the XMLStore with and without the pre-cached XML database, was that the query engine performed ok compared to the text based approach, when it could not use its indices; this also suggests that the XMLStore could gain some performance executing queries by implementing some sort of indexing, this will be described in more detail in section 4.4. But when the XMLStore does not have pre-cached version of the data, it does not perform that good, and it seems like it depends on how many nodes has to be loaded to execute the query. To get a better idea of where the bottleneck lies, we will use JProfiler to profile the XMLStore while executing query 9; as before we are looking at memory and CPU consumption.

no	Method invocation or class initialization	Percent of total CPU
1	java.security.MessageDigest.getInstance	18,7%
2	java.security.MessageDigest.digest	16,6%
3	java.lang.String.<init>(byte[], int, int)	8,8%
4	java.lang.String.getBytes(java.lang.String)	6,0%
5	java.security.MessageDigest.update	5,4%
6	java.lang.String.<init>(byte[])	3,4%
7	java.util.Map.get	3,3%
8	org.planx.xmlstore.references.SimpleValueReference.compareTo	2,6%
9	org.planx.xmlstore.nodes.DVMImmNode.<init>	2,2%
10	java.util.Map.put	2,2%
11

Table 26: Profiling results; hotspots of CPU usage on query 9.

Table 26 shows us the hotspots of CPU usage while executing query 9 on a RawXMLStore; we are only showing the most time consuming method invocations; to minimize or eliminate some of these we will dig a little deeper into what happens when a node is loaded. We will do this because the tests performed on the RawXMLStore and the fully cached ReadCacheXMLStore suggest that the most time-consuming part is the loading of nodes, and the more nodes that is required by the query the more time it takes to execute. Actually the six most time-consuming tasks in Table 26 can be found in the method invocations from the call to load in the XMLStore and the return of a DVMImmNode, in fact four of the most time consuming tasks is executed in the initialization of the DVMImmNode. The five most time-consuming methods in Table 26 are depicted in the sequence diagram in Figure 11 (method invocations that is not time consuming has been left out of the diagram for clarity).

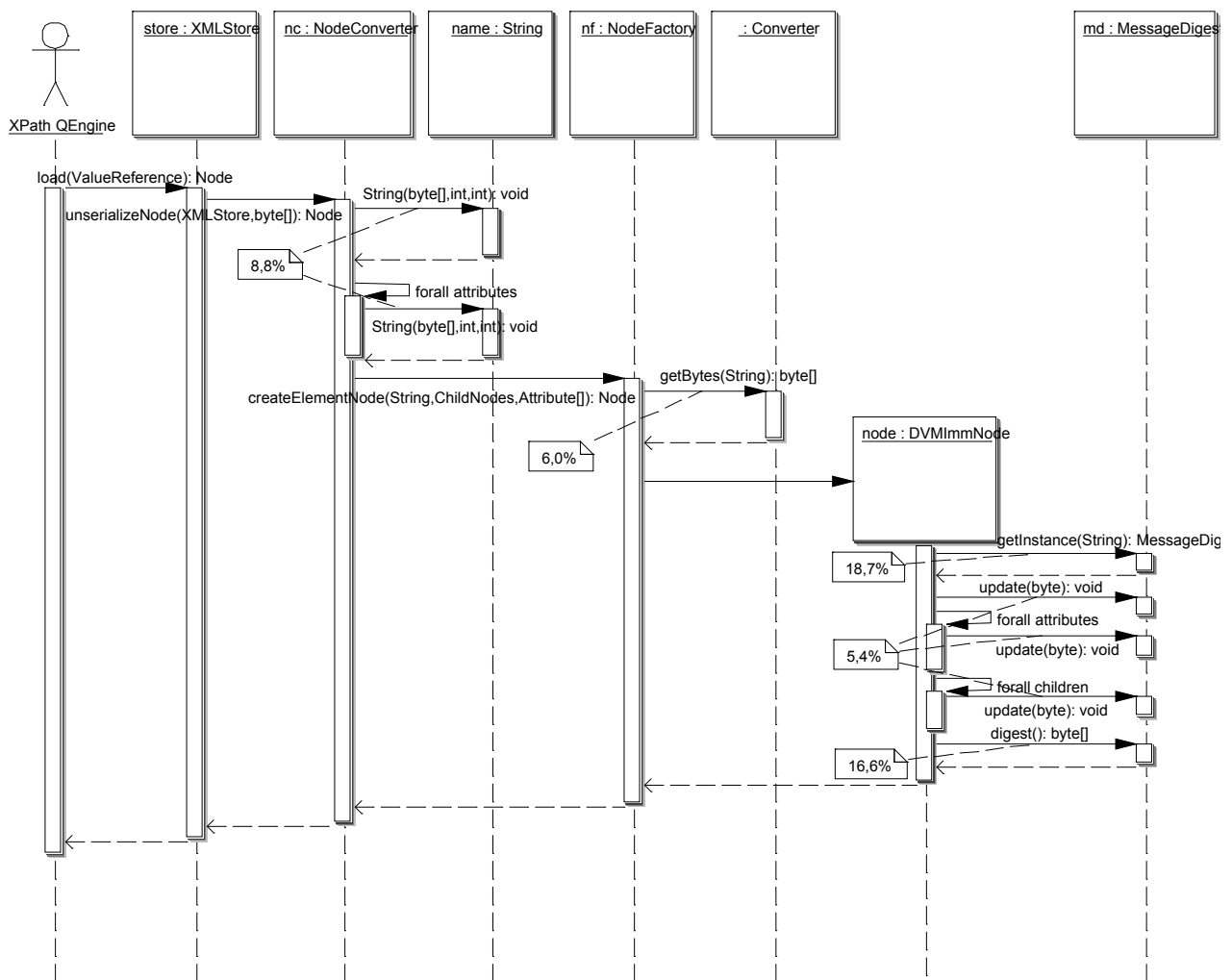


Figure 11: Sequence Diagram showing CPU consuming methods when loading a node from an XMLStore.

In the sequence diagram we can see that approximately 40,7% of the total query time is spent on creating a value reference for DVMImmNodes. The most time consuming method invocation is the call to the static factory method `getInstance(String digest)` of the `MessageDigest` class, which returns an instance of the `MessageDigest` object that implements the given digest algorithm. An easy performance improvement is to retrieve the `MessageDigest` object only once, and then store it in a static variable used by all subsequent calls to `DVMImmNode<init>`. This enhancement will also improve execution time when saving nodes to the `XMLStore`, where the value reference is also created. This leads to another optimization; because there is no reason why we should create the value reference that already has been created when saving the node and the value reference is even used to retrieve the node when calling the `load` method in the `XMLStore`. All we need to do is to pass the value reference as a parameter to the other calls down the chain, to the initialization of the `DVMImmNode`, where constructors are created with value references. This way we can save all the time spent on calls to the `MessageDigest` object, creating the value reference for the node again.

When looking at Table 26 there are some other time consuming operations not involving the `MessageDigest`; namely the methods involving conversions between `byte[]` and `Strings`, this conversion takes place when the `NodeConverter` unserializes a node. All attributes and the name of the node is converted from a `byte[]` received from the `XMLStore` to a `String`, which it passes on to the `NodeFactory`. The `NodeFactory` converts the name back to a `byte[]`, this conversion is

superfluous in both directions. The Attributes are stored in an object where the name and the value are Strings, but these could just as well be byte[], so the conversion could be saved. This way the byte[] will only be converted to a String when this is required by the application, and only the nodes that require this conversion is converted, as opposed to all nodes. In our test database there are very few attributes namely only in the highest elements; but in databases where there is a higher concentration of attributes, the savings will of course be higher. In Table 27 the optimizations described above is compared with the original XMLStore.

Query nr.	Original RawXMLStore			Optimized RawXMLStore		
	ms	MB	MB(gc)	Ms	MB	MB(gc)
1	0	40	40	0	40	40
2	234	56	40	80	45	40
3	13828	238	40	5829	158	40
4	40	48	40	11	42	40
5	40	44	40	10	42	40
6	41	44	40	11	42	40
7	13772	232	40	5927	159	40
8	22254	314	40	9449	237	40
9	22999	314	40	9863	244	40

Table 27: Comparing “old” and optimized RawXMLStore.

With the performance improvements implemented so far, the XMLStore executes the queries at least two times as fast as the original, and the memory consumption is reduced by 20-30% in most cases.

3.6.5 Comparing Optimized XMLStore with Other Implementations

In this section we will compare all tested xml storage implementations. The test results for queries executed on the optimized RawXMLStore, the ReadCacheXMLStore, the Cold Start text approach, the Server based text approach, the Tamino and the Oracle databases can be found in Table 28 (the ReadCacheXMLStore results are from the original test since the query engine has not been optimized, only the procedure loading nodes has been optimized, and all of the XML structure has been loaded into the ReadCacheXMLStore before performing any queries, so the optimizations will not show in the results anyway).

Query nr.	RawXMLStore	ReadCacheXMLStore	Text-ColdS	Text-Server	Tamino	Oracle
1	0 ms	0 ms	3639 ms	1 ms	na	15,5 sec
2	80 ms	10 ms	3444 ms	3 ms	3078 ms	4,2 sec
3	5829 ms	596 ms	3469 ms	8 ms	2750 ms	4,1 sec
4	11 ms	0 ms	3451 ms	4 ms	2594 ms	na
5	10 ms	0 ms	3477 ms	670 ms	2562 ms	na
6	11 ms	0 ms	3451 ms	1441 ms	2719 ms	na
7	5927 ms	589 ms	3474 ms	4 ms	2891 ms	na
8	9449 ms	2403 ms	5989 ms	2383 ms	3140 ms	na
9	9863 ms	2563 ms	3475 ms	10 ms	3172 ms	na

Table 28: Comparing Optimized XMLStore with Other Implementations.

Before commenting on the test we have to note that the tests performed on the Tamino and Oracle databases, only had 384MB of ram available (shared with OS), whereas the other tests had 600 alone. According to [GB03] this resulted in Java out-of-memory errors, this is indicated by the not available (na) in the table, since the query could not execute.

When performing low nested queries the XMLStore outperforms all other implementations, even the RawXMLStore is better than the non-XMLStore implementations. The Oracle database seems to load the whole document into cache in the first query, like the Text-Server, but compared to the

XMLServer where caching takes under 10 seconds, and any subsequent queries maximum takes 2,5 seconds, which is a lot faster than Oracles 4,1 second.

The Text-Server approach outperforms the ReadCacheXMLStore because of its indexes, except of course on the low nested elements at the end of the XML document. The ReadCacheXMLStore outperforms the Tamino database and the Text-ColdS approach on all queries. On all other than low nested elements the RawXMLStore has the worst results. One interesting thing to see is that Text-ColdS has almost the same running times as the Tamino database, this is probably because they do things pretty much the same way; by using a DOM.

4 Enhancements of the XMLStore

This section describes different enhancements to the XMLStore that conceptually should improve the performance of the XMLStores database functionality. Our description of these possible enhancements is superficial; and should only be viewed as ideas for possible new developments for the XMLStore within the Plan-X project.

4.1 Using Schemas to Optimise the Query Strategy

If the XML structure that is being queried is strictly conforming to a schema (and this schema is known) then it is possible to optimise the query processing, by intelligently resolving a strategy for removing subtrees from the search set that does not contain any results, according to the schema. For example, why load subtrees when performing queries with wildcards when the path after the wildcard does not exist in the subtree?

An example on such an optimization could be that we wish to search for persons within the tree structure of Figure 1. Through the use of the schema we could limit our search to objects within the XML document that contains the person element, here the “*persons*” and the “*child*” elements and thereby minimizing query time by not searching sub nodes of other elements.

To implement this solution the query engine should support a query optimizer that is able to read the schema. If a schema is not present the query engine should execute the query normally.

4.2 Optimize Loading

If an XPath query is expected to find something anywhere in the document with the wildcard ‘//’, it might improve performance if the xml-structure is loaded sequentially and all-at-once, instead of loading it in small pieces. This enhancement could be implemented to reduce seeks on the hard disk.

For this to work a query optimizer should be able to translate the query before executing it to see what is necessary to load. Depending on the tokens in the query, the optimizer could decide how the loading strategy should be, in the ‘//’ example the whole document should be loaded at once.

4.3 Reducing Objects with the Flyweight pattern

In XML documents with lots of small elements the overhead of creating objects for all elements is prohibitive. One way to avoid this overhead is to use the idea of the flyweight pattern described in [GHJV94].

The flyweight pattern is about minimizing storage for objects using sharing to support large numbers of fine-grained objects efficiently. The fine-grained objects could be related to the DVMNode of the XMLStore, which represent an element in the XML document, and if many elements exist this pattern could possibly minimize memory consumption [GHJV94].

Our opinion is that this pattern would have to be examined thoroughly, and considered if it is at all applicable to the XMLStore, but this has to be done in a separate project.

4.4 Index

Indices are not supported within the XMLStore, but indices over specific tags in the XML-tree can improve performance in query intensive applications. Indices allow to efficiently retrieve all XML-elements that satisfy search conditions on the search key nodes of the index; which is why XMLStore should support indices. As an example take the query `//tagname/subtagname` which

returns a subtree. The way XMLStore is implemented today it needs to search the whole tree to be able to locate a specific subtree. With an index based on *tagname*, *tag_ref* it would be easier to pinpoint the exact location from where the in the subtree the query should execute. This would especially be helpful in a wide and deep graph, where the <tagname>-tag is deeply nested.

The indexes could either be user defined, or automatically generated over the statistically most queried data. These indexes could of course be stored in the XMLStore itself. A possibility could be to create the index as a B+ tree as these easily can be recreated in XML store because it is possible to imitate a rooted DAG due to the nature of the XMLStore.

5 State-of-the-art XML Storage Technologies

The XMLStore is in its early days and when compared to relational database implementations it does not have a well developed foundation of features helping it to enhance performance. But XMLStore has its advantages. It is based on the DVM model which gives it possibilities to be represented as flat files distributed on different systems. The flat files technology is supported by a transparent lazy loading strategy which the application programmer will not notice. The lazy loading strategy gives the possibility to query large XML documents without using up all memory. The DVM is based on value programming where values are immutable object and that reference is created to point to the value representation. Due to this representation it is possible to share values and then when building data in a tree structure the DVM calculates the references hereby reusing already existing value object and thereby minimizing space overhead. Child nodes are obtained by using these value references. Nodes are not only restricted to be located in one document. One of the ideas with XMLStore is support of nodes being shared between documents as well as a stateless peer to peer network solution where all peers contribute on equal terms.

XMLStore does not include facility to either locking of nodes when two or more threads access a subset of data. If XMLStore had implemented some sort of tree merging algorithm so that nodes could be merged when accessed, locking is not needed i.e. if two threads wishes to update the same subset of nodes then XMLStore would create two different working subsets, one for each thread, where each thread would make its modifications, hereby always having the newest version of data ready for each process. Then for the changes to apply updating of the subsets is necessary and this could be done through the use of a three merging strategy.

	TEXT	DTD	Edge	ATTR	Object	Oracle	Tamino	RawXMLStore
DB1 114MB	320	100	390	295	268			
10MB TCSD						12,95	20,17	24,54
% Usage	280,70	87,71	342,10	258,77	235,08	129,5	201,7	245,4

Table 29: Space consumption of each approach (in MB).

The TEXT, DTD, Edge, Attr and Object approach all was divided into data and indices storage. We have added these data and indices fields to compare the space consumption to Oracle, Tamino and RawXMLStore. This is a very rough overview of the different solutions and we only wish to compare the prior results with XMLStore. What we can see from Table 29 is that the DTD approach is truly optimising the space consumption only being 87 % of the original document and that oracle representing parsed data at 129,5% of the original layout. This could off course vary depending of the structure of the XML data. Further more it worth to notice that the RawXMLStore enters a 5th place with a standard XMLStore implementation but compared to the nearest combatants it performs almost equal to those giving it great credit.

When it comes to querying data we found in our empirical evaluation that there is some overhead associated to loading data from the XMLStore. The lazy loading functionality helps on low nested queries or very specific queries that can eliminate the search and thereby loading of subtrees. This is an ever greater encouragement for implementing some of the special features of the approaches described in [TDCZ02], such as indices and query optimizing would bring down the loading of elements when performing queries as our tests shows of the Text Based Approach. Also the memory consumption is very high while performing queries; we have suggested using the Flyweight patter of [GHJV94] to help on this problem.

Overall we feel that the XMLStore is a solid application, we did not experience any problems at all using it (except for human errors on our part). But to be competitive with commercial systems we suggest that further development is needed on enhancing performance of querying.

Literature

[GB03] "Survey of XML Storage Technologies", by Morten Guld and Eske Bentzen; Bachelor's project, DIKU, May 2003.

[PLANX] <http://plan-x.org/xmlstore/>

[TA03] "XPath Engine for XMLStore", by Thomas Ambus; Master's student project, DIKU, May 2003 (see <http://www.ambus.dk/planx/xpath/>).

[TDCZ02] "Design and Performance Evaluation of Alternative XML Storage Strategies", by Tian, DeWitt, Chen, Zhang; SIGMOD Record, Vol. 31, No. 1, March 2002, pp. 5-10, <http://www.acm.org/sigmod/record/issues/0203/SPECIAL/1.tian.pdf.gz>. See also their 26 page technical report.

[ABSK99] Data on the Web: From Relations to Semi structured Data and XML, by Serge Abiteboul, Peter Buneman, Dan Suciu; Morgan Kaufman, 1999.

[BT02] "Value-oriented XML Store", by Kasper Bøgebjerg Pedersen and Jesper Tejlgaard Pedersen; Master's thesis, ITU and DTU, 2002. <http://www.it-c.dk/~kasperp/xmlstore/pdf/thesis.pdf>.

[APACHE] The Apache Software Foundation, <http://www.apache.org> and <http://xml.apache.org/>.

[LY99] The Java™ Virtual Machine Specification, Second Edition, by Tim Lindholm and Frank Yellin, (<http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>).

[XBENCH] <http://db.uwaterloo.ca/~ddbms/projects/xbench/>.

[RG03] Database Management Systems, by Raghu Ramakrishnan and Johannes Gerhke. Mc Graw Hill, 2003.

[GHJV94] Design Patterns, by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, 1994.

[FK99] A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database, by Daniela Florescu and Donald Kossmann, IRNIA, 1999.

[HN02] XML Store Components (located in XMLStore cvs: /doc/components.txt), Henning Niss, 2002.

[XML] <http://www.w3.org/TR/2004/REC-xml-20040204>.

[SGG02] Operating System Concepts sixth ed. Windows XP Update, by Avi Silberschatz, Peter Baer Galvin and Greg Gagne. Wiley, 2002.

Appendix A

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="a" type="xs:string"/>
  <xs:element name="bib">
    <xs:simpleType>
      <xs:restriction base="xs:string"/>
    </xs:simpleType>
  </xs:element>
  <xs:element name="cr" type="xs:IDREF"/>
  <xs:element name="def">
    <xs:complexType mixed="true">
      <xs:choice minOccurs="0" maxOccurs="4">
        <xs:element ref="cr"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name="dictionary">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="e" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="e">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="hwg"/>
        <xs:element ref="vfl" minOccurs="0"/>
        <xs:element ref="et" minOccurs="0"/>
        <xs:element ref="ss"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:ID"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="et">
    <xs:complexType>
      <xs:sequence maxOccurs="16">
        <xs:element ref="cr"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="hw">
    <xs:simpleType>
      <xs:restriction base="xs:string"/>
    </xs:simpleType>
  </xs:element>
  <xs:element name="hwg">
    <xs:complexType>
      <xs:choice maxOccurs="15">
        <xs:element ref="hw"/>
        <xs:element ref="pr" minOccurs="0"/>
        <xs:element ref="pos" minOccurs="0"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name="pos">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="adj."/>
        <xs:enumeration value="adv."/>
        <xs:enumeration value="conj."/>
        <xs:enumeration value="n."/>
        <xs:enumeration value="prep."/>
        <xs:enumeration value="v."/>
        <xs:enumeration value="int."/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

```

        </xs:simpleType>
</xs:element>
<xs:element name="pr" type="xs:string"/>
<xs:element name="q">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="qd"/>
      <xs:element ref="a" minOccurs="0"/>
      <xs:element ref="w"/>
      <xs:element ref="bib" minOccurs="0"/>
      <xs:element ref="loc"/>
      <xs:element ref="qt"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="qd">
  <xs:simpleType>
    <xs:restriction base="xs:short"/>
  </xs:simpleType>
</xs:element>
<xs:element name="qt">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="6">
      <xs:element ref="cr"/>
      <xs:element name="i" type="xs:string"/>
      <xs:element name="b" type="xs:string"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
<xs:element name="loc" type="xs:string"/>
<xs:element name="qp">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="q" maxOccurs="20"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="s">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="def"/>
      <xs:element ref="qp"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="ss">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="s" maxOccurs="10"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="vd">
  <xs:simpleType>
    <xs:restriction base="xs:string"/>
  </xs:simpleType>
</xs:element>
<xs:element name="vf">
  <xs:simpleType>
    <xs:restriction base="xs:string"/>
  </xs:simpleType>
</xs:element>
<xs:element name="vfl">
  <xs:complexType>
    <xs:choice maxOccurs="45">
      <xs:element ref="vd" minOccurs="0"/>
      <xs:element ref="vf" maxOccurs="4"/>
    </xs:choice>
  </xs:complexType>

```

```
</xs:element>  
<xs:element name="w" type="xs:string"/>  
</xs:schema>
```