

Development of a Demand Driven Dom Parser

**Gaute Odin Alvestad
Ole Martin Gausnes
Ole-Jakob Kråkenes**

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Trond Aalberg, IDI

Problem Description

Today XML is a widely known markup language for a number of purposes, for example to store and transfer data over the internet. W3C DOM is a popular API for accessing XML documents. In traditional W3C DOM implementations, the entire XML document is loaded to memory and accessed as objects. The limitation however, is that the object representation have high memory requirements. By using traditional W3C DOM, the memory requirements can reach up to 4-10 times the size of the XML document. An alternative W3C DOM implementation which can handle large XML documents with lower memory requirements is needed. This thesis includes evaluation of a demand driven DOM parser with lower memory consumption requirements than existing solutions.

Assignment given: 20. January 2006
Supervisor: Trond Aalberg, IDI

Preface

This report presents the final thesis for the Master of Science in Technology at NTNU. The idea was elaborated by the group members and the thesis was supervised by Trond Aalberg at the Department of Information and Computer Science.

The project was carried out by the following group members Ole-Jakob Kråkenes, Ole Martin Gausnes and Gaute Odin Alvestad. The work and struggle with this master thesis has been long and challenging, but has also a very inspiring process. Through this period we have had many problems but in the end we came up with a result we are pleased to present. We have also grown through this process, both on a personal and a technical level. Throughout the work with this thesis, the collaboration between the participants has been good. We have successfully developed a prototype, which we think has a growing demand in the industry. The thesis has been based on existing research, implementations and our own contribution.

We would like to thank supervisor Trond Aalberg for his support and help during the process.

Abstract

XML is a tremendous popular markup language in internet applications as well as a storage format. XML document access is often done through an API, and perhaps the most important of these is the W3C DOM. The recommendation from W3C defines a number of interfaces for a developer to access and manipulate XML documents. The recommendation does not define implementation specific approaches used behind the interfaces.

A problem with the W3C DOM approach however, is that documents often are loaded in to memory as a node tree of objects, representing the structure of the XML document. This tree is memory consuming and can take up to 4-10 times the document size. Lazy processing have been proposed, building the node tree as it accesses new parts of the document. But when the whole document has been accessed, the overhead compared to traditional parsers, both in terms of memory usage and performance, is high.

In this thesis a new approach is introduced. With the use of well known indexing schemes for XML, basic techniques for reducing memory consumption, and principles for memoryhandling in operation systems, a new and alternative approach is introduced. By using a memory cache repository for DOM nodes and simultaneous utilize principles for lazy processing, the proposed implementation has full control over memory consumption. The proposed prototype is called *Demand Driven Dom Parser*, D3P.

The proposed approach removes least recently used nodes from the memory when the cache has exceeded its memory limit. This makes the D3P able to process the document with low memory requirements. An advantage with this approach is that the parser is able to process documents that exceed the size of the main memory, which is impossible with traditional approaches.

The implementation is evaluated and compared with other implementations, both lazy and traditional parsers that builds everything in memory on load. The proposed implementation performs well when the bottleneck is memory usage, because the user can set the desired amount of memory to be used by the XML node tree. On the other hand, as the coverage of the document increases, time spend processing the node tree grows beyond what is used by traditional approaches.

Contents

1	Introduction	1
1.1	Motivation	1
1.1.1	Scenario	1
1.2	Problem Identification	1
1.3	Problem Text	2
1.4	Approach	2
1.5	Solution	2
1.6	Result	2
1.7	Report Outline	3
1.7.1	Prestudy	3
1.7.2	Solution	3
1.7.3	Evaluation and Discussion	3
1.7.4	Conclusion and Further Work	4
I	Prestudy	5
2	XML	6
2.1	Introduction to XML	6
2.2	History	7
2.3	XPath	8
2.4	XSLT	8
2.5	XQuery	9
2.6	Data and Document Centric XML	9
3	Handling XML	11
3.1	Parsing XML	11
3.1.1	Parsing control	11
3.1.2	XML Data Extraction	11
3.2	XML Processing	12
3.3	DOM - The Document Object Model	14
3.3.1	DOM Development	14
3.3.2	W3C DOM Example	17
3.3.3	Pros and Cons of the W3C DOM	18
3.4	Updating XML Documents	18
3.5	XML Databases	19
4	Memory Handling	20
4.1	Introduction to Memory handling	20
4.2	Caching and Virtual Memory	20
4.2.1	Cache	20
4.2.2	Virtual Memory	20

4.3	Replacement Strategies	22
4.3.1	Random	22
4.3.2	FIFO	22
4.3.3	LRU - Least Recently Used	22
4.3.4	Clock Algorithm (Second Chance)	23
4.4	Trashing	24
4.5	Load Control	24
4.6	Managed Memory Model	24
5	Indexing and Information Retrieval for XML	26
5.1	Introduction to XML Indexing	26
5.2	Retrieval Directions	26
5.3	Index Representation of XML Data	26
5.3.1	Index Types	27
5.3.2	Extent	30
5.3.3	Target	31
5.3.4	Control	32
6	State of the Art	34
6.1	A Classification of Products	34
6.2	Apache Xerces2	34
6.2.1	Xerces2 Parser Components	35
6.2.2	Use of Symbol Table	35
6.3	VTD-XML	36
6.3.1	The VTD Processing Model	38
6.3.2	Properties of the Processing Model	39
6.3.3	Navigating VTD-XML, User Level	41
6.3.4	VTD-XML Performance	42
6.3.5	Limitations	42
6.4	Other Traditional Implementation	42
II	Solution	45
7	Design	46
7.1	Demand Driven Dom Parser (D3P)	46
7.1.1	W3C DOM Support	46
7.1.2	The D3P Library API	46
7.2	Architecture	47
7.3	Use of Index	48
7.3.1	XML Node Collapsing	48
7.3.2	Symboltable	50
7.3.3	The Structure Index	50
7.3.4	The Value Index (Name Tag)	54
7.4	Loading and Unloading XML Nodes - Lazy Loading	59
7.4.1	Cache Algorithm for XML Structure	59
7.4.2	Linear Structured Cache	60
7.4.3	Tree Structured Cache	61
7.4.4	Behavior of the Cache Algorithms	61

CONTENTS

7.5	Utilizing the Managed Memory Model	63
7.6	Updating the XML Document - Lazy Approach	64
8	Implementation	67
8.1	Component Overview	67
8.2	Cache	68
8.3	DomProducer	69
8.4	XML Data Source	71
8.5	XML Index Storage	72
8.6	Parser	72
8.7	XmlWriter	73
9	Configuration	76
9.1	Configuration Tool	76
9.2	CollapseLimit	77
9.3	Max Cache Size	79
9.4	Use Cache	80
9.5	Data Storage	80
9.6	Xml Reader	80
9.7	Cache Algorithm	80
III	Evaluation and Discussion	83
10	Testing	84
10.1	Software Performance Testing	84
10.2	Testing DOM Performance	85
10.3	Test Harness	86
10.3.1	Compared W3C DOM Parsers	86
10.3.2	Test Files	87
10.3.3	The Tests	88
10.3.4	Test Technique	89
10.4	Test Preparations and Configuration	90
10.4.1	Index Storage	91
10.4.2	XML Data Reader	91
10.4.3	Cache Algorithm	91
10.4.4	Cache Size Impact on Performance	92
10.5	Test Result	92
10.5.1	Document Build Time	94
10.5.2	Tree Walk Time	94
10.5.3	Document Memory Usage Before and After Walk	96
10.5.4	Partial Traversal Test	96
10.5.5	Partial Traversal Test Repeat	96
10.5.6	Remove Test	99
10.5.7	GetElementsByTagName Test	99
10.5.8	Document Walk Time and Memory, Percentage Coverage	99

11 Discussion	103
11.1 Evaluation of the Implementation	104
11.2 Application Strength	104
11.2.1 API	104
11.2.2 Memory Consumption	105
11.3 Application Weakness	105
11.3.1 Performance	105
11.3.2 Finding Optimal Configuration	105
11.4 Deferred Overhead	105
IV Conclusion And Further Work	107
12 Conclusion	108
12.1 Comparison with Others	108
12.2 Summary	108
12.3 Conclusion	108
13 Further Work	110
13.1 Analysis Tool	110
13.2 Update Index	110
13.3 Extended DOM Support	110
13.4 Perfect Writing	110
13.5 Switch Memory Model	111
13.6 Serialize Dirty Nodes	111
13.7 Save State	111
13.8 More Test Results for Evaluation	112
V Appendix	117
A Glossary	117
B Document Object Model	119
C Sample Documents	120
C.1 Sample XML Document 1	120
C.2 Sample XML Document 2	121
C.3 Sample XML Document 3	122
D D3P UML Diagrams	123
E W3C Standard	126
F Testing	128
F.1 Test Preliminary Settings	128
F.2 Internal Performance Tests	129
F.3 General Tests	130
F.4 Special Tests	141
F.5 Percentage Tests	143

CONTENTS

F.6	Deferred Overhead	144
G	CD content	145
G.1	D3P library	145
G.2	.NET Test Suite	147
G.3	Java Test Suite	147
G.4	D3P Console Parser	147
G.5	D3P GUI Parser	148
G.6	XML File Generator	148
G.7	D3P Library Documentation	148
G.8	Testfiles	148

List of Figures

1	Sample XML document	6
2	XML with CDATA	8
3	XQuery example	9
4	Loosely structured XML file, document centric	9
5	Four approaches to reading XML data	13
6	DOM representation tree	15
7	Example of using W3C DOM interfaces	17
8	Reading values from a XML document	18
9	CPU cache methods	20
10	Virtual memory	21
11	First In First Out algorithm	23
12	Least Recently Used algorithm	23
13	Clock algorithm	24
14	The garbage collection	25
15	Li and Moon numbering scheme	28
16	Hu and Tang numbering scheme	28
17	Hu and Tang inverted index	29
18	Indexing with addresses	29
19	CD node tree	30
20	Range on CD node tree	31
21	2-level indexing scheme, selection process	32
22	Xerces2 components	36
23	XNI parser configuration	36
24	Xerces2 symbol table	37
25	VTD vs DOM and SAX	37
26	VTD record	38
27	Location cache representation	39
28	VTD context object	40
29	VTD navigation examples	41
30	Architectural overview	47
31	Sample XML node tree	49
32	Symbol table	50
33	Sample index table	51
34	Resulting index table	51
35	NodeList with no and one node visited	52
36	Partial constructed DOM tree	53
37	Tag name index, example entry	55
38	Tag name index construction	55
39	Resulting tag name index	56
40	NodeList before creating nodes	57
41	NodeList, single node created	57
42	NodeList, multiple nodes created	58
43	Static linear structured cache	60
44	Cached nodes in the tree	62
45	Node tree before cleaning	64
46	Node tree after cleaning	64

LIST OF FIGURES

47	Altered nodeList	65
48	Component overview	67
49	Class Diagram: Cache	68
50	pseudo code for consulting weak references	69
51	pseudo code for the node cleaner	69
52	Class diagram: DomProducer	70
53	Pseudo code for constructing nodes	70
54	Class diagram: XML data source	71
55	Class diagram: XML index storage	72
56	Class diagram: Parser	73
57	Pseudo code for initial parsing of XML	74
58	Class Diagram: DomWriter	75
59	Pseudo code for writing back XML	75
60	Indexing tool, index tab	76
61	Indexing command line tool	77
62	Indexing tool, settings tab	78
63	Cache size impact, time usage	93
64	Cache size impact, memory usage	93
65	Full traversal, file 5 time usage	95
66	Full traversal, file 5 memory usage	95
67	Partial traversal, file 16 time usage	97
68	Partial traversal, file 16 memory usage	97
69	Partial traversal repeat, file 16 time usage	98
70	Partial traversal repeat, file 16 memory usage	98
71	RemoveTest, File 16 time usage	100
72	RemoveTest, File 16 memory usage	100
73	GetElementByTagName, file 16, measured time	101
74	GetElementByTagName, file 16 memory consumed	101
75	Traverse time for percentage document cover	102
76	Memory consumed for percentage document cover	102
77	XML DOM core level one	119
78	Xml sample document #1	120
79	Xml sample document #2	121
80	Xml sample document #3	122
81	UML cache diagram	123
82	UML parser diagram	124
83	Generic example to get a node in the hierarchy	125
84	Internal tests	129
85	Deferred overhead	144

List of Tables

1	State of the art implementations	34
2	Test files	87
3	Document build time. test file 5	94
4	Document build time. test file 2	94
5	Document walk time. test file 5	96
6	Testfiles for the XML parser test harness	128
7	Test settings	128

1 Introduction

1.1 Motivation

Today XML is a widely known document standard for a number of purposes, for example storing data and transferring data over the internet. W3C DOM is a popular API for accessing XML documents. In traditional W3C DOM implementations, the entire XML document is loaded to memory and accessed as objects. The limitation however, is that the object representation have high memory requirements. By using traditional W3C DOM, the memory requirements can reach up to 4-10 times the size of the XML document. An alternative W3C DOM implementation which can handle large XML documents with lower memory requirements is needed.

1.1.1 Scenario

W3C DOM is commonly used because of its benefits, but it also has some disadvantages. In these scenarios, problems with traditional DOM parsers are presented. Our solution is aimed to solve the problems which appears.

- A user only utilizes 10% of a large XML document. By using a W3C DOM implementation which only loads the necessary parts of the document, the memory requirements will be reduced.
- A user has a XML document which will, when using traditional W3C DOM parsers exceed the limits of the internal memory when parsed. The user needs the functionality provided by the W3C DOM, but at the same time a parser which has low memory consumption independent of the XML document size.

1.2 Problem Identification

Traditional W3C DOM implementations load the entire XML document to memory, represented as objects. This approach has high memory requirements, hence the size of the XML document limited by the internal memory available on the computer used.

By implementing a W3C DOM, which only loads nodes actually used, we give the opportunity to omit the high memory requirements. To accomplish this, the implementation must have knowledge regarding the structure of the XML document. This enables the option to have a controlled memory consumption, by holding a low number of node objects in memory at a time.

1.3 Problem Text

By the discussion in the *Problem Identification*, the basis for this thesis which can be described as:

“Development of a demand driven DOM parser with lower memory consumption requirements than existing solutions.”

1.4 Approach

To get a foundation for the subject of this thesis, the basics of XML and XML handling techniques has been firmly examined. To accommodate the needs in memory handling, basic concepts has been reviewed. In particular, the understanding of how the managed memory model works was of interest. Indexing and caching has also been a part of the foundation for the implementation.

To get an overview of the state of the art, a number of XML processors have been examined. Both W3C DOM based implementations and others with alternative approaches, regarding improved memory consumption, have been reviewed. Concepts from these implementations have been taken to consideration when developing the prototype.

The implementation has been under constant development since the start, always performing better, both in terms of memory consumption and processing time. Techniques have been tested. As a result of constantly improving the prototype, much testing has been done over and over again, resulting in the outcome of this thesis.

1.5 Solution

A number of components have been assembled to construct a W3C DOM implementation which does not have the limitation of high memory consumption.

The proposed solution has a memory cache for holding XML nodes currently in use. The size of this cache can be dynamically altered at runtime. An index which describes the structure of the XML document is actively used when navigating the object node tree. Only accessed objects are constructed in memory. When the memory used by the application is exceeding the memory cache size, memory has to be reclaimed by removing node objects from memory. A decision of which nodes to remove are done by a least recently used algorithm (LRU), which selects the node which has not been visited for the longest period of time.

1.6 Result

This thesis describes the implementation and evaluation of a demand driven DOM parser. The implementation has been compared to other implementations, both traditional implementations and others with techniques for using less memory.

The proposed prototype performs well when the bottleneck is memory usage, since the user can adjust the amount of memory used by the application. On the other hand, as the document coverage increases, time spent processing grows beyond what is used by

traditional approaches. This is expected through theory of lazy loading, and confirmed by the result of this work.

1.7 Report Outline

This report is organized in the following matter:

- Prestudy (ch 2 - 6).
- Solution (ch 7 - 9).
- Evaluation and Discussion (ch 10 - 11).
- Conclusion and Further Work (ch 12 -13).

1.7.1 Prestudy

In chapter 2 and 3, XML and techniques used for parsing and processing XML is presented. Special attention is aimed towards W3C DOM which represents much of the basis for this thesis.

Chapter 4 presents techniques for handling memory, such as caching, virtual memory and the managed memory model. Knowledge of these techniques is important for the basis of controlling memory consumption.

Chapter 5 contains background knowledge of indexing and retrieval techniques for XML in particular. This chapter is important for making the right decision about the construction of indexes.

Chapter 6 deals with the state of the art. A number of existing solutions are presented and examined. Implementations which utilize techniques for controlling and minimizing memory consumption are particularly emphasized.

1.7.2 Solution

Chapter 7 introduces the design of the proposed prototype. All approaches used in the solution are discussed and thorough accounted for. In particular the indexing approaches and memory handling techniques are issued.

Chapter 8 deals with implementation specifics. Here, a component overview is presented and each component of the proposed prototype is presented in a technical manner.

Chapter 9 presents configuration of the implementation. The implementation has a number of configuration alternatives which can have impact on performance.

1.7.3 Evaluation and Discussion

Chapter 10 presents both the purpose of basic software testing and testing of the Document Object Model in particular. To evaluate the prototype, a number of tests are

adapted from other sources, but some are also constructed for the purpose. A test harness for the solution is presented.

Chapter 11 discussed both the proposed solution and test results and. Strength and weaknesses of the proposed approaches is presented.s

1.7.4 Conclusion and Further Work

Chapter 12 and 13 presents a summary, conclusion and further work for this thesis. Further work deals with aspects of this thesis which still has potential for improvement, both in development and implementation.

Part I

Prestudy

In this part we present research and state of the art in the field of this thesis. As this thesis leads to an XML processing library we start with a introduction to basic XML in chapter 2. We further move on to techniques for handling XML in chapter 3 with special attention to W3C DOM in chapter 3.3. W3C DOM is the object model used as a basis for the proposed prototype implementation. Memory handling, presented next is essential for controlling the memory consumption and lower the memory requirements. With approach to create a demand driven DOM parser, an index is required to hold information regarding the structure of the XML document. To make the right decisions when constructing the index for our need, thorough background knowledge is required, therefore XML indexing techniques are presented in chapter 5.

It is highly recommended to examine this part to get a deeper understanding on the subjects, which is later on put further in relation in the solution presented in section 2.

2 XML

2.1 Introduction to XML

XML [5] (eXtensible Markup Language) is a markup language developed by the W3C¹ [4] in 1998. It is a human readable format which is widely used for exchanging information in internet based applications. XML is a self describing and hierarchical markup language, which do not require pre defined tags. Example of an XML document is shown in figure 1. Sample document 2 is used.

```
<?XML version="1.0" encoding="UTF-8"?>
<books>
  <bookstore>
    <book>
      <title lang="eng">Harry Potter</title>
      <price>29.99</price>
    </book>
    <book>
      <title lang="eng">Learning XML</title>
      <price>39.95</price>
    </book>
  </bookstore>
  <storage>
    <book>
      <title lang="eng">XML for real programmers</title>
      <price>34.50</price>
    </book>
    .....
    <book>
      ....
    </book>
  </storage>
</books>
```

Figure 1: Sample XML document

As the XML document in figure 1 shows, the first line provides process instruction information about the document. In this case it informs that this document is of type XML version 1.0, and is encoded in UTF-8. If no encoding is provided, UTF-8 is used by default. This document provides a set of self describing data, which describes a bookstore containing multiple books. Each book contains a *title* and a *price* element. Note that the *title* element has an attribute assigned to it (*lang = "eng"*), which is used to provide additional information about an element. XML has a number of advantages [24]:

- *Simplicity*: The XML standard is short (50 pages) and XML is a human readable markup language.
- *Open standard*: XML is both platform and system independent. It builds on prior ISO standards, and supports all languages in the world.

¹W3C (World Wide Web Consortium) founded, in 1994, is an international consortium working together to develop Web standards. They developing protocols and guidelines that ensure long-term growth for the Web.

- *Scalability*: The users can define tags themselves, making the markup language highly flexible.
- *Divide between data and presentation*: XML is a write once, run everywhere markup language. XML contains the data, but the data can be presented in various ways, by the use of style sheets and/or transformations.
- *Load balancing*: XML is not dependent on an on-line data source. The user can use the XML locally.
- *Integrations*: XML integrates with different data sources.

The W3C proposal for the XML document standard defines of requirement, in order to determine if an XML document is valid. Some of the most important requirements are:

- The documents must only have one root.
- All elements must have a start and end tag, or the element must be self-closing.
- Elements must be logically nested. In example: `<this><that></this></that>` is not valid, but `<this><that></that></this>` is valid XML syntax
- The reserved character `<`, `>` and `&` is in use of the XML and cannot be used for other purposes. Instead, `<`, `>` and `&` respectively must substitute for these.
- Attributes are in quotes (`"` or `'`. The user can choose, but the end quote must match the start quote.

All documents satisfying these requirements are in definition well-formed XML documents. Since XML is so flexible, the need for constructing sets of restrictions was announced. The XML standard includes restrictions in the sense of a DTD [6] (Document Type Definition). An XML document is considered valid if the content matches its definitions in the DTD. The DTD specifies which tags and attributes are allowed in the XML document. For example, a DTD can say that the "this" tag may be inside "that" tag, but not the other way around. The benefit of DTD is that applications using the XML document has knowledge of how the document is structured. A parser which perform validity checking using DTD, can verify the structure of the XML document. This way, content and error checking in the application is not necessary. The DTD is fairly simple, and more flexible restriction frameworks has later been released, such as the XML Schema [6].

Reserved characters might cause problems when the data stored in an element explicit uses these characters. For example, when documenting software code or mathematical equations. To deal with this eventualitet, data containing reserved data is encapsulated in a CDATA[] clause. An example of CDATA usage is shown is figure 2.

2.2 History

Already in the sixties, the task of developing a standard markup language for documents started. The first result was SGML (Standard Generalized Markup Language - ISO 8879) [16]. The SGML was a comprehensive and complex framework built on IBM's

```
<math-lesson week="43">
  <math-for-dummies>
    <![CDATA[for N > 0, N is positive]]>
  </math-for-dummies>
</math-lesson>
```

Figure 2: XML with CDATA

own markup language, GML. SGML never made it to the top; it was too difficult to use. Many markup languages have been developed on the basics of SGML, including HTML and XML.

HTML is an immensely popular markup language on the internet, developed by the W3C. However, it has its limitations. When first introduced, HTML contained approximately 100 predefined tags. As time passed more tags were added to satisfy needs in the industry. As applications on the internet became more popular, a need for a more dynamic markup language grew. Developer wanted their own tags added to the HTML, and the W3C could obviously not satisfy everyone. XML was a result of the limitations of HTML. With the basics of HTML and a framework like SGML, W3C developed XML as the best of both worlds. XML is described as 20% of the specification size and 80% of the functionality of SGML. In other words, it should be easy to use.

2.3 XPath

XPath is a query language to make it easier to select specific parts of an XML document. It can also be used to define navigation in an XML tree, using 13 different axes. The functionality of selection data with XPath is divided into three parts:

Xpath expression: Selects the path in the node tree. For example, `"/bookstore/book"` selects all elements named "book" which is child of "bookstore", and "bookstore" is the root-element.

Xpath functions: Selects an operation to be performed on the selected node set. The `"count()"` function, for example, returns the number of nodes in the set.

Predicates: Acts as a filter for the selected node set. Only nodes which are returned from the filter are part of the new returned node set. Using the same example as in the XPath expression, only books which cost over \$30 are selected: `"/bookstore/book[price > 30]"`.

2.4 XSLT

XSLT [6], eXtensible Stylesheet Language Transformations, is a programming language used to convert XML to other outputs, i.e. another XML document, HTML or any other text based formats. XSLT uses XPath for selection of data to be transformed. For example, if a company wants XML data formatted in a specific manner, in order to integrate with their system. If existing XML data does not already have the correct format, an XSLT document describing the transformation can be applied to the XML document. A XSLT

processor is needed to transform the XML document using the XSLT style sheet. XSLT is also written in XML, making XSLT integrate perfectly in legacy XML based systems.

2.5 XQuery

XQuery is for XML what SQL is for RDBMS (Relational Data Base Management System), a query language for retrieving data from XML Documents. XQuery is a superset of XPath, and aims to overtake the role of XPath. However, both these are used today. XQuery is in contrast to XPath not XML based, and utilize the benefit of XML Schema to optimize querying. XQuery does not support for updating XML documents in version 1.0, but future versions (version 2.0) will have this support. The benefit of XQuery is its ability to query a collection of documents, and not limited to one document as with XPath. This makes XQuery the perfect XML Query language for XML databases. The XQuery syntax is based on the phrase **FLWOR** or **F**or **L**et **W**here **O**rdery **R**eturn. An example XQuery expression using example XML document 2 is:

```
{
  for $x in doc("books.xml")/bookstore/book
  where $x/price>30
  order by $x
  return $x/title
}
```

Figure 3: XQuery example

2.6 Data and Document Centric XML

XML documents are typically divided in two types [7]; data centric and document centric XML. Data centric XML has a strict schema, defined by a rule set like DTD or XML Schema. These types of documents are ideal for communication in which data has a predefined structure. Documents of this type have less requirements for self-describing tags, because humans are less likely to read these documents.

```
<?XML version="1.0" ?>
<storage>
  <product><description><productname>W800i</productname> from
    <manufacturer>Sony
      Ericsson</manufacturer> has received a mark of <evaluation>4</evaluation>
    from <reference>www.amobil.com</reference></description>
  </product>
  ...
  ...
</storage>
```

Figure 4: Loosely structured XML file, document centric

Document centric XML has a more loosely structured schema. Actually, the document is likely to not have an associated schema at all. This means data can be registered in

the XML document with no regards to rules, other than it must be a well-formed XML document. Figure 4 shows an example of loosely structured XML. Document centric XML are usually more difficult to handle than data centric, especially if no schema is associated.

3 Handling XML

3.1 Parsing XML

Parsing XML documents is more than just reading character data. The purpose of the parser is to abstract the process of reading and interpreting of data streams, presenting the interpreted data to the user through an API. There are different approaches for how data is read to the parser, and how reading is controlled. Another aspect of parsing is the decision of how to use the XML data once it is read. Chapter 3.1.1 and 3.1.2 address there two aspects.

3.1.1 Parsing control

There are basically two approaches for controlling parsing of XML data [2]. Push and pull based parsers:

Push Parser: Runs through the entire document, triggering events as it encounter nodes. The developer however, has limited control over the data stream, and the approach can produce complex code. This is the most used XML streaming approach today, and is generally referred to as SAX [17, 6] (Simple API for Xml). SAX was developed as an initiative from the xml-dev mailing list. Even though SAX is the fastest approach, it has a number of disadvantages as well. To retrieve specific parts of a large XML document, the user has to wait until the parser reaches this part. There is no opportunity to 'skip' parts of the document, for fast forwarding. Nor can SAX parse backward from the current position in the file. This means that finding a small part of the document can time consuming. The entire document has to be parsed again and again if the user wants to use the same part of the document many times, because nothing is loaded permanently to memory. By default, SAX does not permit any update of data. It is merely a parser for retrieving data.

Pull Parser: This is the most basic approach to parsing, and the developer is always in control of the stream. The entire file is read, one character at a time. When it encounters a node, the control is passed to the application, which can take further action. For example, when an element is found, the developer can ask what type of node comes next. This is impossible in a traditional push parser; the next node must be triggered as an own event, providing less control. As with Push parser, there are no going back in the stream with a pull parser.

3.1.2 XML Data Extraction

When the parser has read and interpreted the data, it is ready for further processing. Both with push and pull based parsing, events are triggered when the interpreted data is found to have a structure or context which is defined in a rule set. The action performed is often referred to as a part of the parsing process, but at a higher level of abstraction.

The most common method for extracting XML data from a parser is called *extractive* parsing [1]. This method parses through the XML document, extracting relevant node information, and builds an in memory object of the data. After the parser is done, the source document can be discarded, as the object in memory represents the XML

document. This is the most established and well-known approach for processing XML documents. This is also the basics of the Document Object Model further discussed in chapter 3.3.

In recent years, an alternative approach has been introduced to process XML, known as non-extractive parsing [1]. This approach also parses through the XML document, but does not extract all the information as the extractive approach does. Instead tokens² are created by the start offset and length of each node and is stored in some sort of table/index. When data is to be used, the nodes are extracted on-the-fly from the source document with the help of the start offsets and length values. This means that the document source can not be discarded, as the data still is required. Some new and untraditional XML retrieval implementations have utilized this approach, such as VTD-XML [38]. VTD-XML is later introduced in chapter 6.3. The advantages by using non-extractive may not be evident, but this approach has some interesting features:

Memory consumption

The non-extractive approach will potentially consume less memory, as it does not require a complex representation of the nodes within a XML document.

Selection And Modifiability

One of the most interesting features is the improved abilities for selecting, modifying and writing nodes. Some of these are:

- Any token can be removed, added or updated without re serializing the file. Using the traditional “extractive” style of parsing, a number of string concatenations might need to be performed in order to compose an updated document. The “non-extractive” style allows unmodified regions of the content to be directly copied to a new file. Only parts which are modified are constructed and rewritten.
- Any fragment of the source data can be addressed by a pair of integer (offset + length). Because a fragment in a source data consists of a group of adjacent tokens, a fragment of the source data can be addressed or even removed by calculation the starting offset and length of the fragment.
- Fragments from multiple sources can efficiently be pulled out and splices together to compose a new document, as long as all sources are tokenized non-extractively.
- The original data is fully preserved and dealing with whitespace is easier. The original document is maintained in memory or on disk and no information is lost after the “non-extractive” tokenization.
- The integer pairs for every token in the source data can be written to a binary file, making it possible to reuse the tokens, or “parse once, use many times”.

3.2 XML Processing

When processing XML, text has to be transformed to format known to the user. Typically, this implies parsing and interpreting an XML stream. As discussed in chapter 3.1.1,

²A token is an categorized block of text

there are two approaches to control the XML data reading [36]: push or pull-based. To sum up, when using a push based parser, the activity resides with the XML source. Data is pushed to the parser, forcing the parser to precede the reading without stop. With pull-based controlled parsers, the parser is in control and pulls out of the document what it requires.

On the other hand, the type information about the document can either be interpreted or compiled. When interpreted, the parser must interpret the data at run time, with no previous knowledge about its content. For example, the XML data source can be an XSLT processor, giving different XML data due to input parameters. When using compiled typing, the document can be pre processed to give the parser additional knowledge of the document structure or content. The non-extractive parsing discussed in chapter 3.1.2 is an example of a compiled approach.

Combinations of the different control and type information approaches give four classifications of how to extract data from XML document [36], shown in figure 5.

		Type information	
		Interpreted	Compiled
Control flow	Pull	Lazy	Demand-driven
	Push	Eager (traditional)	Data-driven

Figure 5: Four approaches to reading XML data

Eager is the traditional approach, where the source is in control of the data, and the receiver interprets all data when it has been delivered. When reading the entire document, this approach is expected to have the best performance. If only a part of the document is addressed, this approach will deliver too much data, which in turn can decrease performance.

Lazy Evaluation pulls out data needed and interpreting it at runtime. When smaller parts of the document are addressed, this will result in increased performance, compared to the eager approach.

Data-driven only implements pre processing³, but all data is still delivered. When preprocessing, data accesses can be done faster.

Demand-driven combines both pull processing and pre processing. Data access is done faster, and data is delivered at request.

Selecting a XML processing fitted for the needs in the application is crucial for performance. Generally, push based approaches are best when the entire document is to be addressed. Pull based approaches will carry performance penalty because it must fetch

³Pre processing is the act of processing data before it is parsed for final usage

data as it goes. As document coverage decreases, pull based approaches is preferred. If compiled approach is an available option, it will generally deliver higher performance than interpreted.

3.3 DOM - The Document Object Model

Perhaps the most common line of action when processing tree structured documents like HTML and XML is using a Document Object Model (DOM). DOM makes a tree structure representing the document and provides an API for accessing the content and structure of the document. DOM originates from the model that describes how elements in a HTML page, like input fields, images, paragraphs etc. are related to the topmost structure; the document. Document object models were created at the time browsers started to support JavaScript, giving programmers necessary access to the HTML through an object model API.

Many vendors have implemented this concept, both for HTML and XML. W3C [4] developed a recommendation for a DOM in 1998, referred to as W3C DOM. It is a platform and language independent object model, stating interfaces and functionality to be implemented.

The W3C DOM [6, 11, 14] transforms documents into a logical object oriented tree structure of nodes. All nodes can be accessed through the DOM node tree. Their contents can be extracted, modified or deleted, and new nodes can be created. All of the structures in a DOM tree inherit from a mutual interface called *Node*. From this interface, specialized types of nodes are created and inherit all of the methods and properties of the *Node* interface. Like *CDATA*, *Element*, *Text*, *Attributes*. The complete chart over the different interfaces in DOM can be seen in appendix B. The specialized node types have methods and properties suitable for their purpose. For example, the text node do not need methods like *GetAttribute*, so this is only included in the *Element* interface. Figure 6 shows the mapping between the DOM structure and an XML document. The DOM tree is based on XML sample document 1.

3.3.1 DOM Development

As development of document object models passed, a number of models have been proposed. DOM is currently divided into 5 levels:

- The Level 0 DOM, supported from Netscape 2 onwards by all browsers.
- The two Intermediate DOMs.
- The Level 1 DOM, or W3C DOM.
- The Level 2 DOM. Also a W3C DOM. Supported by all newer browsers.
- The Level 3 DOM. Last recommendations from W3C.

Level 0 DOM

The first DOM [12] was invented by Netscape at the introduction of JavaScript. This offers access to a few HTML elements. Most important perhaps, the access to

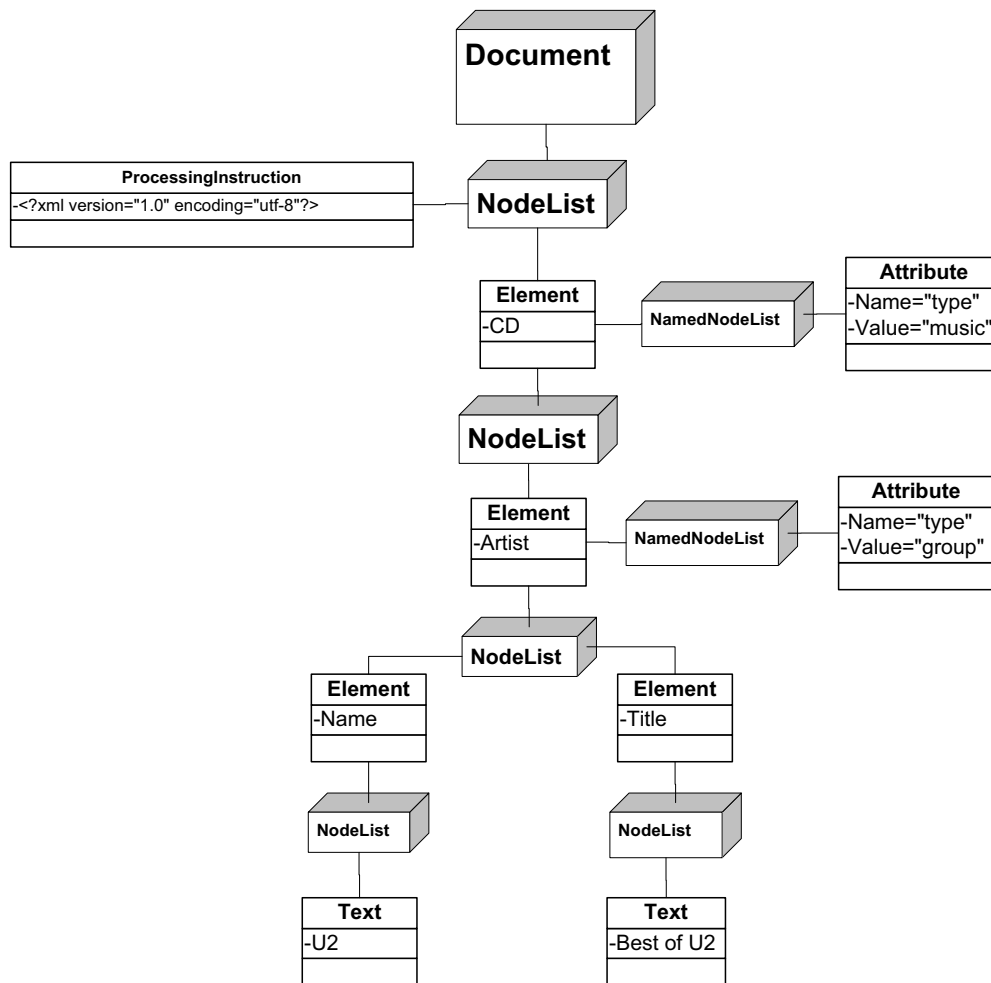


Figure 6: DOM representation tree

HTML forms. Microsoft introduced the JavaScript and DOM Level 0 in Internet Explorer 3.0. The DOM offers backwards capability and newer versions still offer support for the DOM Level 0 specifications.

Intermediate DOMs

The browsers used at the introduction of DHTML (Dynamic HTML) needed a tool to access properties on the page. DHTML needs to access the different layers on the page. These layers have properties used to manipulate the page. Like for example moving the layers. At the time, Netscape and Microsoft with their Internet Explorer decided to make their own proprietary versions.

Level 1 DOM

While Microsoft and Netscape developed proprietary versions of the DOM, W3C developed a recommendation for a standardized DOM interface. The first re-

comendation of the DOM was called the Level 1 DOM [13]. W3C DOM Level 1 concentrates on the actual core of DOM, and contains functionality for document navigation and manipulation. The purpose of the specification is to define an interface that is language independent and that can update structure content of documents. Vendors can implement the W3C DOM with their own proprietary data structures and the developers only need to be concerned about the standard interfaces defined in the W3C recommendations.

The goal of the DOM Level 1 is to define a programmatic interface for XML and HTML. The DOM Level 1 specification is separated into two parts; the Core and HTML version. The Level 1 specification provides a set of low level interfaces to represent a structured document and some additional functionality for XML documents. The HTML Level 1 provides higher level interfaces which are used with the fundamental interfaces defined in the Core Level 1.

Level 2 DOM

The W3C DOM Level 2 Core [14] builds upon the DOM Level 1 Core, hence it is backwards compatible. DOM Level 2 includes a style sheet object model, and defines functionality for manipulating the style information attached to documents. It also enables advanced traversal functions, defines an event model and provides support for XML namespaces.

Level 3 DOM

The W3C DOM Level 3 Core [15] introduces a set of new tasks, amongst other document-loading and -saving, as well as content models (such as DTDs and schemas) with document validation support. In addition, it also addresses document views and formatting, key events and event groups.

In earlier W3C DOM recommendations, renaming elements was not allowed. A new element had to be made, and all data had to be copied. With the DOM Core 3, the Document interface contains a rename method to accomplish this in a single operation. One of the new features for comparing nodes is introduced, which differs between identity and equality. For two objects to be identical they have to be the same objects in memory. Equality compares the actual data in the nodes. DOM Level 3 brings new set of methods for just this purpose. Two nodes can be compared and the value offset in respect to each other can be found.

Another new feature is methods to query XML info set information. For example, users can now query and modify information in the XML declaration like version, standalone and encoding. Also, a new property is available in the Node interface, which indicates whether a text node only contains ignorable whitespace. In early versions of W3C DOM there was no opportunity to choose which features should be a part of the implementation. Vendors of DOM APIs might have many additional features in their implementation, which most users do not need. W3C DOM core 3 supports choosing which modules should be utilized, so users do not have to suffer unnecessary performance overhead and memory consumption. A new feature in the Document interface is called `normalizeDocument`. By default the operation performs these actions:

- Consolidates adjacent text nodes into a single one.
- Updates content of entity references according to entities they refer to.
- Verifies and fixes namespace information, making namespaces well formed.

Future versions

Future versions of DOM may specify interfaces with a underlying window systems to prompt the user. It may contain a language interface, address multithreading and synchronization, security and repository.

3.3.2 W3C DOM Example

A W3C DOM core specification defines a number of interfaces to access XML data. These interfaces are defined by an IDL (Interface Definition Language), That can be translated into several language specific versions. The example code in figure 7 shows Microsoft DOM in the .NET framework 2.0 programmed in C#. The MS DOM implements W3C DOM.

```
XmlDocument xdoc = new XmlDocument();
XmlElement docElem = xdoc.CreateElement("XML-example");
XmlElement nameElem = xdoc.CreateElement("Parser-Name");
nameElem.SetAttribute("Name", "D3P");
XmlText textNode = xdoc.CreateTextNode("Demand-Driven-DOM-Parser");
nameElem.AppendChild(textNode);
docElem.AppendChild(nameElem);
xdoc.AppendChild(docElem);
xdoc.Save(testFile);
```

Figure 7: Example of using W3C DOM interfaces

The example code in figure 7 does this:

1. Create a document with a rootnode. In DOM, this node is referred to as the *DocumentElement*.
2. Makes a element node and appends an attribute to it.
3. Makes a text node with content.
4. Appends the text node as child of the element
5. Appends the element to the rootnode.
6. Appends the rootnode to the document.
7. Saves the document.

Accordingly, the XML document is created as an internal node tree structure. The structure is written to disk as XML, when the save command is issued. Note that opening and saving documents are not a part of the W3C DOM Core 1 and 2, but first implemented in the DOM Core 3 recommendations.

To retrieve data from a serialized XML document, the file has to be loaded, as shown in figure 8. This C# code will make a popup box displaying the content of the text node made in figure 7.

```
XmlDocument xdoc = new XmlDocument();
xdoc.Load(testFile);
MessageBox.Show(xdoc.DocumentElement.FirstChild.FirstChild.Value);
```

Figure 8: Reading values from a XML document

3.3.3 Pros and Cons of the W3C DOM

The W3C interface definition can potentially improve productivity, since developers only have to be familiar with one common set of interfaces. A number of implementation of the W3C DOM has been developed based on these interfaces. This, of course makes development easier. Since most DOM implementations build the whole tree in memory in some sense of a hierarchy of objects, the object representation will be significantly larger than the XML file itself. The object representation is not an ideal approach in managed memory models such as Java and .Net, where object creation is the single most expensive operation. The size of the in memory node tree is determined by implementation-specific techniques and algorithms, but often the DOM tree can be 4 to 10 times the size of the XML document. This sets a constraint on how large files can be processed in memory. This is of course not scalable for large documents.

3.4 Updating XML Documents

The W3C DOM includes methods for updating the XML structure. The document is altered in memory, and a completely new file must be written from the object representation in memory. An alternative approach of updating an XML document is to run it through a XSLT processor, writing changes to a new file as the document is traversed. An update facility for the XML Query Language (XQuery) [3] is under development, and exists now as a working draft. When completed, XQuery can be used for altering XML documents, independent of storage format. Another query language which aims for updating XML data is the XUpdate [19] from the XML:DB initiative [20]. This language was developed due to lack of update support in XQuery.

A proposal for lazy updating has been done by Cantania [18], describing how to use an update log to build the structure of the new XML file. They propose a dynamic indexing scheme built on segmentations, which makes it possible to easily update the index. It uses a hybrid between static and dynamic index identification mapping; using local offsets in each segment, and the segment has dynamic, global addressing. This

makes it necessary only to update the global numbering for those segments not changes during index update.

3.5 XML Databases

XML databases [7, 8, 9] might be the most obvious choice for handling large XML documents. Many XML databases have W3C DOM interfaces as one of many access methods, making it possible to use the database the same manner as any W3C DOM implementation. XML databases has various approaches for storing XML. XML enabled databases are traditional relation databases which through a plug-in are extended to support XML structures. These databases usually shred the XML document in rows and columns, putting the data in a relational database. Native XML databases use XML as their native logical storage. It can still be stored in plain XML text or in a proprietary format. Some XML databases uses PDOM (Persistent DOM) [9], which is serialized DOM objects, making DOM the natural access method since objects does not have to be converted to other representations before usage. Also, XML databases are tuned for the purpose of handling a repository of XML data, making it ideal for accessing huge amounts of XML.

However, a database often represents yet another service on the computer. Some databases communicate in a client-server architecture matter, making it slower for small user applications. Servers like this can require administrator or extended user rights to install or use, possible making this option harder.

4 Memory Handling

4.1 Introduction to Memory handling

Memory handling is crucial for any performance critical application. Incorrect use of memory or misconception in the understanding of basic techniques can result in terrible performance. Memory is a resource which are limited in size, which means that the memory available must be handled with care to get the most out of it.

In this chapter, many of these techniques are extracted from their context to get into depth with each one of them. Starting with the basics of caching and virtual memory, known from operation systems. Further, examining at replacement strategies for holding the right data in memory at all times, load control and trashing. Finally presenting how the managed memory model works, and pros and cons of this model.

4.2 Caching and Virtual Memory

4.2.1 Cache

When caching data [21], a copy of the data are moved closer to the user. This results in less access time when data is referred to next time, because the user can use its local copy, and does not have to get all data from the original source. This principle is not only used in computer hardware, but also in operating systems, as well as web browsers.

In computer hardware, the data copy is kept in a faster memory which resides closer to the CPU (Central Processing Unit). This memory gets more expensive the faster it gets, making it necessary to limit the size of the fast memory. There can be many levels of cache, each one smaller and faster than the next. Moving from right to left in figure 9, the address space of the component is decreasing.



Figure 9: CPU cache methods

In figure 9, only parts of the data needed by the application is copied from the disk to the memory. This carries on for all levels; the CPU cache only has copies of parts of the data which is in memory.

4.2.2 Virtual Memory

Virtual Memory [21] is a memory management technique which utilize caching, making a large virtual memory address space. The size of the virtual address space does not have any relations to the size of the physical internal memory. Basically, data which are not needed at the time are temporarily stored on the disk, but data currently needed by the CPU resides in memory. The user however, does not know where the psychical data is; only the virtual memory space is visible. Most modern operating system has

some sort of virtual memory management.

Physical memory space is limited, forcing the system to decide which data should be allowed in memory and which can be written temporarily to disk. Data are handled in chunks of data called *pages*. A page in a modern operation system is from 512 to 8192 bytes. To copy data pages from memory to disk (and back) is called swapping, hence the area which the pages are written to on the disk is a swap file or a swap hard disk partition controlled by the operating system.

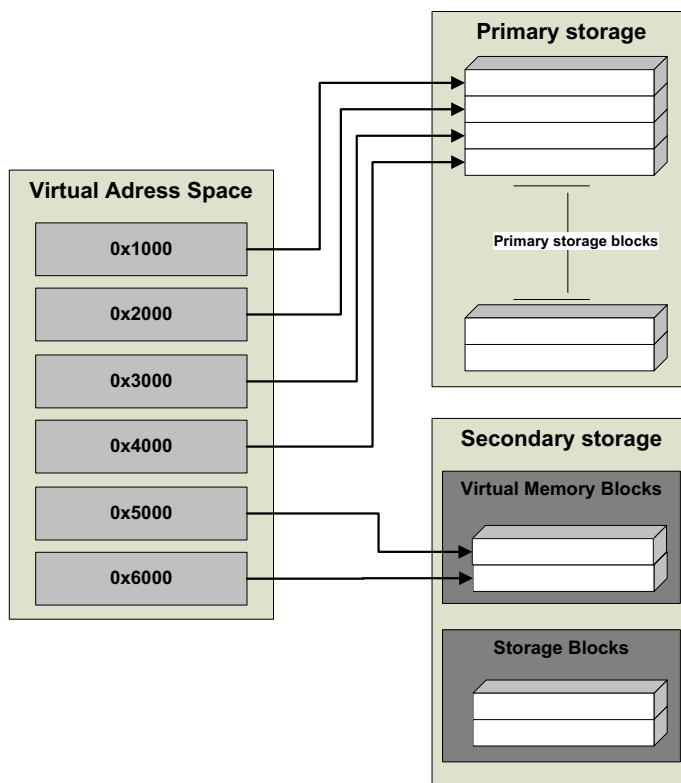


Figure 10: Virtual memory

In figure 10, the data at address 0x5000 and 0x6000 are not currently in use, and are stored on the disk. If any of these pages are referred to, they must be loaded to memory before it can be used. When an address which is referred to is found in cache, data can be used directly. This is called a *cache hit*. A *cache miss* is issued when the referred address is found to be in a page on secondary storage. If the memory is full, other pages must be removed for the newly referred page to fit. The decision of what page should be removed from memory is called a replacement strategy, further discussed in chapter 4.3. When implementing virtual memory, these issues has to be addressed [21]: *Address Mapping Mechanisms, Placement Strategies, Replacement Strategies, Load Control and Sharing*. For modern operating systems, all of this is already implemented as a part of the system. Users of the OS do not have to address these issues, it is already done. The page size is crucial for performance. The page size will determine how much data should be loaded in or swapped out of memory at a time. If the page size is small, there

will obvious be more expensive disk accesses. If the page size is larger, there will be fewer accesses, but if only a small portion of the data is used, the effort of reading the large page would be a waste. The two most important issues, replacement strategies and load control are issued in chapter 4.3 and 4.5 respectively.

4.3 Replacement Strategies

As discussed in chapter 4.2.2, physical memory is of limited size, and pages of data has to be removed from memory when another data page is due to enter memory. When selecting a page to remove from memory, it is desirable to select the one which gives the best performance. In theory this is [21]:

Select for replacement that page which will not be referenced for the longest time in the future.

It is however, not als intuitive which page will not be used in the near future. To help decide, the Principle of Locality [21] can be used:

Locality in time : *If an item is referenced to, it will tend to be referenced to again soon.*

Locality in space : *If an item is referenced, items whose address is close by tend to be referenced soon.*

The principle of *locality in time* argue that a page which has just been referred to should not be selected for replacement and the *locality in space* argue that other addresses in this same page are also likely to be referred to. Several replacement strategies are used in caches. This chapter issues four of them, from the least common used; the random replacement algorithm to the most used; the Least Recently Used (LRU) algorithm.

4.3.1 Random

Random replacement [21] means replacing pages in a random matter. This completely contradicts the Principle of Locality, where no assumption about whether the replacement will improve performance.

4.3.2 FIFO

The FIFO [21] (First In First Out) algorithm says that once a data page is loaded it is placed on top of a stack. The stack has a limited size, so when the stack is full, the page on the bottom of the stack is selected for replacement as shown in figure 11. It has no respect to whether a page is referred to again or not during its lifetime on the stack. This is only advantageous when a data page is used over a very small period of time. So when unloading, it is not referred to again in near future.

4.3.3 LRU - Least Recently Used

The LRU [21, 23] algorithm states that once a page is loaded it is pushed on a stack. So far, it is equally to FIFO discussed in chapter 4.3.2. The difference is that the LRU algorithm will put the page back on top if it is referred to again during its lifetime on the stack. Therefore, a frequently used page will als remain on top of the stack and in memory. This is showed in figure 12. Those pages not referred to in its lifetime in the

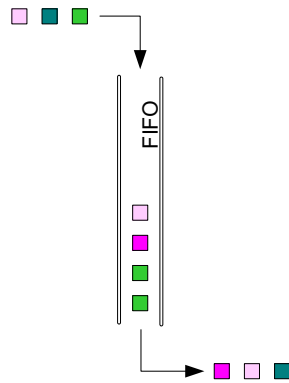


Figure 11: First In First Out algorithm

stack will be considered unlikely to be referred to again in the nearest future. Therefore, the page on the bottom of the stack is chosen for replacement.

An LRU algorithm implemented in software applications can be expensive in terms of performance. Moving objects around in a stack or list every time it is used consumes resources. A number of approaches have been suggested. For example, using time stamp for each page and update this on each reference. On replacement, the page with the oldest timestamp must be found.

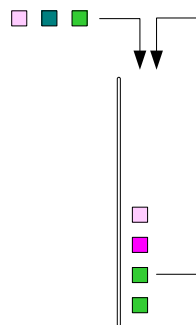


Figure 12: Least Recently Used algorithm

The LRU algorithm are considered the most accurate in terms of making the best decision of which page to replace, consequently the LRU and versions of it is used in many database systems and operation systems.

4.3.4 Clock Algorithm (Second Chance)

The Clock algorithm [21] approximates the LRU algorithm and is more cost- efficient, but is more complex to implement. All pages starts with a used-bit set to 0. When a page is referred to or loaded, the used-bit is set to 1, which means it is used and not replaceable. The list of pages is continuous cyclic scanned, looking for replacement

candidates. When a used-bit set to 1 is found, it is reset to 0. If a used-bit is found to be 0, it has not been referred to since the last scan, making it a candidate for replacement. This gives the pages a second chance of surviving. Once referred to, it must be scanned twice before assumed to be a candidate for replacement.

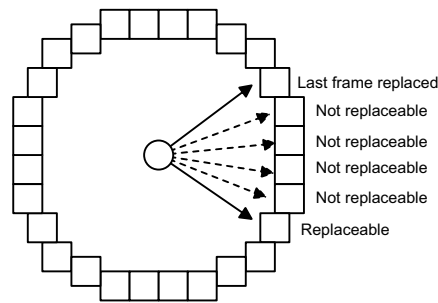


Figure 13: Clock algorithm

4.4 Trashing

The term *trashing* [21] indicates an operation on a computer that has no or limited progress, often because a resource is exhausted. For example, a running process is assigned a resource which another process are about to use. When this pattern repeats itself, much time is wasted delegating resources, and none of the processes have good progress. Trashing is known to be the worst scenario when replacing pages. That is, a page is removed from main memory, which is soon to be used by the same or another process. The purpose of any good replacement strategy is to limit trashing, in selecting pages to replace which are less likely to be addressed in near future.

4.5 Load Control

Load control [21] gives the ability to make a decision whether to use static or dynamic loading. By using static loading, all data will be loaded into memory in advance. This means that the process als has data ready and no additional I/O operations have to be performed. A static loading can be advantageous when all of the data is due to be used, but will carry performance penalty when only some of the loaded data are referred to. On the other hand, dynamically loading data will load data into memory as it is required by the application. That is, the system checks whether the required data is in memory or not. If not, only the required data is loaded to memory. This cause more I/O operations, hence will lower performance. The advantage of loading just what is required is that no operations will be performed on data which is not used. Dynamic loading is often called *lazy loading* or *demand paging*, because it loads data when it is demanded.

4.6 Managed Memory Model

Frameworks which utilize a Managed Memory Model [35], releases the developer from the responsibility of releasing memory for created objects. The memory management

does this for all objects no longer in use by the application. Examples of frameworks implementing the managed memory model are the Microsoft .Net framework and Java. In both these frameworks, applications are executed from a virtual machine⁴ which is responsible for the memory management. A Garbage Collector (GC) is an functionality within the virtual machine, which is responsible for removing objects which are not referred to either by the call stack, the current context or the application root.

The basics of managed memory management are as follows: All created objects are allocated of a heap⁵. In the sense of the virtual machine, the heap has a limited size. If the heap is full or nearly full, the Garbage Collector (GC) is issued and starts to remove objects from the heap which are no longer referred to. Figure 4.6 shows a heap which is full, with both live and dead (no longer referred to) objects. After the collector is done, all dead nodes are removed from the heap, leaving space for new objects to be created. If the heap is full and the GC does not find any objects to remove, no more memory can be claimed by the application. If the application tries to allocate memory at this time, the application will crash. It simply has no more memory resources available.

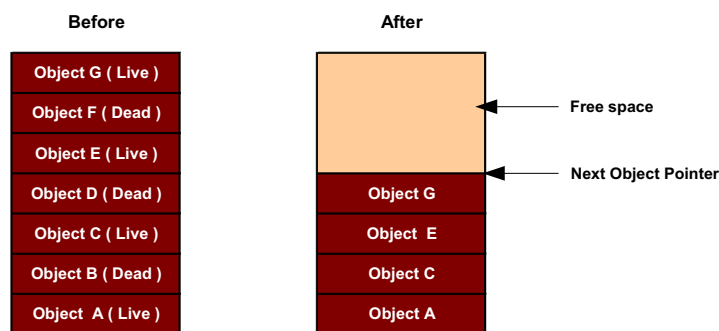


Figure 14: The garbage collection

Using the managed memory model, the developer only has to make an object unreachable, so the GC to collect it and release it from memory. The Garbage Collector can be forced to run, even though the heap is not full, but in most cases, it is best to let the GC run only when necessary.

⁴Virtual Machine (VM) is a generic emulator, executing byte code. The VM can be implemented for many platforms, hence the code can be “Written once, run anywhere”.

⁵A heap is a specialized tree-based data structure.

5 Indexing and Information Retrieval for XML

5.1 Introduction to XML Indexing

Document indexing techniques provides the possibility for faster retrieval, but when indexing a document, a number of alternatives have to be addressed. Accuracy, speed and index size are contrasts which must be taken into concern when developing an index. XML, which is a hierarchical structure represented as text coded files, can be a challenge to index for all purposes. When implementing an on demand framework from W3C DOM, an index will compensate for some of the information regarding structure and content in the XML document.

A number of XML indexing and numbering schemes are developed in the recent years to compensate for the growing interests in use of XML. Numbering schemes for XML are most used in XML databases, where the documents and collections of XML data are huge. Indexing has the objective of reducing the overall recall time when querying the whole, parts of or a collection of documents.

5.2 Retrieval Directions

Dealing with large collections of data makes information retrieval strategies important. By defining search and retrieve criteria's, a custom index structure can be designed.

There are two directions for information retrieval [25], namely information retrieval (IR) and data retrieval (DR).

IR deals mostly with text documents with information that may or may not be relevant to a submitted keyword query. It uses methods such as stemming, stop-words, n-gram and synonym lists [25] to analyze keywords. Further algorithms, such as bool, vector and probability are used to determine the relevance of information (based on submitted keywords), and retrieve them in a ranking order based on relevance. A typically example of IR is web search engines like Google. For example, the query can be "Car Dealership" and the response could include documents containing phrases such as "Bob's Car-Dealerships in Las Vegas", or "Crazy Carl deals out free spaceships to NASA" or even "Bubba's Automobile Marked" depending on which methods and algorithms the search engine is using.

DR deals with retrieval of concrete data from a specific query. It returns only requested data and nothing more. In contrast to IR, where additional information can be retrieved, DR only retrieves data in a known format that matches the query. There is no relevance factor to whether data can be found or not; either it is present or it is not. A database using SQL query is a typical example of DR. For example the SQL query "SELECT car_dealership FROM dealer WHERE location = Las_Vegas" would only retrieve the car dealerships located in Las Vegas, if any.

5.3 Index Representation of XML Data

As discussed in chapter 2.1, XML is a self describing semi structured hierarchic markup language. The structure makes it a challenge to decide what, how and at which extent

to design an index. An index is often related to the addressability the indexed data [10]. That is, for an index to be able to refer to data, it must be addressable. Even when data is not directly addressable, it is still possible to access the data. This is accomplished through further searching within addressable units.

The term granularity is often used as a measurement for the extent of an index. A very granular index takes concern of everything addressable by access, representing a larger aspect of the data. A more coarse grained index might use other high level addressable units. For example, a very granular index might have references to every node in a XML document, while a coarse grained index could refer to a XML document.

Index occupies space, both on disk and in cache, but generally it makes retrieval faster [10]. Indexes are also often slower on updates than retrieval. If too much time is spent updating the index, the overall benefit of using an index can be eliminated. This means that a static index which never changes, is to prefer when possible. The design of an index depends on various parts, such as structure, extent, target and control.

5.3.1 Index Types

An index should be structured to accommodate for the context in which it is used [27]. Since XML documents have a hierarchic structure, an XML index must be designed accordingly. This type of index are clearly related to *Data Retrieval* (DR) discussed in chapter 5.2. A number of XML indexes have been proposed, both static and dynamic and with different levels of information and granularity. XML can make use of both structured index (for navigation) and value index (for value retrieval). To make value and structural searches possible, each node must have knowledge of: *Parent to children relationships and General relationships (siblings)*. This is the basics to make it possible to navigate the XML tree.

Structured Index

Li and Moon [27] proposed a structure index which has long been considered to be the ruling numbering scheme. The numbering scheme is based on Dietz's scheme [26] which defined: *for two given nodes x and y of a tree T , x is an ancestor of y if and only if x occurs before y in the preorder traversal of T and after y in the postorder traversal*. Based on this knowledge, the scheme in figure 15 was introduced.

In this scheme, each node is associated with a pair of numbers, $\langle \text{order}, \text{size} \rangle$. This is an identifier which can pinpoint physical nodes. With this knowledge relationships among nodes can be determined:

Parent relationships

Node A is a parent of another node B if $\text{order}(B) > \text{order}(A)$ and $(\text{order}(B) + \text{size}(B)) < (\text{order}(A) + \text{size}(A))$. This means that node B is within node A 's interval.

Sibling relationships

For two nodes A and B which has the same parent, would node A be the predecessor if $(\text{order}(A) + \text{size}(A)) < (\text{order}(B))$.

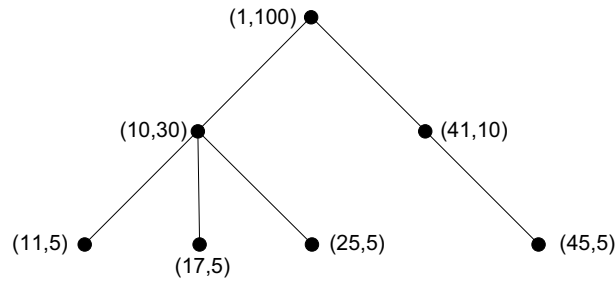


Figure 15: Li and Moon numbering scheme

Based on this, the node in figure 15 with order 10 and size 30 (10 , 30) contains all nodes with order from 11 to 30. There are unused values in the range, which is available for new node to be inserted to the XML node tree. However, there are limitations in this approach. When all reserved values are used by new nodes, no new nodes can be added to the numbering scheme. To get avoid this, an alternative indexing scheme has been suggested by Hu and Tang [28]. This suggestion takes consideration of frequent node insertions and uses a dynamic scheme that does not limit the space for new nodes.

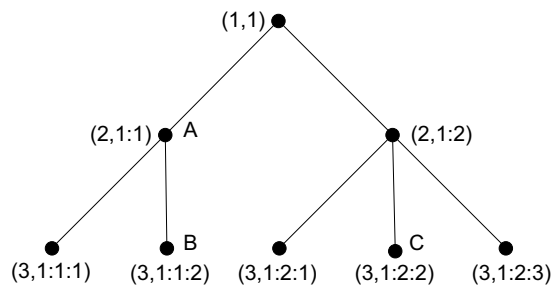


Figure 16: Hu and Tang numbering scheme

In this scheme, each node is associated with $\langle \text{level}, \text{path} \rangle$, where level indicates which level in the hieratic structure a node resides and path is of type “path to parent : current sequence number ” as shown in figure 16. The root node has als the identifier (1,1). For example, for node C in figure 16 the level is 3, path to parent is 1:2 and the current sibling-number is 2 resulting in the identifier $\langle 3,1:2:2 \rangle$. This scheme provides the opportunity to add new nodes, but only as the last sibling. Inserting nodes which are not the last sibling would require reorganizing of the numbering scheme. Similar, when deleting nodes which are not the last sibling, reorganizing of the numbering scheme is required.

As shown, flexible indexing schemes for hieratic structured data are challenging. If searching for numeric nodes, Hu and Tang scheme would perform very well.

Consider the expression “XPath:/[1][2][2]” which is node C in figure 16. This makes it easy to create an inverse index, by referring to the node identifier as shown in figure 17.

Element value	nid (element numbering value)
Hash(bookstore)	(1,1)
Hash(book)	(2,1:1)

Figure 17: Hu and Tang inverted index

Both these indexing schemes have limitations of updating. If updates were no issue, indexing scheme could remain static. Approaches for static indexing schemes has been suggested, which uses offsets inside the document or collection as reference [30, 29]. To make use of the functionality from Li/Moon [27] and Hu/Tangs [28] numbering schemes, level and sibling sequence number also has to be taken into concern. A typical complete key would look something like <documentID, start-address, endaddress, level>, as shown in figure 18.

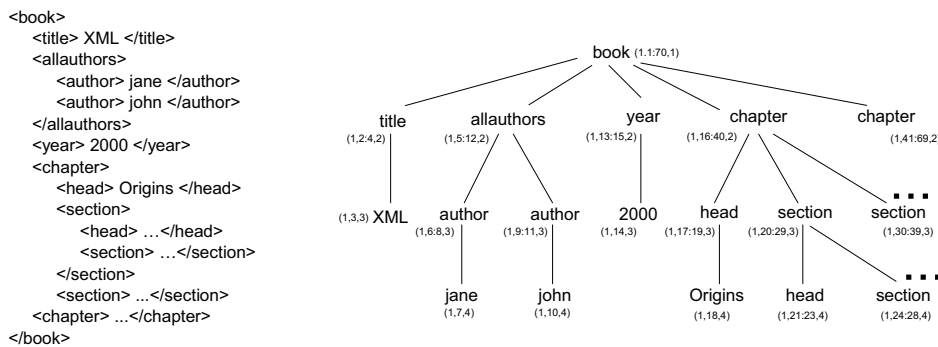


Figure 18: Indexing with addresses

Value Index

Value based indexes for XML often includes information regarding elements and attributes [10]. For example, searching for all nodes with a specific value, would only return those addressable units actually containing the search criteria ([25], DR). Also, indexes based on partly string matches could be designed. Such as a “starts with” operation, thus enabling retrieval all node values that has a specific prefix. Additional tools such as formatting of values (XPath 2.0, XQuery 1.0) and definition of values (DTD, XML Schema) could further help with more specific queries, boosting performance.

Full-text indexes enable searching through whatever element or attribute in the entire data collection. This includes searches for text which is part of bigger context, or words which have some context ([25], IR). This is part of the information retrieval strategy, discussed in chapter 5.2. This is the most demanding indexing

type, because it is difficult to predict queries, thus making it hard to design indexes based on pre defined criteria's. Full-text searches are expensive and should be used only for special purposes.

5.3.2 Extent

The extent of an index says how much of the data collection the index should include. In XML, the extent of the index can vary from node level to huge collections of data. A very granular index would accurately respond to queries from the entire collection, but would be large and slow. A special index, referring to particular pre defined granular units, or certain part of the collection would be faster with searches on the specified criteria, but would only be useful for this criteria. It is important to distinguish the purpose of the index, and choose an extent accordingly.

An index over an XML document with the highest granularity would result in an index quite large in comparison with the document itself. To get around this potential problem, Duda and Kossmann [31] suggested the introduction of the notion range, used in structural index over XML. They present range as a less granular unit, where one or more nodes (chosen representation of XML data) are encapsulated within the range. Consider an XML document with two records of a CD entry, with similar structure as in figure 15. The node tree would result in the node tree shown in figure 19.

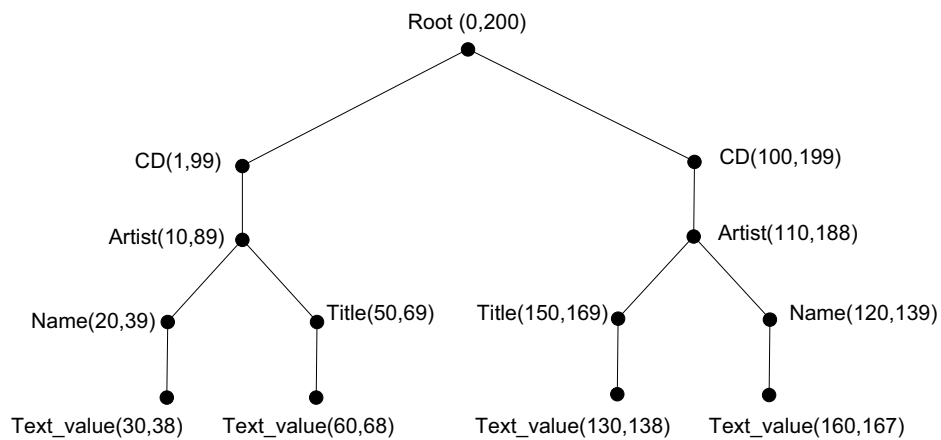


Figure 19: CD node tree

With a granularity of each node, this index would contain 15 entries. But by introducing the less granular unit (range unit), encapsulating more nodes, index entries can be reduced. The encapsulation can be based on node sizes or node count, within the range unit. If the "CD" node is encapsulated there would only be 3 entries (including root node) as shown in figure 20.

This approach would reduce the number of index lookups if more than one node is selected for retrieval in the same range unit. The decision of how much to include in the

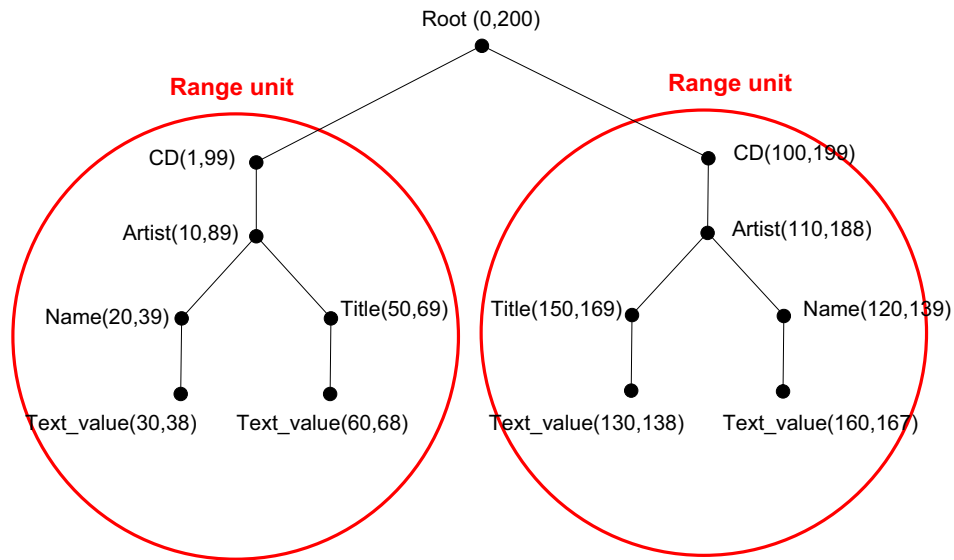


Figure 20: Range on CD node tree

index would require a balance between size of the range unit and the size of the XML document. For example a range unit size which is bigger than the address range of an entire XML document would be useless; implying the document as the range unit.

5.3.3 Target

The target is what the index is intended to retrieve. A common line of action with XML is to refer to and retrieve entire nodes [10]. Another approach is to refer to and retrieve parent nodes. This will also work as a caching mechanism where additional information is immediately available for possible future queries. The *range unit* discussed in chapter 5.3.2 uses this approach. The top node in the range unit can be the addressable unit, but sub nodes are still accessible. Having knowledge of how to process the data in the range unit, it can be further extracted. For example, an element can be returned and further parsed. Additional indexes can also be used to retrieve desired nodes inside a range unit.

In a static collection of data, data that do not change the physical address within the collection (byte, offset, word, etc..) can be the index target, thus minimizing the size and complexity of the index. In dynamic collections of data, a direct addressing scheme would be unfavorable. Every time data is altered, partially or complete reindexation is required. In this case, a numbering scheme that uses virtual addresses is preferred. It is advantageous for the index to have its own reference list, so the index do not have to mind updating any other entries than those directly concerning the node in question.

5.3.4 Control

The control aspect of the index deals with the time for the actual indexing [10]. It is also advantageous to have the index ready whenever a lookup is necessary. A usual approach is to index data at application startup or even before the application startup, as a pre processing phase. Indexing can also be done automatically, either in timed intervals or at application specific events. Examples where timed intervals are used are web search engines, where updating is not critical for the success of the operation that the system is rapidly updated. An example of event based indexing is in situations where the consistency of the system is crucial, as with business critical applications.

When the system knows when to index, it is also important to know what to index. Indexing is usually based on an instruction set, predefined for the purpose of the system. Indexes can also be made solely on the basis of some user defined criteria. For example, if a user wanted to find the owner of a car solely based on plate numbers, it would be unnecessary to index the car type, model, weight, etc. Therefore it is important to determine what data to index and what retrieval directions to take into concern.

A number of approaches [32, 33] utilize more than one index to accommodate for different retrieval requirements. These could either be completely separate indexes for each retrieval task, or the indexes could overlap and have dependencies with one another. Consider the example discussed in chapter 5.3.2 where introducing the notion of range and indexed based on the range units. This solution would only be effective on relationships queries. Node searches based on node names would be considerably expensive. A possible solution could be to create a 2-level index scheme, as suggested by Ponce, Vila and Hersch [34]. They suggested having tags in a separate index that would accommodate the name tag name retrieval process, further utilizing a second data index to retrieve the actual data of the node. The structure and sequence of events in such a scheme is presented in figure 21.

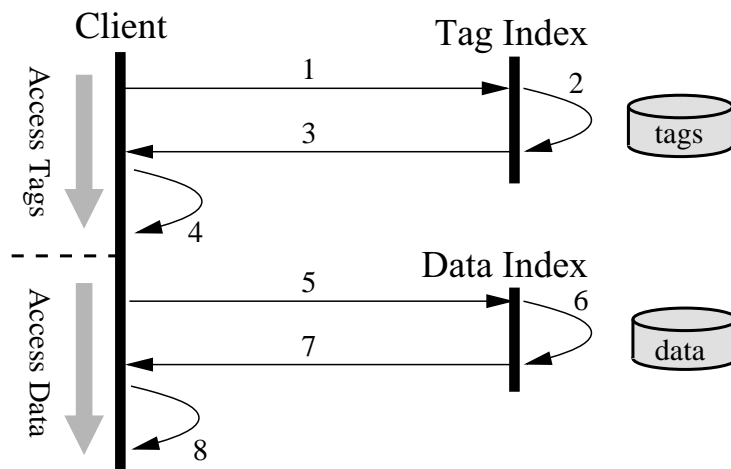


Figure 21: 2-level indexing scheme, selection process

In the case of the scheme presented in figure 21, the tag index is depended on the data index to completely retrieve nodes based on name tags. Although this would require two index lookups, the advantages comes in the form of a total smaller index and faster retrieval, as this would both reduce the data accessed and transferred compared to one large index.

6 State of the Art

6.1 A Classification of Products

Since the introduction of XML in 1998, there have been many different products to process this markup language. Most of them implement the W3C DOM interface which has developed to be de facto industry standard, providing a common API. Approaches for handling XML which tries to exceed the limitations of the W3C DOM recommendation has recently emerged, resulting in a number of different APIs

Table 1 shows XML parsers and processors most used today. In the following chapters, the most interesting of these are examined in depth, namely Xerces2 (chapter 6.2) which utilize demand driven node building, and VTD-XML (chapter 6.3) which uses a non extractive approach.

Product	Object Model	Proc Approach	Tokenization	Mem usage
Xerces2 normal	W3C DOM	Eager	extractive	8x XML
Xerces2 deferred	W3C DOM	Demand driven	extractive	0-10x XML
VTD-XML	VTD	Eager	non-extractive	1.3-2x XML
Crimson	W3C DOM	Eager	extrative	3-8x XML
JDOM	DOM classes	Eager	extrative	4-8x XML
DOM4J	D4J classes	Eager	extrative	4x XML
MS .Net DOM	.Net classes	Eager	extractive	3-6x XML
SAX	stream	Eager	extractive	low

Table 1: State of the art implementations

6.2 Apache Xerces2

The Apache Xerces2 parser [42] takes advantage of demand driven XML loading to improve performance. It has two modes, with or without demand driven parsing, which can be switched on or off with a simple command. Xerces2 call the demand driven mode for “deferred mode”, and will be referred to as this from now on. In normal mode, the entire XML document is read at document load, and is accessed as a complete W3C DOM node tree in memory. Apache Xerces2 implements all levels of the W3C DOM interfaces discussed in chapter 3.3.1.

In deferred mode, the DOM tree is created as it is traversed. When the Document node is created, the DOM tree consists of this node only. The remainder of the nodes is created as they are accessed. A technique the Xerces2 developers call *deferred node expansion*. This shortens the time used in the initial building of the document. On document load, it does not build the DOM tree but a non object oriented data structure that contains the information needed to create the parts of the tree when needed. Because only a few objects are created, the initial memory consumption is very small compared to normal mode. When the client application requests a node, the implementation creates the inquired node and all of its children. It only creates nodes which have not already been created already.

A deferred DOM is useful and effective when only parts of the XML document are accessed. If the whole document is traversed, a traditional DOM implementation would be preferred. This is due to the overhead of constructing nodes in several sessions. This results in higher memory consumption and more processing time. Xerces2 does not predict which nodes are to be requested; hence it is slow on selecting single nodes in deferred mode.

6.2.1 Xerces2 Parser Components

The Xerces2 parser is built upon the Xerces Native Interface (XNI) which defines a basic approach for building parsers components and configurations. Each component has certain dependencies which a developer must take into consideration. There are some dependencies between the components in the configuration. The dependencies can be seen in figure 22. The components are as follows:

- Fundamental Dependencies
 - Symbol Table (Further discussed in chapter 6.2.2)
 - Error Reporter
- Individual Component Dependencies
 - Document Scanner
 - DTD Scanner
 - Entity Manager
 - DTD Validator
 - Namespace binder
 - Schema Validator

The XNI parser configuration framework provides an easy to construct a custom configuration. The framework separates the API from the configuration, making it easier to choose components to use without re implementing the parser code. The information flow in the system are shown in figure 23.

- Information flows through a pipeline of components in the parser configuration
- Information goes to the parser which creates the correct API (DOM or SAX) for the configuration.

6.2.2 Use of Symbol Table

The reference implementation of Xerces2 uses a symbol table to represent common strings in the document [22] as shown in figure 24.

For lexically equivalent strings, the same identification is returned. This both reduces the amount of objects created while parsing and also gives components the possibility to perform comparisons directly on the references. The symbol table in Xerces2 also gives the possibility of providing an alternative hash algorithm. A common problem is that simple string hashing algorithm often fails to provide balanced set of hash code for symbols that are mostly unique. This is mainly postponed to strings with similar leading characters. The symbol table in figure 24 is based on example document 1.

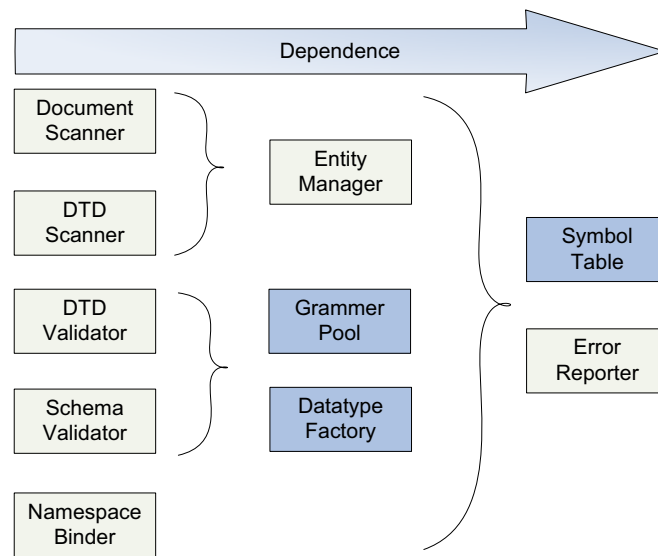


Figure 22: Xerces2 components

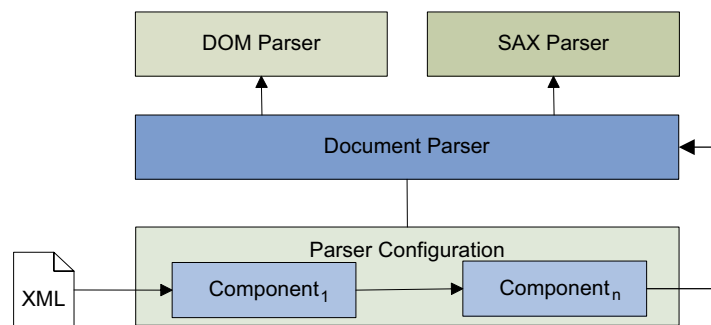


Figure 23: XNI parser configuration

6.3 VTD-XML

VTD-XML [38, 37] is an open source, non extractive XML processing API for several programming languages. To sum up from chapter 3.1.2, extractive XML means that XML is kept in its original format. Only parts of the XML which are needed are formatted for the user in mind. This reduces the large memory overhead which DOM normally produces. The amount of objects created are reduced, and since objects creation is the single most expensive operation for frameworks like Java end .Net, the load time is heavily reduced as well. The VTD-XML does not implement the W3C DOM, but has a completely different model which does not use a node tree to hold XML data and structure information. Figure 25 shows an overview of the classification of VTD-XML compared with SAX (streamed parsers) and DOM.

The API mainly differs from W3C DOM in the design to support random access without

Symbol table

ID	Type
1	CD
2	Type
3	Artist
4	name
5	title

DOM symbol representation

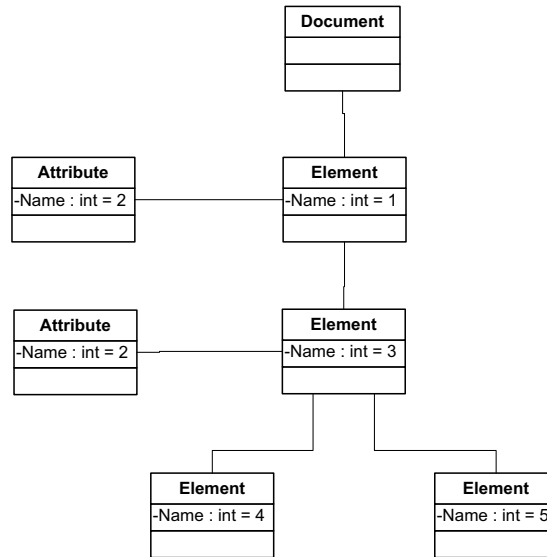


Figure 24: Xerces2 symbol table

	VTD-XML	DOM	SAX/PULL
Processing Model Description	In memory cursor based on "non-extractive" VTD records (Virtual Token Descriptor)	In memory object model based on Node object and "extractive" tokens	Low-level tokenizer based on "extractive" tokens
Performance	Fastest (outperform SAX with Null Content handler)	Slowest	Second fastest, Raw speed un-indicative of its real world performance
Memory Usage	1.3x ~1.5x of the document size	Normally 5x~10x of the document size	Doesn't grow with document size
Random Access	Yes	Yes	No, Forward only
Usability	Normally shortest code but unknown interface to programmers	Very good	Poor
Incremental Modification	Yes	No	No
Inherent Persistence (avoid parsing every time)	Yes	No	No
Hardware Implementation	Yes	N/A	N/A

Figure 25: VTD vs DOM and SAX

incurring excessive resource overhead. VTD-XML is based on VTD [38] (Virtual Token Descriptor). The VTD is a binary specification that encodes information about tokens in the XML document such as offset in the source, length, type and nesting depth. VTD-XML keeps the raw XML unchanged in memory, and uses the VTD tokens to navigate

through the data.

6.3.1 The VTD Processing Model

While making non-extractive tokenization of text usually requires only start offset and length of data, the tokenization of XML documents needs additional information. The VTD token is fit to retain this information. Figure 26 shows the content of a VTD token record. This representation of a bit level layout of the VTD token is:

- Starting offset: 30-bits.
- Length: 20-bits.
- For some token types:
 - Prefix length: 9-bits.
 - Qname length: 11-bits.
- Nesting Depth: 8-bits.
- Token type: 4-bits.
- Reserved bit: 2-bits are reserved for marking namespaces.

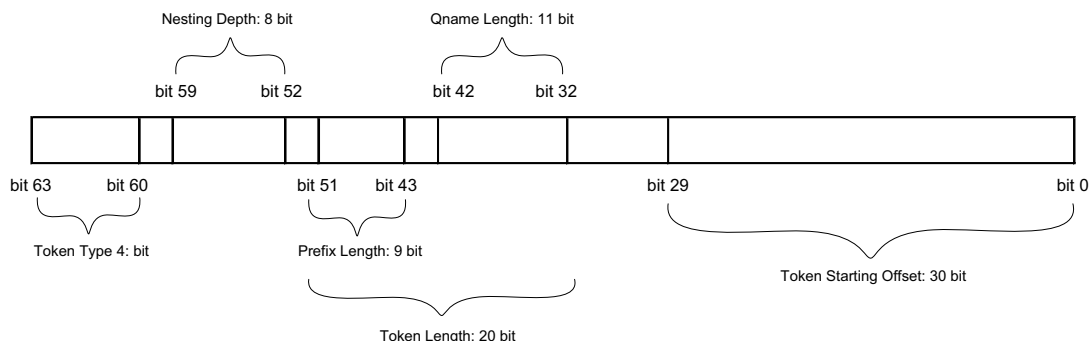


Figure 26: VTD record

The hierarchical view of an XML document is created through a collection of VTD records, in chunk-based structures called *Location Cache* (LC). There is one Location Cache for all levels of nodes in the XML document. A LC entry is a 64 bit integer where the most significant 32 bits contains the address of the VTD record and the least significant 32 bits corresponds to the VTD (and LC) record of the first child element. By navigating from the top of the XML node tree, the LC gives information of how to navigate to siblings and children of a VTD entry. Figure 27 shows the logical flow when retrieving a XML node. The root Location Cache (LC-1) consist of only one entry (XML documents has only one root node). This entry has the ID of the first child LC entry in LC-2 in its least significant 32 bits. Siblings in the XML node tree are also listed in the same Location.

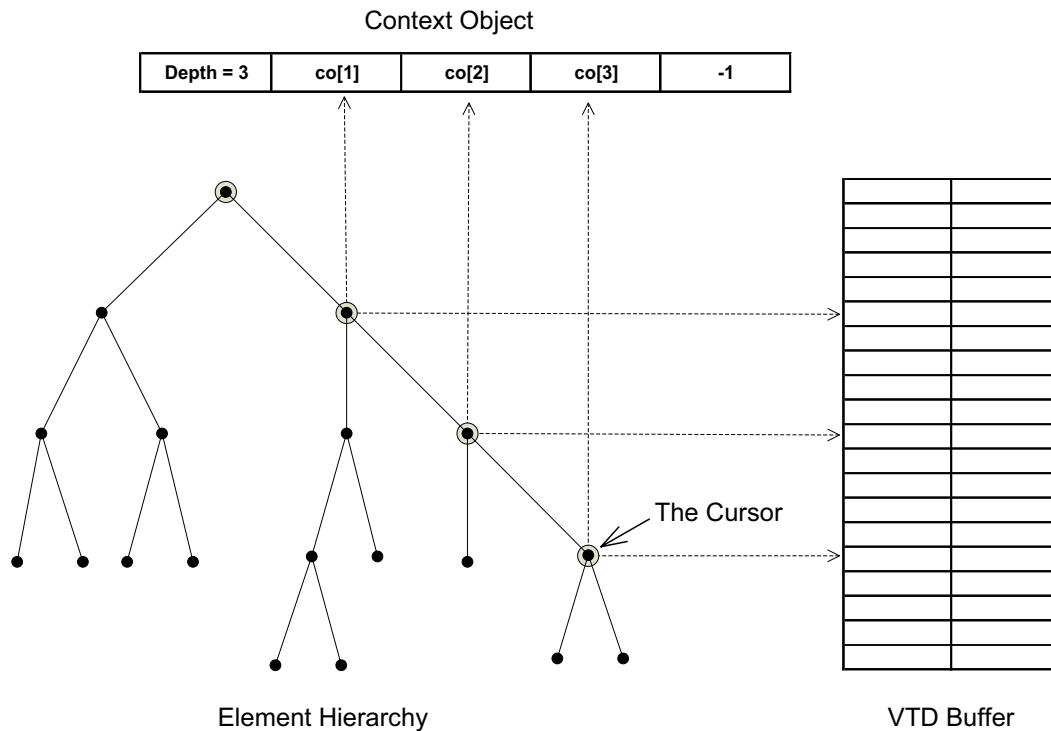


Figure 28: VTD context object

- Keep the XML data intact in memory to be used when a token is accessed. The XML data is only used when a token needs data, otherwise the VTD records is enough.
- The parsed representation makes extensive use of 64 bit integers. Both for the LC and VTD storage units.
- Compact encoding. The VTD and LC records squeeze maximum amount of information into every record. Making a perfect compromise between space and efficiency.
- Constant record length makes the records very effective. In extractive tokens, excessive use of pointers are used to associate the records with each other. When using chunk based buffers, they are associated with each other by natural adjacency. For example, attributes are also located nearby its element token, and a sibling is also after the LC entry. This removes a lot of overhead, both in terms of performance and memory consumption.
- Records are accessed with indexes sorted by integers instead of pointers. The hierarchy only consists of element nodes.

6.3.3 Navigating VTD-XML, User Level

The navigation API is divided into three layers:

- VTDGen (VTD Generator) which parses the XML data into an internal representation using VTD.
- VTDNav (VTD Navigator) which is a cursor based API so that users can traverse the tree almost like a DOM API.
- The Autopilot class which provides element traversal.

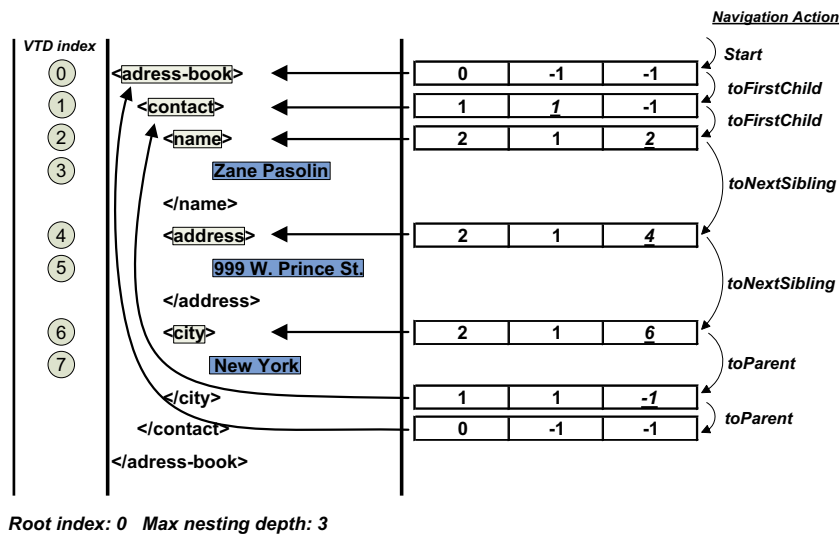


Figure 29: VTD navigation examples

On document load, the XML data is copied to a memory buffer as raw data. The data is assigned to a new instance of VTDGen, which generates the VTD-tokens for the document. When completed, the document is ready to be traversed through the use of VTDNav.

Figure 29 shows a navigation example. VTD records are shown in the table on the left hand side, and the Content Object is shown on the right hand side. Initially, the cursor of the VTD-Nav points to the root element. As traversing, the depth value of the CO and the corresponding entry (the underscored value) are updated to reflect the cursors position. To move up the hierarchy, the last CO entry must be popped and the depth value decremented. Unused entries in the CO has the value of “-1”.

The hierarchy of VTD-XML consists of element nodes only. There is a cursor for every instance of VTDNav, which is used to navigate the tree. The VTD-XML uses integers for assigning navigation direction and has a method called “toElement”. One variant of this method takes an integer that is a command for the navigation direction in the

tree. Commands like *root*, *parent*, *first_child*, *last_child*, *next_sibling*, *prev_sibling*, etc, are available. When the cursor is positioned at a specific element the user can issue the `getAttrVal` and `getText` commands to retrieve attributes and text values of the element. Member methods of the `VTDNav` such as “`getAttrVal`” return integers, representing VTD records describing the requested token.

6.3.4 VTD-XML Performance

The random access capability of the VTD-XML provides a very good performance in comparison to traditional DOM and SAX implementations. Using SAX, the user can get complex code and data might have to be parsed several times to get required results. With VTD-XML the user get most of the functionality provided by DOM and SAX but with a improved performance.

In W3C DOM, every single node type is defined as a *Node*. This means that some node types are provided with functionality and complexity which are never used. An example is text and attributes nodes. Neither have any children, thus functionality for child nodes do not need to be included. Treating these nodes as objects, increase both the memory use and performance on traversals. In VTD-XML, the VTD record is assigned a node type, where all types are treated differently with no functionality overhead.

Because VTD records consists of integers, not objects, there are no overhead in creating expensive objects. The constant length of the VTD record makes it flexible, and can be stored in large memory blocks which are easy to allocate. This leads to significant higher performance and lower memory usage. VTD-XML typically consumes memory in magnitude 1.3 to 2 times the XML document size, which is significant less than a traditional eager W3C DOM implementation

6.3.5 Limitations

One of the limitations of VTD-XML is the lack of support for entity declarations in DTDs, besides the five built-in entities (`&`; `>`; `<`; `'`; `"`). There is also a memory requirement of approximately 1.3 - 2 times the size of the XML document.

6.4 Other Traditional Implementation

Crimson

The Crimson [43] is formerly known at the Java Project X and was bundled with JDK 1.4. Crimson exists because of a disagreement between some of the IBM engineers about the internal design of the parser. The developers of Crimson argued that the parser was significantly faster than the Xerces, but test shown the complete opposite. In early releases of Crimson, Xerces was several times faster. Crimson was known to be extremely slow on walk and modify performance. Recently the development of Crimson is halted, and both the Crimson and Xerces teams are now both cooperating in the development of Xerces2.

JDOM

JDOM [39] is a simplified version of the original DOM for Java. It is an alternative to DOM and SAX and integrates well with both of them. The object model differs

from W3C DOM in two aspects. There are no interfaces, but concrete classes which simplify the API but limit the flexibility. The approach also makes extensive use of the java Collection classes which simplifies development for java programmers who is already familiar with this technology: In the documentation, the stated goal for JDOM is to provide approximately 80% of the functionality of W3C DOM with only 20% of the effort. It also provides integration with W3C DOM.

Microsoft DOM

Microsoft API for DOM [44] is included in the .NET framework, and are widely available on Microsoft's homepages. This API support all the Level 1 and Level 2 Core recommendations from W3C, and have in addition functionality to synchronize XML documents. The API can be implemented by all of Microsoft's programming languages (C#, VB, VC++, VC) provided in the .NET framework. The source-code for the API is proprietary, and no common W3C interfaces are made available in the framework. The Microsoft W3C implementation simply uses traditional eager approaches, reading entire XML data to memory as objects. Microsoft has prompt that their implementation is the most effective implementation for the MS Windows platform [48].

Dom4J

Dom4j [40] is an open source XML framework for Java which allows reading, writing, navigating, creating and modifying XML documents. Dom4J integrates with W3C DOM and SAX and is seamlessly integrated with full XPath support. However, Dom4j uses its own format internally, but W3C DOM documents can be loaded and saved to and from dom4j. The strength of the API is the use of collections classes, making it possible to have a more direct approach to programming, even though it can be more complex.

SAX

Event though SAX [17, 6] is a stream based API which does not build a object model in memory, it is a immensely popular API for simply extracting data from XML files. SAX is also mentioned in chapter 3.1.1 as the push approach of XML parsing. SAX is an option where document object models simply use too much memory, making it ideal for handling large XML documents.

Part II

Solution

*This part shows how we solved the task of implementing a demand driven W3C DOM prototype with limited memory consumption. We successfully designed and implemented a W3C DOM library capable of adjusting the memory consumption. This implementation has the name **D3P** and means **Demand Driven DOM Parser**. Further, the prototype implementation will be referred to as this.*

We start off by introducing the D3P library, explaining its purpose and extent. Then presenting the specification of the implementation, showing index schemes (structural and value), caching and replacement algorithms used to control the memory consumption and techniques utilized to pull benefits from the managed memory model.

Further, in chapter 8, a more in-depth technical overview of the prototype is presented, with help of component- and class diagrams. Finally we wrap up by present the solution from a user perspective in the form of a GUI and console tools. Here, we show different configuration alternatives, and explain how it can affects application performance.

7 Design

7.1 Demand Driven Dom Parser (D3P)

A user of the D3P library should be able to utilize this API as any other library implementing the W3C DOM interfaces, but with the ability to adjust the memory consumption. Users might experience some performance recession in terms of total time taken to do an operation, depending on the usage. This is however, expected for total traversal as mentioned in chapter 3.2. This implementation provides the possibility of using very large files without a tremendous memory consumption and without having another service running, like an XML database. D3P is simply a W3C DOM library for large files.

7.1.1 W3C DOM Support

The D3P library supports most of the functions stated by the W3C DOM CORE Level 1 specification, and some of the Level 2 CORE. It supports both structural and value specific searches related to the Level 1 CORE. This includes basic function for navigating a DOM tree structure in any direction and name based searches of the node tree. A complete list of supported functions are listed in the "W3C standard" document in appendix E.

7.1.2 The D3P Library API

This thesis describes an implementation of the W3C DOM. The W3C [4] provides a specification of the interfaces in IDL (Interface Definition Language). The IDL does not give language specific syntax information. Each programming language has to map the IDL to its own syntax.

Although programming languages like C and C++ provides the developer with more control in memory management, scripting languages like Java and C# has been adopted as the most used development platform for XML based applications. Hence this implementation is written in C#. At the time being, no alternative DOM implementation to .NET could be found. The .NET framework from Microsoft does not have the W3C DOM interfaces available, only specific classes; therefore must new interfaces be made in C# from W3Cs IDL specifications.

7.2 Architecture

The overall architecture of the solution is a three level architecture. Figure 30 defines these three logical components of the Demand Driven DOM Parser:

XML Source holds XML data. In general, this data could be supplied from any source, as long as it can be available as bytestreamed data.

Memory Management is responsible for controlling the memory consumption. It also has knowledge of how to find data in the XML Source in sense of an index. This component includes caching and replacement strategies for controlling W3C objects.

W3C DOM Interface Implementation is responsible for the logic of navigating the node tree. It has less knowledge of how the subsystem works; it simply navigates through a node tree which for this component exists as a full node tree.

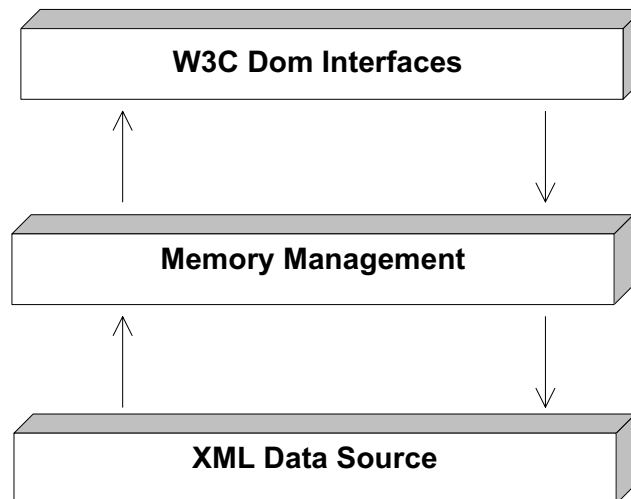


Figure 30: Architectural overview

7.3 Use of Index

Using a preprocessor to index an XML document utilize the compiled approach discussed in chapter 3.2. Information regarding the structure of the XML document must be stored for future navigation, and a value index must be used for the special `GetElementsByTagName` access method. As discussed in chapter 5, there are several approaches of representing the XML structure in index. In example, static offset-addresses can be used as index target when the document is static and never changes its content. This is the fastest approach, because the data is compiled. A dynamic index will force usage of interpreted data, since the exact data location might not be known. On the other hand, the dynamic index have better support for changes in a XML document. An static index, aiming for static documents is chosen, making direct access faster.

The proposed index builds on principles from the non-extractive approach discussed in chapter 3.1.2. This approach has substantial benefits when the document appears as static. The approach makes it possible to refer to offsets in a data stream and use the data directly. This will however, not work for an implementation of the W3C DOM. But the same type of index can be used. The non-extractive approach maps the entire XML structure, and constructs references to offset address in the stream. When the structure later is accessed, the system always knows where different parts of the document are located. Using the non-extractive approach for indexing, a basic approach for indexing the structure of XML documents based on offsets are proposed:

For each node, the *start-offset*, *end-offset* and the *start-offset* of all its children is stored.

If the node is an element, the element name is also stored for faster retrieval. The value stored is not the string, but an integer value which corresponds to the element name in a symbol table. The symbol table is further discussed in chapter 6.2.2. All nodes in the entire XML document is not indexed, this would generate a huge amount of index entries, making it both large and slow. To reduce the amount of index entries, the granularity of the document has to be changed. This is issued in chapter 7.3.1.

In this chapter, XML example document 3 found in appendix C.3 is used. Figure 31 shows the object representation of the sample XML document.

7.3.1 XML Node Collapsing

The basic idea by indexing the start and end address of every node is that it will generate a huge number of index entries, one for each node in the tree. One file access will be required for every single node on retrieval. In traversing the entire tree, this will cause terrible performance issues. Most XML elements containing child elements often do not contain any data themselves, and in data-centric documents (as shown in chapter 2.6), data tuples rarely has much data.

To avoid a lot of source accesses for reading relative small portions of data, an index granularity based on the node-size has been selected, potentially encapsulating nodes as discussed in chapter 5.3.2. If a node is under a defined minimum size, this node and all sub nodes will not be indexed. The nodes are still reachable however, but since they

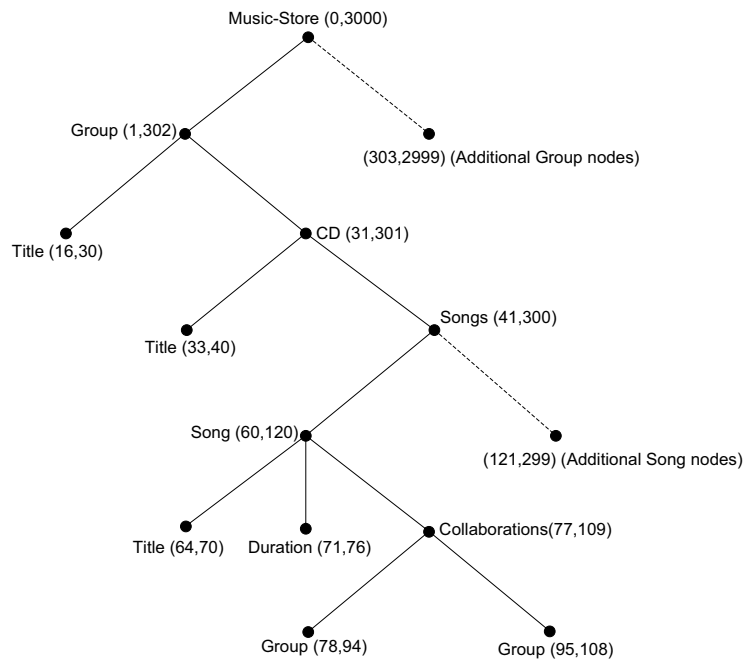


Figure 31: Sample XML node tree

are not indexed individually, they have to be parsed all at once.

The decision on the minimum size must be set at indexing time. A high minimum size will result in a smaller and faster index. In addition, the indexing itself will be faster due to fewer insertions of entries in the data storage.

In retrieving the node, the index must be aware of this technique as well. If the index detects a collapsed node, it will generate a D3P node tree from this and all its sub nodes. That is, in this small portion of the XML tree, an eager approach is used instead of demand driven approach, giving a performance penalty if only a very small portion of the partial generated tree is accessed. Due to the different accessibility of nodes, the nodes in both the structure and value index are divided into tree types:

Indexed Node: These nodes are directly reachable from index. When querying the structural index, a response containing navigation information is expected.

Blocked Node: These nodes are not directly reachable from index. But however, a blocked node is referred to by an indexed node. Logically, the parent has information regarding the start byte offset of the Blocked node.

Indirect Blocked Node: These nodes are not reachable from the index. They are contained inside a blocked node, meaning that for these nodes to be revealed, the blocked node must be parsed.

7.3.2 Symboltable

The concept of a symbol table is simple. It is basically an alias list where character data are associated with a numeric value. The reason usign a symbol table is to consume less memory space, with little performance overhead. Also, integer values are generally faster to compare than strings. The concept is the same as in Xerces2 (chapter 6.2), but D3P also uses the symbol table in indexing the XML document. The symbol table for the XML data represented in figure 31 in file is illustrated in figure 32. This implementation utilizes symbol table by using a combined hash table and array. In chapter 7.3.3 and 7.3.4, string names are used in examples for simplicity, but the implementation uses integers which are used with the symbol table.

Symbolist	
Character	Value
Music-Store	1
Group	2
Title	3
CD	4
Songs	5
Song	6
Duration	7
Collaborations	8

Figure 32: Symbol table

7.3.3 The Structure Index

XML documents are hierarchic structured, making navigation easy through the W3C DOM interfaces. Traditional implementation can reach every point of this structure by traversing through the node tree. In this Demand Driven Dom Parser, all nodes can not be retained in memory at once, hence an index is created to accommodate for the lack of in memory nodes. The index will use principles from non-extractive parsing (chapter 3.1.2) combined with Li/Moon indexing scheme (chapter 5.3.1) to determine relationships between nodes.

Description

The proposed index will store data containing four columns as shown in figure 33: *start*, *end*, *children* and *name*.

- The first column acts as the identifier in the index. It is also the byte offset address where the node starts in the XML document, hence it is unique for the file.
- The second column is the respective end address of the node.
- The child column consists of a list of all child nodes (start offset).
- The forth column consists of the name of the node. Note that in the implementation, the respectively integer value from the symbol table are stored in this column. If the node is not an element node, the integer value for this column is “-1”, meaning “not in use”. If the node is an element, but the element has attributes, the value is also “not in use”. This is due to that

attributes are not covered by the symbol table. When the “not in use” value is issued, the data has to be read from file, either because it is not an element or the element has attributes.

Index table			
Start	End	Children	Name
0	3000	1,303,606,....	Music-Store

↑ Where the node starts (INT) ↑ Where the node ends (INT) ↑ A list of childs nodes start addresses (INT []) ↑ The name of the node (INT)

Figure 33: Sample index table

Construction

The index construction is a “parse once, use many times” operation (discussed in chapter 3.1.2) and is required to make the implementation work. This is done with a custom XML parser (for technical specifications, see chapter 8.6).

Each time the parser encounters a node, it checks its size (end - start). If it is below the CollapseLimit, the parser simply do not index this node, making it unreachable. This way, less nodes are indexed, resulting in a higher granularity, as shown in chapter 7.3.1. The data are stored with the given format in the generic index storage discussed in chapter 8.5.

For example, the XML representation in figure 31 is to be indexed. Assume that the additional Group listings are at the addresses 303, 606, etc, and only has one CD entry each. Further the CollapseLimit is set to 300 bytes. The index would then have the structure shown in figure 34.

Index table			
Start	End	Children	Name
0	3000	1,303,606,....	Music-Store
1	302	16,31	Group
303	605	318,333	Group
606	900	621,639	Group

Figure 34: Resulting index table

For the sake of simplicity, only the first three Group nodes have been accounted for. The figure shows that only the “Music-Store” and “Group” nodes has been index. This is because the CollapseLimit is set to 300 bytes, and the only nodes that have a greater address range is these three nodes.

Retrieving

W3C DOM uses relations to navigate through the XML node tree. When retrieving



Figure 35: NodeList with no and one node visited

nodes by traversing, there is only a single entry point, namely the root node. As the index contains information about where the node start and who their children are, structural navigation in the node tree is possible.

When a node is requested, the byte offset of that node is used as an identifier. This identifier is used consulting the index. The index takes different actions according to whether the identifier was found in the index or not. If the index entry is **found**, the index returns the same data which was indexed for this node, namely the *start*, *end*, *children* and *name*. This information can be used to make the requested node directly. If the node is an element, and it has a value for the *name* column, the XML source do not have to be accessed at all.

If the index entry on the other hand is **not found**, the node has not been indexed since its size was considered too small at index time (smaller than the collapse limit). Additional parsing is required to build this node. Using the build-in .NET XML Pull Parser, the node starting at the byte offset (the identifier) and all its children are parsed as nodes of the D3P. As for sure, the length of the XML data under this node is lower or equal to CollapseLimit limit value.

If the identifier was found, the new node can be made. However, if this node is an element, the child nodes are not constructed directly. The fact that this identifier existed in index shows that this node is considered large (at least larger than the CollapseLimit set at indexing time), and it is not desirable to parse all children to memory. The children to this element will simply remain as integer values. That is, their byte offset in the file and identifier in the index. The NodeList in D3P is constructed in such a manner, that if a node in the NodeList are accessed, and this node exists in the form of its identifier, the NodeList makes the corresponding node and replaces it with the number. This node can in turn either exist or not exist in the index and are handled accordingly. On the left hand side in figure 35 a graphical representation of a NodeList with no child nodes visited is shown. On the right hand side, the same NodeList is shown, but the second node has been visited. Note that the first integer remains unchanged until visited. This ensures that only nodes which are requested are built.

To illustrate what will be built when retrieving a node, an example where a node is to be fetched is presented. The documents in figure 31 and 34 is used and the desired node is the "Group" located at address 78. The query for this node would in W3C DOM syntax be: *"Root.FirstChild. SecondChild⁶. SecondChild. FirstChild.*

⁶W3C DOM does not state a "secondChild" attribute, This is solely for logical traversal. In W3C, secondChild would have been: ChildNodes.Item(1)

SecondChild. FirstChild”.

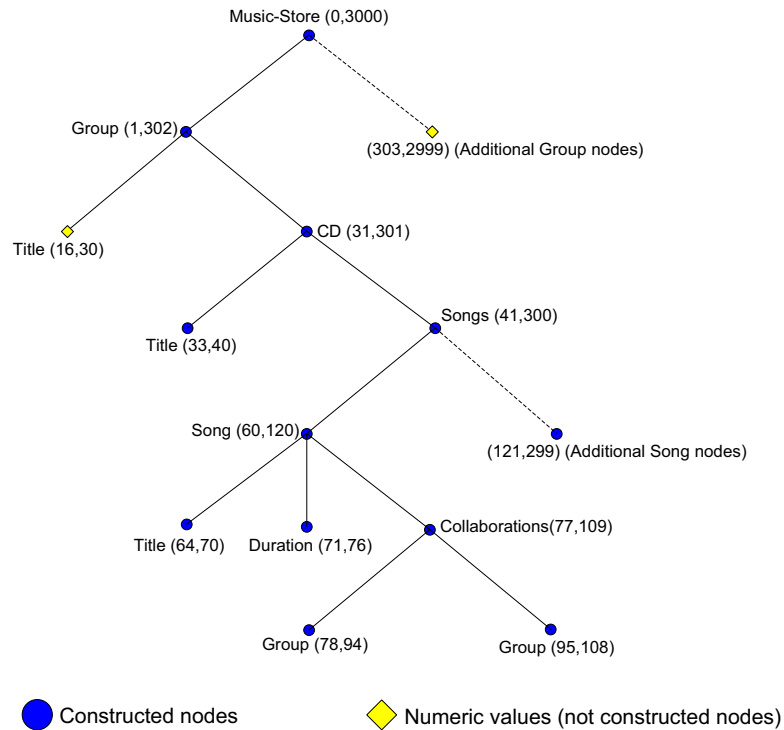


Figure 36: Partial constructed DOM tree

The actual constructed data (node tree) is illustrated in figure 36. Nodes are built as the nodes in the tree are visited. Only the first “Group” node is (partly) constructed, and only the child “CD” node is fully constructed. The process is as follows:

1. The entry point is established by constructing the root node (“Music-Store”) from the index, which only has numeric values as children. The Root node is not accessed by its identifier, since there is only one root node. This node is treated as special and is retrieved by the index separately.
2. The first node (id 1) of the child list is requested. It contains a numeric value and checks whether the identifier exists in index. As it turns out, the node exists (this is the “Group” node) in index and gets constructed in the similar manner as the root node.
3. The second node (id 31) of the child list is requested, and the response from the index is that this node does not exist. That means that this is a collapsed node and parsing is necessary to build the node.
4. The node (“CD” node, address 31) will be parsed and the entire node regardless of structure will be built. The resulting node is root node of a node subtree, which is attached to the real node tree. When this node is built, no more index or file accesses are necessary to reach the desired node.

Building sub tree of nodes also acts as a caching mechanism where nearby nodes are available for future access. For example, if the requested node is the “Title” element at address 64, it would already exist in the built node tree, and returned at once. The content of a collapsed node are either all in memory or nothing in memory. This has consequences when nodes are removed from memory. This will be further discussed in chapter 7.4.

7.3.4 The Value Index (Name Tag)

The W3C DOM states a function for retrieving elements based on their tag name through the “getElementsByTagName” method in Element and Document, with the actual name as main argument. This method returns a list of nodes (NodeList) with all the elements with this specific tag name existing within the context of the Element or Document. The resulting NodeList can contain nodes from various parts of the document. The list is ordered in order of occurrence in preorder traversal of the document tree.

The structural index can not perform this task without parsing the entire XML document, which would result in performance issues. Therefore a 2-level indexing scheme as discussed in chapter 5.3.4 has been implemented to accommodate the retrieval of nodes based on names. This means that an additional index for retrieving nodes which has a specific name, while the structural index in chapter 7.3.3 is utilized for retrieving the data used for navigating and constructing nodes.

The ability to access nodes directly provided an element name gives yet another entry point to the node tree. Both the W3C Document node and Element node has this function, but this implementation only supports the Document node version, retrieving all elements with a specific name in the entire XML document.

Description

The index can be described as a table consisting of two columns, *name* and *occurrences*. This is an inverted index, referring directly at identifiers of elements which has the specified tag name. The proposed structure of the index is shown in figure 37. The first column is the name column, holding each unique name, and act as the identifier. The second column consists of an array of byte offset locations, where the name occurs. Note that the offset address is the identifier in the structural index too. The location addresses accordingly refer to either an *indexed node* or a *blocked node*.

Since *indirect blocked* nodes are not reachable from the structure index, the *blocked nodes* are used as representation for these nodes. And since the structure of the blocked node is unknown, only a counter for how many tag names with the specific value exists inside this blocked node are used. The counter is necessary for the NodeList to insert Elements at the correct position.

In figure 37, the element named “Title” is associated with a list of occurrences. The first occurrence is identifier 56. This node is obvious a blocked node. It has a total of 10 occurrences of “Title” in its sub tree. The D3P library must know how many occurrences there are under these blocked nodes, due to correct retrieval according to the W3C DOM specification. If no counter for the blocked nodes was

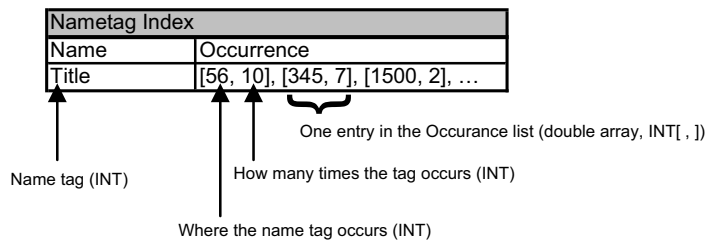


Figure 37: Tag name index, example entry

provided, the NodeList would not know the total length of the list nor where to correctly insert Elements in the list.

Construction

The tag name index is constructed simultaneously as the structure index. The tag name index consists of a hash table with the name as key and a sorted list as value, as shown in figure 38. A sorted list is basically a hash table that sorts automatically based on the key value when it is being inserted.

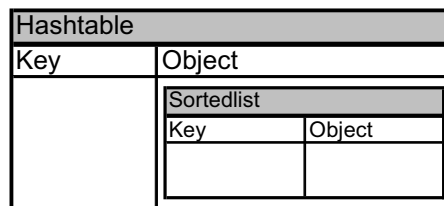


Figure 38: Tag name index construction

The key in the hash table is a unique tag name that resides within the XML document. Each time the parser encounters an element name, it checks if the name exists in the table, then either creates or add the occurrences in the appropriate manner.

The W3C DOM standard say that `getElementsByTagName` function return a `NodeList`, where the elements are encountered in a preorder traversal of the document tree. This is why a sorted list is used, sorted on node identifiers. The sorted list associated with the name key in the hash table, has entries of its own. It consists of the node offset address and a number indicating how many occurrences are present at this offset address. Note that for indexed nodes, this number is always one, because an indexed node does only contain itself. For a blocked node on the other hand, the number is one or higher. Parsing the sub tree of a blocked node will reveal these nodes. The occurrences are counted while parsing and is inserted as value to the appropriate address offset key in the sorted list. To elaborate, consider the following example: The elements in figure 31 shows nested elements with the specific name "Group".

For the sake of simplicity, only the first Group element and first Song element shown in figure 31 are accounted for in the index. Furthermore a CollapseLimit of 300 bytes is assumed. This would indicate that the “Group” element will be a blocked node in the structure index (see chapter 7.3.3). The tag name index for this scenario is shown in figure 39.

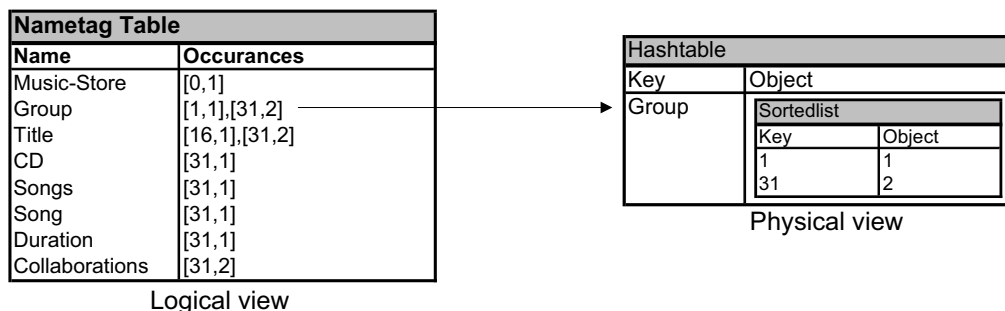


Figure 39: Resulting tag name index

The table in figure 39 means that “Music-Store” occurs once at address 0, “Group” occurs once at address 1 and twice at address 31, “Title” occurs once at address 16 and twice at address 31, etc. Note that all nodes under the element with identifier 31, uses 31 as its reference, due to that all these nodes are indirect blocked nodes. The only way to access these nodes are through constructing the entire sub tree under node with identifier 31.

Retrieving

To properly understand the retrieval process of nodes based on their tag names, a closer look at the NodeList implementation is required. This list is basically an array of objects (ArrayList), which means that the contents of the list may be any object. In this implementation, a listing can either be a D3P Node or a numeric value, as discussed in chapter 7.3.3. Also, a string is associated with the NodeList. This is basically a flag that indicates if it is a node list constructed by the “getElementsByTagName” method or not (if the string is empty the list has been constructed under normal traversal).

To illustrate what happens when retrieving a node list by tag name, consider the following example. Using illustrations from figure 31 and 39, the requested tag name is “Group”. First, the occurrences of the tag name is retrieved from index and a node list is constructed, as shown in figure 40.

The resulting NodeList has three entries, reflecting the index entry for the name “Group” in figure 40. The NodeList has one entry for address 1 and two entries for address 31, resulting in the three entries. Any other W3C DOM implementation would return this number of entries, but the D3P only contains the numeric values from the index. This is done to prevent any unnecessary disk accesses, not to mention the memory advantages (a numeric value consumes a lot less space than a constructed D3P node), and collocate with the principle of retrieving nodes only when it is requested.

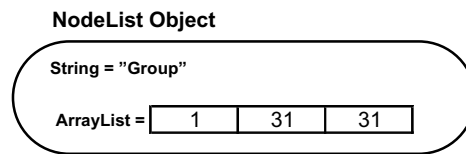


Figure 40: NodeList before creating nodes

When accessing the nodes in this NodeList, the methodology is different according to whether the accessed node is an *indexed node* or a *blocked node*. When a node is accessed, the integer value (identifier) is consulted in the index. If the node is found in the index, the new node is constructed from the information retrieved from the index and replaced at the proper position in the NodeList. This approach is similar to that discussed in chapter 7.3.3.

However, if the identifier is not found in the index, the entry is referring to a blocked node. In this case, there are one or more elements inside this blocked node which has a matching element name. The exact number of nodes is found by analyzing the node list. There might be identifiers both before and after the current entry holding the same identifier value. The total count of entries with the same identifiers are the number of Elements with the requested name existing in the sub tree of the blocked node. This is also the same number of occurrences which are indexed for a blocked node.

In the following example, the first node is retrieved from the node list in figure 40. The node with identifier 1 is found to hold an entry in the index, resulting in an easy construction of a new node. The node is replaced at the first position, resulting in the NodeList shown in figure 41.

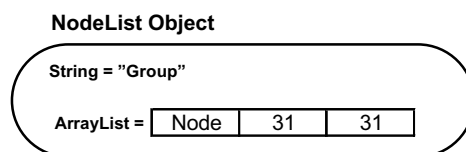


Figure 41: NodeList, single node created

When accessing third entry in the NodeList shown in figure 41, a more complex approach must be used:

1. The identifier is consulted in the index, but as shown in figure 34, the identifier (31) is not in the index, hence it is a *blocked node*.
2. The element starting at the address offset 31 is parsed and all elements in this sub tree matching the requested name are inserted to an array. The resulting

array has elements stored in preordered traversal of the document. In this case, two nodes are located.

3. In the NodeList, the first entry with the same identifier (31) is located, which in this case is entry number two. The array of nodes is replaced from this entry and ahead. The number of entries in the NodeList with the same identifier should match the number of nodes in the array, resulting in the NodeList in figure 42.

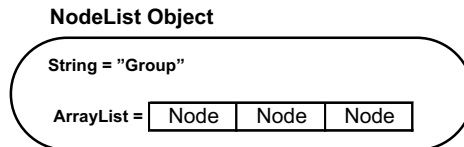


Figure 42: NodeList, multiple nodes created

The node list is now completely constructed. Notice that all the Elements within the same *blocked node* will be inserted in the NodeList when either is being accessed. When requesting the second entry in this list in figure 42, the node is now already constructed and would be returned at once.

In order to make the NodeList sorted in preorder traversal, the sorted list from the indexing is necessary. The NodeList needs to know the order in which nodes appear (beginning to end) in the file, so that it can insert nodes in the correct position in the NodeList.

7.4 Loading and Unloading XML Nodes - Lazy Loading

The principle of caching from operating systems shown in chapter 4.2.2 is used in this implementation of W3C DOM as well. In cases where the node tree would exceed the internal memory, caching must be used. That is, holding only a small portion of the data in memory at a time. There are two basic approaches how to do this:

Use OS paging

The Operating System (OS) have a working caching method by loading chunks of data (pages) to disk when memory is full. The XML node tree can be huge, but only portions of the data is in memory. There are however inconvenience by this approach. The operating system does not take the XML structure in to concern. When the OS flushes a page to disk to release memory it only takes into account whether the page has been visited recently, with no concern that parts of this page can be a part of a larger context. The OS also has limited swapping space on disc, setting the limitation of the size of XML documents which can be loaded. This space is shared with other applications running on the computer. In addition, the process of writing data to disc is a resource demanding process, potential to strangle other processes.

Manually Unload Nodes

By manually unloading nodes in the D3P library, full control over which nodes are in memory at a time is obtained. For example, all parent nodes of the nodes last recently used nodes are more likely to be referred to sooner than others. That is, when the node tree is traversed, the parent node is required to move to either previous or next sibling. By holding the control in the D3P library, a more intelligent caching algorithm can be utilized.

By continuous sorting nodes in memory after the LRU algorithm (chapter 4.3), the system knows at all times which nodes are due to be removed from memory. However there are technical limitations of implementing this at a node basis. A software LRU algorithm is generally made out of some sort of ordered list. As nodes are added to the top of the list, the time of searching this list is increasing, gradually, to the point of extremely bad performance. Therefore an LRU algorithm which registers nodes or a collection of nodes have been implemented, which makes up a larger block of data. This way, there will be fewer entries in the LRU list and more memory will be released for each LRU operation. However, there must be some sense in selecting the minimum size for a block which is inserted in the LRU list. The size of the node collection handled by the algorithm are dependent on the chose strategy cache structure discussed in chapter 7.4.2 and 7.4.3.

In addition, nodes are never moved in a sorted list as the principle of LRU indicates. A timestamp is set on each collection of nodes when it is used. This approach was discussed in chapter 4.3.3. The implemented LRU algorithm rather checks the list for the lowest timestamp when nodes has to be removed from memory.

7.4.1 Cache Algorithm for XML Structure

Cache algorithms for linear data structures are well known and widely used in operating system. A file on the hard drive is for the operation system always linear data. XML is

logically tree-structured, so the principle of caching must be altered to fit this structure. The following weak statement are proposed as basis for the cache algorithms in this thesis. They build on the principle of locality discussed in chapter 4.3:

- If a node is referred to, the node and its children will tend to be referred to again soon.
- If a node is referred to, its parent tend to be referred to again soon.

In other words: parent and children of currently visited nodes are more likely to be addressed soon.

All test implementations of caching in this thesis uses timestamp for identification of the least recently used nodes. As discussed in the beginning of this chapter, moving objects in a LRU list is expensive in terms of performance. With the use of timestamp, objects never gets moved around, but their timestamp is updated. When a candidate for replacement must be found, the entire list is searched for the active object with the least timestamp.

Two approaches are proposed. The first emulates the operating systems virtual memory paging, dividing data into ranges, making no concern of parent-children relationships. The latter takes concern of parent-children relationship by never removing parent nodes to nodes recently used. These are discussed in chapter 7.4.2 and 7.4.3 respectively.

7.4.2 Linear Structured Cache

This structure gives no regards to child-parent or sibling relationships. The data area is divided into static intervals with a fixed size. This means that data is divided in the same manner as if it was linear data. This method is foolproof, but when traversing a tree, a parent node which is dismissed from memory are forced to be recreated when moving up in the tree.

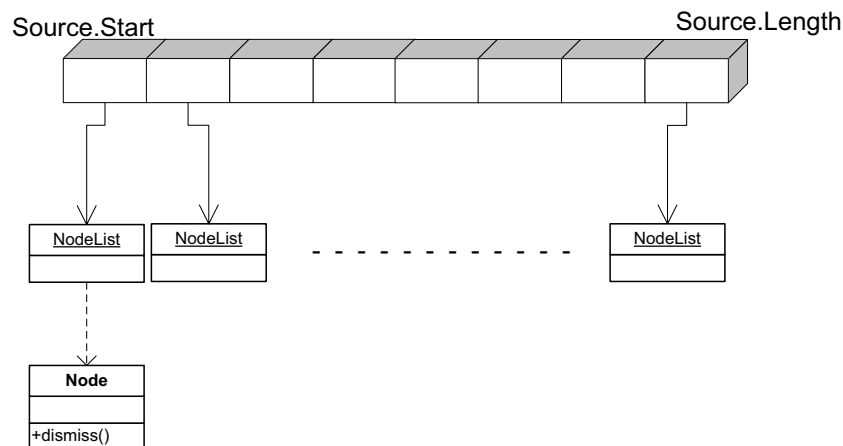


Figure 43: Static linear structured cache

The number of intervals, n , must be predefined when making the cache. The data-size in each interval will be (XML data length / n). An estimate of serialized XML versus XML in memory using D3P nodes is a magnitude of eight. (this value alters for how the XML document is structured) This means that in each interval there resides approximately M units of memory (equation 1).

$$M = \frac{XMLData.Length}{n} * 8 \quad (1)$$

When an interval is selected for removal, all nodes which have identifiers (offset address) inside this interval are removed from memory. Hence, for the least recent node to still be retained in memory there must be at least two intervals in which there are nodes. This way, it is certain that the interval with the recently used node is not the one selected for removal. To ensure this, the maximum memory usage allowed must be over M (the amount of data in a node). For example, if an XML document is 400 MB and the maximum desired memory usage is 10 MB, the number of intervals will be 320 as shown in equation 2.

$$N = \frac{XMLData.Length}{M} * 8 = \frac{400E^6B}{10E^6B} * 8 = 320 \quad (2)$$

7.4.3 Tree Structured Cache

This structure takes consideration regarding child-parent relationship of nodes most recently used. When a node is accessed, the node's timestamp is updated. This timestamp traverses upwards in the node tree, so that all ancestors of the current node have the same timestamp as the current used node. When selecting the least recently used node, the algorithm starts on top, traversing the tree. If a node has more than one node in its nodelist, the node which has the lowest timestamp is dismissed. The node with the highest timestamp is obvious the same as the current most recently used node, hence a node with this timestamp is on the path to the MRU⁷ node, and should not be dismissed.

It takes less time and resources to dismiss a collection of nodes than one at a time, therefore the first least recently used node found when traversing from top of node tree is dismissed. In figure 44, the next node to be dismissed is node B, and with this, all its children. If the node tree is traversed in preorder traversal, the tree structured cache will always remove nodes in postorder traversal.

7.4.4 Behavior of the Cache Algorithms

There are basically only one way of tuning the algorithms. This is altering how much memory to be released when the maximum cache memory limit is reached.

In the *static cache*, an interval of data represents how much data that potential to be released from memory at once. In the *tree cache algorithm*, no control is attained over how much memory is being removed at a time, only how many nodes. The latter approach is potential to spend time removing only insignificant amounts of memory at a time. This is the case for very wide XML structures.

⁷MRU means Most Recently Used. It is the opposite of LRU and refers to the node which is on top of the LRU algorithm stack

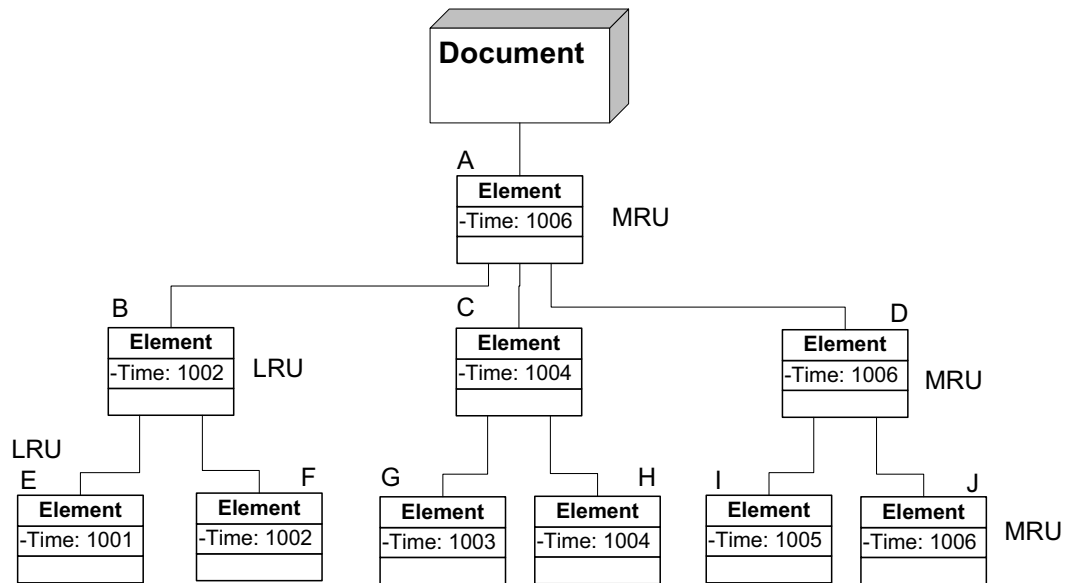


Figure 44: Cached nodes in the tree

The special “GetElementsByTagName” function discussed in chapter 7.3.4, does not traverse the node tree starting at the root node, and nodes can be scattered located around the XML node tree. In this particular case, the *tree cache algorithm* will potentially use many resources constructing parent and ancestor nodes to set the timestamp, even when the ancestors are never to be used. In this case, the *static cache* can make a better and faster decision of nodes to remove from memory.

7.5 Utilizing the Managed Memory Model

The essence in demand driven parsing of a XML object representation is the construction of nodes as they are requested. Using the principle of demand paging and swapping discussed in chapter 4.5, nodes have to be removed from memory to make space for demanded nodes. The removal of objects from memory is the job of the Garbage Collection (GC) as discussed in chapter 4.6. The GC removes objects not referred to, but in the case of a hierarchical structure like the W3C DOM node tree, all or nothing of the tree is referred to. It is however possible, to make the GC clean the selected objects.

The nodes to be removed from memory must be located before the Garbage Collector is issued. Then, all references to these nodes must be removed. This way, when the GC runs its collector, these objects (D3P nodes) will be considered garbage and deleted from memory.

The problem however, is that the system has to know in advance that the heap is nearly full and the GC is about to be issued. If the GC runs its collector before the system has deleted sufficient nodes, there might not be enough size on the heap for future nodes to be loaded, resulting in a full heap and the application will break.

Therefore, this implementation has a additional thread, continuous checking the size of the heap. A maximum allowed amount of memory can be set, which must be under the maximum heap size. When the thread finds the heap size to be over this limit, the cache algorithm is issued, finding and removing references to these node(s) due to specifications of the algorithm. The next time the GC run, all objects not reachable from the root of the program or from the current context will be removed from memory. Hence the GC has to run **after** the removal of references to the node. The best practice advice is to let GC run itself whenever it finds this necessary, but the GC has to be run immediately after dismissing nodes from the node tree to release memory.

This approach is not straight forward. If a client using this implementation has a reference to the Node, this reference will not be deleted from memory. The system will remove all references from parent, child and siblings, but the clients reference will still exist. The client using this object can try to navigate the node tree through it, but all references in all navigational directions have been removed, due to the attempt of disposing the node. This means that each node has to have the knowledge of how to rebuild references to parent and child nodes. This is done by replacing the node object back to the identifier integer value which represents the offset address of the node. For example, a node in the tree has a parent node. The parent node is selected for removal, so all references to it have to be removed. The node referring to this parent, replaces its reference pointer by an integer representing the identifier (byte offset) of the parent node. This way, if the node later tries to access its parent, the node is made by using the identifier as discussed in chapter 7.3.3. The same approach is performed with child nodes.

A node which still exists, but are not in the node tree is called a *disconnected node*. The user might have a reference to this node. If the user navigates the node tree again to the same node, the user should have two references to the same node, not two different

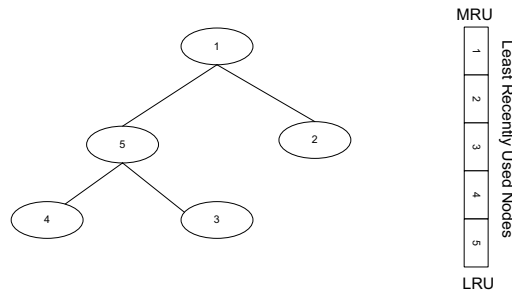


Figure 45: Node tree before cleaning

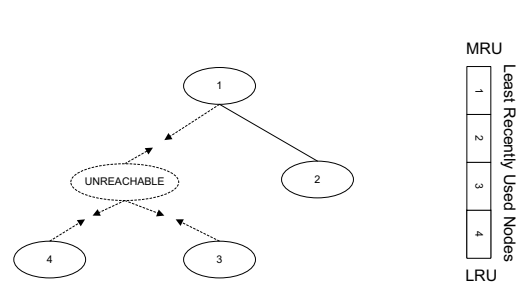


Figure 46: Node tree after cleaning

ones. This means that the system must know which nodes a user still has a reference to, so when navigating the tree to a node which is referred to elsewhere, the reference to the same node must be returned.

To solve this problem, a function in the managed memory model is used, namely *weak references*. A weak reference is a reference to an object which might or might not be valid. When a garbage collector is issued, weak references are not taken into concern. That is, when the weak reference is the only reference to a node, the object is considered garbage. In this implementation, there is a list of all D3P nodes which are created, all referred to by weak references. Always, before making a new node, this list is consulted. If the weaker reference is valid, the garbage collector has either not been issued since the node was disconnected or the user has a reference to the node outside the node tree. In any case, the reference to this node is returned.

Figure 45 and 46 shows how references are removed from nodes, making the node unreachable from the rest of the tree. As shown in figure 46 there are still a indirect reference in sense of integer identifiers, meaning the reference can be rebuilt if needed.

7.6 Updating the XML Document - Lazy Approach

When updating the XML document, there are many considerations to take. Basically, the XML document is only a text file, which is fairly easy to both edit and create. Traditional W3C DOM implementations all have write support for writing altered XML documents back to file. This is not an issue, since all data is in memory; it is only converted to string and flushed to file. The D3P prototype however, does not have every node in the node tree in memory. The easy approach is to traverse the tree as usual, writing it back as it goes, resulting in a lot of unnecessary I/O and object creation. Data which are not altered from the old files do not have to be converted from text to nodes and back to text data. It is already in text, stored in the XML document.

The changes in the XML object representation is not reflected on the file immediately, lazy write back [21] is used. In terms of basic lazy write back techniques, the data is marked as dirty when it is retained in cache, but written back to storage when it is flushed from cache. But if the file was to be rewritten every time a dirty node was removed from cache, performance could be an issue. As the number of altered nodes

increases, too many resources will be spent rewriting the file. Hence in this implementation, the file is not rewritten until the user gives the command for this.

This implementation makes an update log of every change made to the tree. That is, insertion of nodes, deleting nodes and altering nodes. The update log simply holds information about which node is altered, that nodes ID and if the altering concerns the node *itself*, *its child nodes* or *both*.

Insert: A node with a unique, negative ID in the signed 32 bit integer address space, is inserted to the NodeList. The owner Element of this NodeList is added to the update log. The entry is marked with “children”.

Delete: The node is simply removed from the NodeList and the owner Element is added to the update log, marked with “children”.

Attributes insert, removed or modified: The owner Element is added to the update log, marked with “self”.

If an Element is added to the update log and already exists in the log with the same mark, no action is performed. If it exists, but with the opposite mark (in example, want to set a Element marked as “children” when it already exists in the update log marked with “self”), the entry is marked with “both”.

In the W3C DOM API, a nodes name cannot be altered. If a node name is to be redefined, the node has to be removed and a new node with the desired name has to be added in its place [13]. Any new inserted nodes in the tree, has to be inserted into some already existing NodeList, hence new nodes are never added to the update list, only the owner of the NodeList it is inserted into.

The entries in the update log are sorted due to its ID, which actually is the character offset in the source XML document. This means in other words that the entire file can be copied from the source XML file, besides the nodes which can be found in the update log. These nodes have to be handled in another manner. The mark in the update log is checked, writing only what has been changed in this node. When a node is marked as “children”, each of the children has to be handled individually. The result will be a new XML document, with merged results from the original source XML document and entries in the update log.

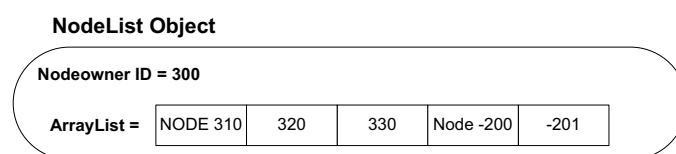


Figure 47: Altered nodeList

The NodeList in figure 47 shows five nodes as children of an Element. The first node is in memory, because a node is in its place. It has a positive ID, which means it is

referred to in the memory. The second and third node is either not made yet or has been dismissed from memory cache, due to low usage in recent time. The fourth entry consists of a new node which reside in memory, and the fifth is a node which is new, but has been dismissed from memory. This latter node has to be stored temporarily when it is removed from memory cache, because it obviously cannot be found in the source XML document. The ID of this node is used in registering the temporary node, so that the same node can be brought back to life if it later is referred to. As further discussed in *further work* in chapter 13, this prototype does not write dirty nodes to disk when they are removed from memory cache, as should have been done in a final implementation. Dirty nodes which are removed from memory cache are still retained in another repository in memory. This means that as the number of dirty nodes increases, more and more of the memory cache will be used holding these nodes.

8 Implementation

8.1 Component Overview

In the D3P library, each module is shown separately. Implementations of the W3C DOM interfaces are not shown in this thesis, due to that the behavior of these classes are stated by W3C, and can be found at [14].

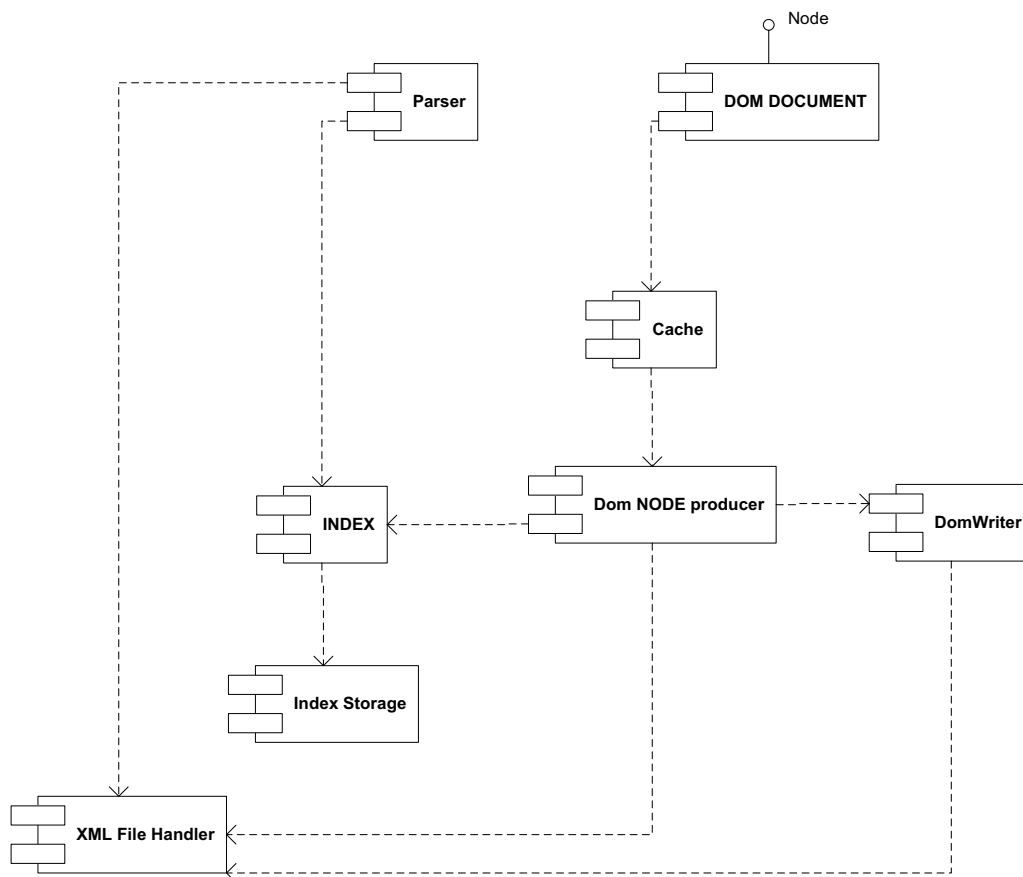


Figure 48: Component overview

The XmlDataSource, XmlIndexStorage and the cacheAlgProvider all use a hybrid between a factory and a singleton pattern⁸. This is due to the available configuration settings in the library. The settings are set as global variables in a static class called Constants. The values in the file are default values, set at compile time, but they can be altered for the session in the GUI, the command line tool or through the API.

To get a better overview of the application, the component diagram in figure 48 is

⁸Patterns (Design Patterns) is a common method to exchange knowledge of specific program design problem. Singleton states that only one instance occurs. Factory states that some object is constructed due to a factor (such as a input parameter)

presented. Each component is further discussed throughout the chapters. An UML sequence diagram is available in appendix D. This diagram shows a scenario where a node is accessed, and how the different components interact to get the desired node.

8.2 Cache

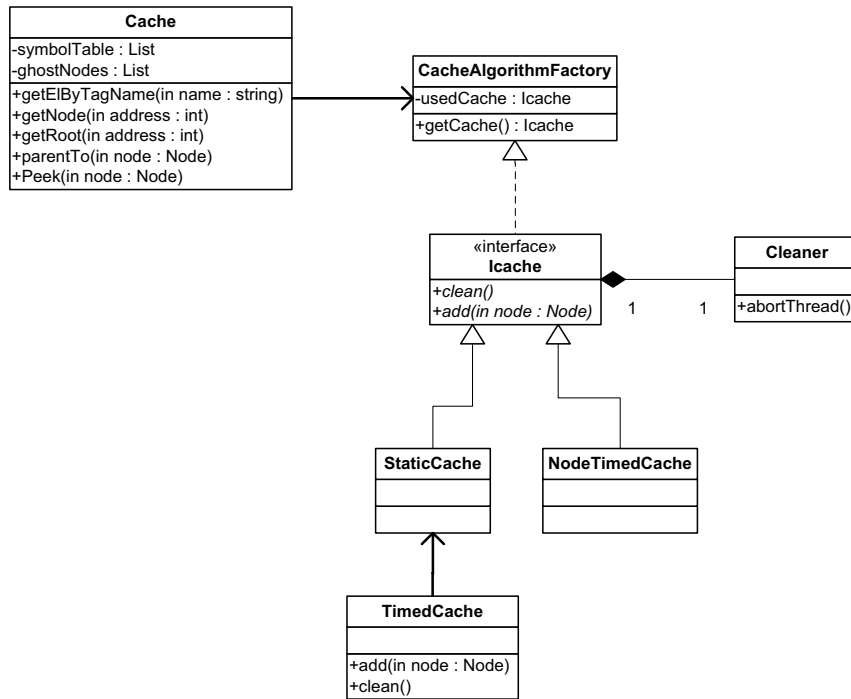


Figure 49: Class Diagram: Cache

The Cache class controls which nodes who are in memory at any given time. It is responsible for registering constructed nodes, and make a decision which nodes who are due to be removed from memory. The Cache possesses a symbolTable, a ICache and a list over Ghostnodes:

symbolTable of type List

Holds all element names used in the XML data to make traversing use less memory. When the symbolTable is used and an element does not have attributes, the element does not require any XML source access to build its node. But however, attributes are not in the symbol table, and elements in possession of attributes must be parsed for retrieval of attributes. Whether an element has attributes or not is known at indexing time, and registered in the structural index. This approach is slower than assigning a string to a node-object, but more memory efficient.

cacheAlg of type ICache

Holds all the existing nodes which reside in the node-tree. Which algorithm who is being used is stated by the configuration, and the correct implementation of ICache is returned by the CacheAlgorithmFactory. The available cache algorithms is presented in chapter 7.4.1.

```

if node with this address exist in ghostNode list
  get node from ghostNode list
end if
else
  make this node from XML source
  register node in ghostNode list
end else

return the node

```

Figure 50: pseudo code for consulting weak references

ghostNode

Holds a weak reference⁹ of all D3P nodes who are created. This gives a second chance of holding a reference to a node who was not collected as garbage by the GC. For example, a scenario where a user holds a node, which the LRU algorithm dismisses as the next to be removed from memory, this node will be removed from the tree, but not from memory. If this node is visited again by navigating the node tree, the nodes reference will be returned from the ghostNode-List rather than making a new object for this node. The pseudo code for this part in figure 50.

Cleaner instantiated in cacheAlg

The Cleaner class is responsible for checking the current memory usage. If the memory consumption is over the desired limit stated in the configuration, the Cleaner activates the selected ICache implementation in cacheAlg, using a visitor pattern¹⁰. The Cleaner will continue to activate the ICache implementation until the memory consumption has decreased under the desired limit. The pseudo code for this operation is shown in figure 51. In addition, an activity diagram is provided for the Cleaner. This can be found in appendix a:uml.

```

while abort value is not set
  while memory used is less than memory Allowed To Use
    Find least recently used node or bunch of nodes
    Remove links to the nodes
    Remove from memory all nodes which have no links to it
    Update memory usage value
  end while
  wait a small period of time
end while

```

Figure 51: pseudo code for the node cleaner

8.3 DomProducer

Domproducer is the class which is responsible for making D3P node from the XML source. It is dependent on the IndexDataStorage and the XmlDataReader.

⁹Weak references are discussed in chapter 7.5

¹⁰The visitor design pattern is a approach to separate an algorithm from an object structure. The cleaner visits the LRU algorithm which are registered.

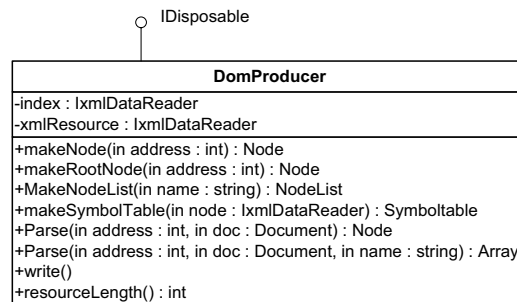


Figure 52: Class diagram: DomProducer

As mentioned in chapter 7.3.1, nodes who are smaller than a defined size are considered too small to be indexed, and becomes a *blocked node* or an *indirect blocked node*. In the function DomParser:makeNode(node Node) the decision must be made, whether to parse the entire node or make the node directly from index data. This is show in the pseudo code in figure 53.

```

if node address exist in index
  if node is element
    if element has attributes (known from index)
      read start tag from source
      make a new node with name from source
      make attributes and append to new node
    end if
    else
      make new node with name from symbol table
    end else
  end if
  else make respective node
end if
else
  Get data stream from data source
  Use MS XmlTextReader to make this node and children from the
  data stream
end else

return the newly constructred node

```

Figure 53: Pseudo code for constructing nodes

When making a node which resides in index, index information is actively used to reduce parsing from the XML data source. If the index provides symbol table information, it is preferred to use this value prior to parsing from source. The symbol table gives only information for the nodes of type element. Other nodes, such as text nodes, comments and processing instructions, must be parsed from the XML data source. The index gives however, not detailed information about attributes. It only says if attributes are present or not. If attributes are present, data must be parsed from XML data source. The symbol

table makes it possible to never parse XML data for *indexed nodes* (see chapter 7.3.1 for the different node types) which do not have attributes. When making nodes which are part of a *blocked node*, a stream is provided to Microsoft's own XML Pull parser (XmlTextReader) included in the .NET framework. The nodes made from this stream are D3P nodes as well.

8.4 XML Data Source

The XML Data Source component is responsible for reading XML data from a XML source. In this implementation, two approaches are available, a reader which reads directly from XML source file (XmlDataReader), and a reader which uses a memory stream (XMLDataBufferReader) to buffer the whole source file. This latter approach will require memory of same size as the XML document, and will hence be equal to VTD-XML in chapter 6.3 regarding initial memory requirements. Both approaches implements the *IXmlDataReader* interface and the static *fileStorageProvider* returns the implementation defined by the configuration.

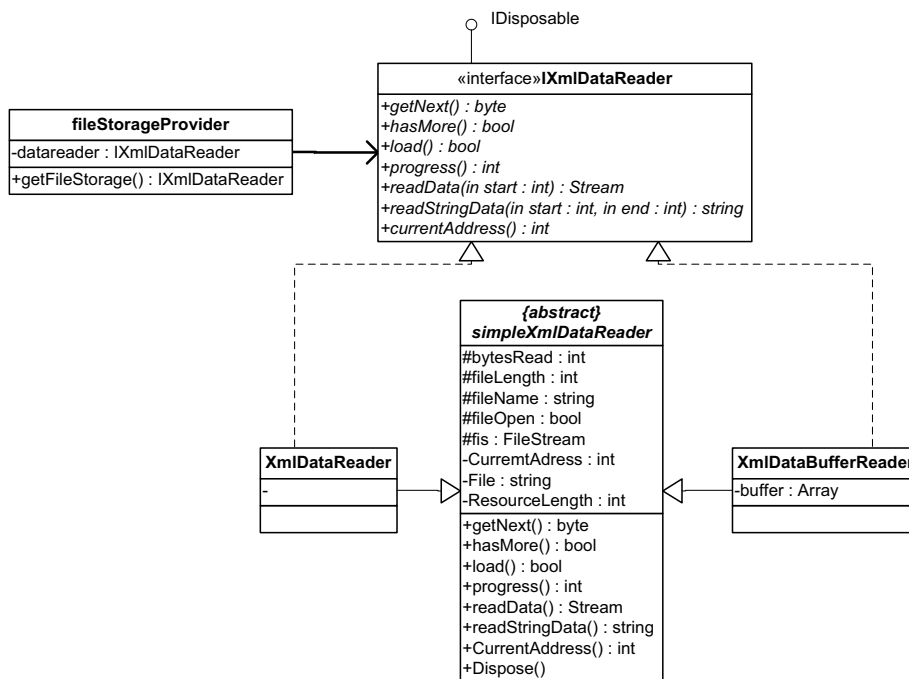


Figure 54: Class diagram: XML data source

Both these readers only support 8-bits ASCII (same as UTF-8 in US/English). This is because the parser always have to be in control of how much data is read at any given time. The number of character read prior to an node is that nodes start offset in the XML document. By using 8-bits ASCII coding only, one character will always be exactly one byte.

8.5 XML Index Storage

The XML Index Storage is responsible for holding the information regarding the structure of the XML document. How this is done, is abstracted through the IndexDataStorage interface. Currently, two approaches are tested: using an embedded object database, and using memory structures for holding index data.

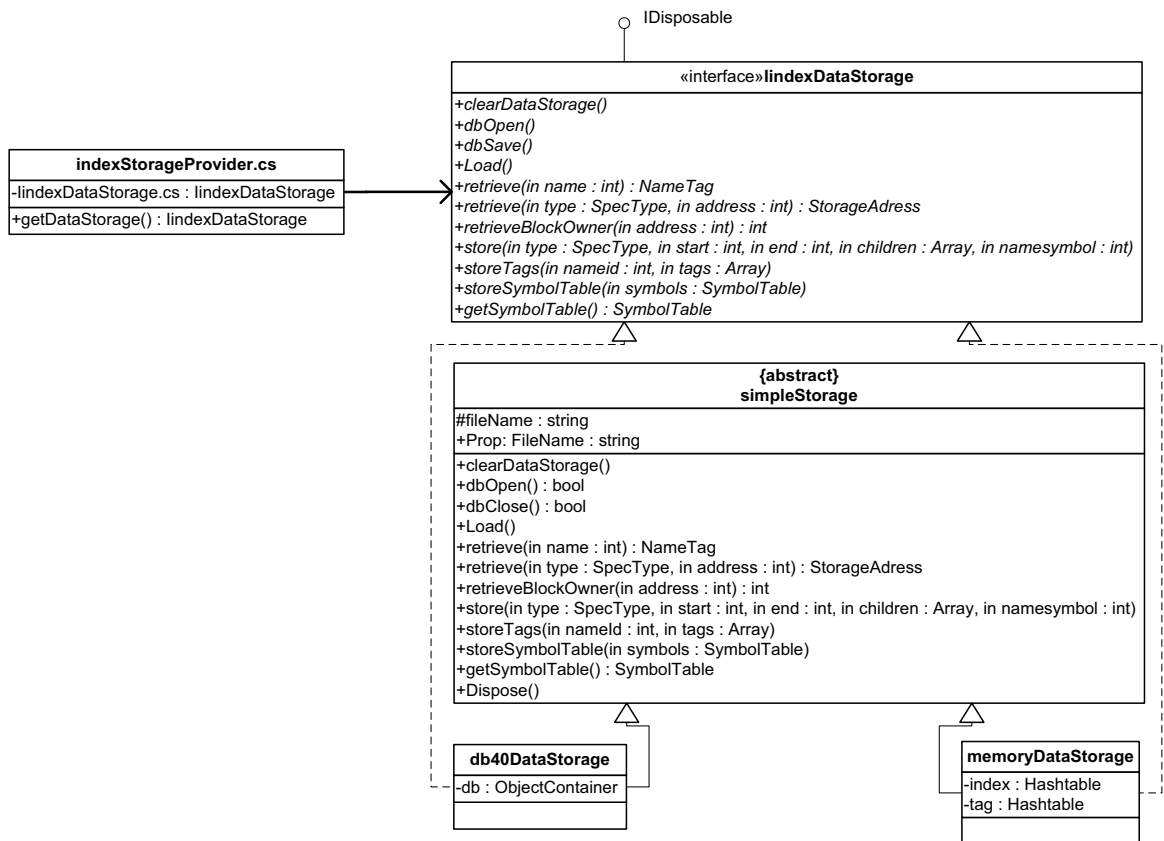


Figure 55: Class diagram: XML index storage

The Embedded database (Db4O datastorage [41]) runs in its own thread and stores data in a database file. In this matter, the data is available next time the database file is opened. The embedded database used ,DB4O, has memory caching and intern index therefor making the process very fast. The memory data structure uses a Hashtable from the .NET library, storing objects in memory. After parsing the structure of the XML document, the index data is serialized to disk. When a XML document is loaded for usage in D3P, the data has to be de serialized back into memory.s

8.6 Parser

The parser is responsible for pre processing the XML document so it can be used with the D3P library. The parser maps the entire XML document and extracts both information for the structural as well at the tag name value index discussed in chapter 7.3.3 and 7.3.4. The entries which for the structural index are flushed in postorder traversal. There is no way of knowing the size of a node until both the start and end tag is found.

If the size is over the pre defined limit (the CollapseLimit configuration option), the node is indexed with the address as key.

This parser is implemented for the specific purpose for supporting D3P, hence it is not completely developed to support namespace, Xlink and PCDATA-tags since neither the D3P library is implemented to support this.

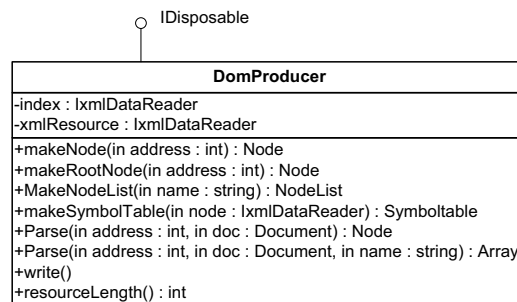


Figure 56: Class diagram: Parser

This parser basically does the same as any other SAX parser, with one crucial difference. This parser can get the exact start and end offset in the data source. No other SAX parser in C# or java is known to have an API with this ability. The code which controls the offset address when searching for start and end tags are shown in pseudo code in figure 57. An overview of the parser functionality is available as UML activity diagram in appendix D.

8.7 XmlWriter

This class is the specific implementation of the approach discussed in chapter 7.6. Changes are collected in an unordered list in the DomWriter as a temporary repository. When the write()-command is issued, this list is sorted and a log for updating is made. The nodes in the updateLog is marked with a tag, showing what data is altered in the node. It can be the node itself (which basically means its content, element name or element attributes), its children if the node is an element (stated by the NodeList attached to the element) or both.

This gives the opportunity to only rewrite the part of the node which is altered. If the state of an Element or Attribute is altered, only the start tag and attributes has to be written from D3P nodes, all of the children can be directly copied from the XML source file, releasing D3P from much processing. The children do not even have to be constructed. When a node is added or removed from the NodeList, the owner of the NodeList is registered as an altered node. Figure 59 shows the behaviour of the write function in pseudo code.

IMPLEMENTATION

```
while still data available on source
  get next character
  until "<" is found
    skip forward
  end until
  remember Start Address

  until ">" is found
    append to string
  end until
  remember End address

  send string to analysis
  if string is startTag
    if stack is not empty
      peek object at top of stack
      append myself as child to object
    end if
    make new object
    append Start Address to object
    push new object on stack
  end if
  else if string is endTag
    pop object of stack
    append End Address to object
    send object to index
  end else if
  else if string is end tag at once
    peek object at top of stack
    append myself as child to object
    make new object
    append Start and End Address to object
    send object to index
  end else if
  ... else if ... // other nodetypes
end while
```

Figure 57: Pseudo code for initial parsing of XML

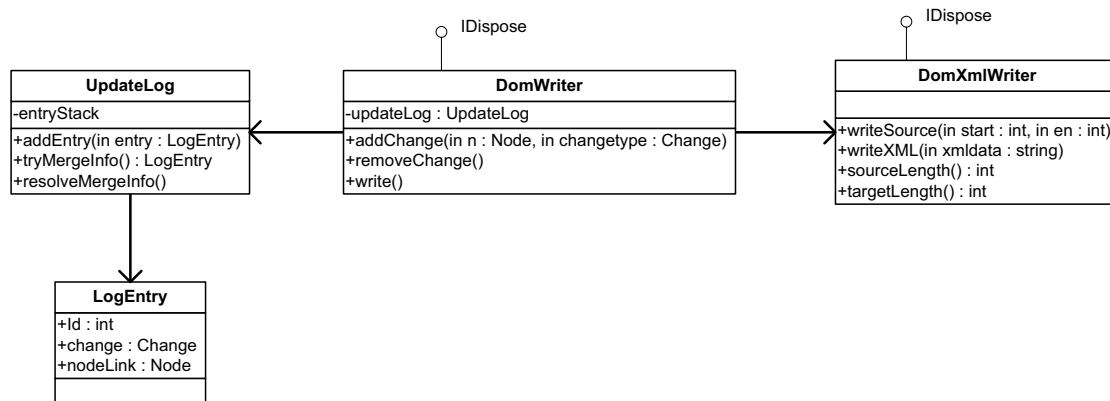


Figure 58: Class Diagram: DomWriter

```

function writer (StartAddress, EndAddress)

  Get next LogEntry in UpdateLog
  if LogEntry is greater than EndAddress
    copy source to target from StartAddress to EndAddress to target file
  else
    Rapport LogEntry as successfully fetched
    if LogEntry.Change is a Block
      write entire XML block from LogEntry.NodeLink
    end if
  else
    copy data from Start address to LogEntry.Start
    if LogEntry.Change is Self or Both
      write new element start tag
    end if
    else if LogEntry.Change is Children or Both
      foreach node in child list
        if node is new
          write new node from LogEntry.NodeLink
        end if
        else
          writer(node.Start , node.End )
        end else
      end foreach
    end else if
  end function writer
  
```

Figure 59: Pseudo code for writing back XML

9 Configuration

9.1 Configuration Tool

The D3P includes a graphical user interface and a command line tool which can be used for setting configuration parameters and index the XML document. Both run on the .NET platform and are supplied with the library. Indexing can be done on-the-fly when loading the XML document to the D3P at runtime, or it can be done in advance using the index tab shown in figure 60 or the command line tool shown in figure 61. It is recommended that the XML document is indexed in advance to minimize delay in the application using D3P. Once the file is indexed, the index can be used for as many sessions as desired, as long as the source XML document is not altered. This states a ‘parse once, use many times’ situation for the index.

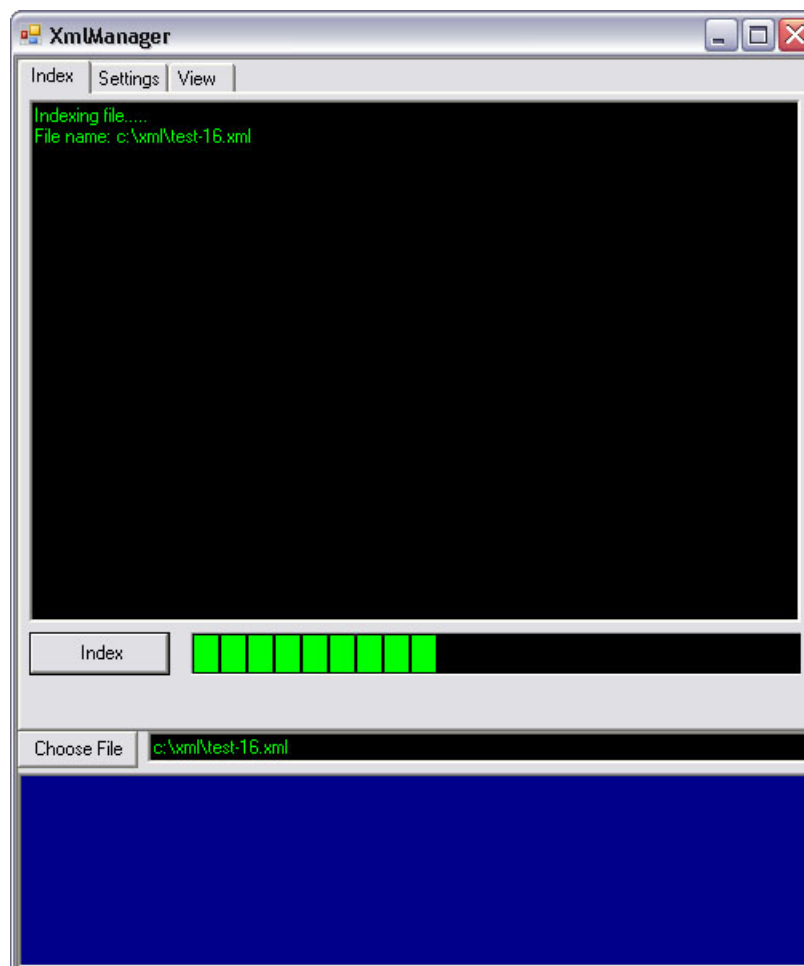
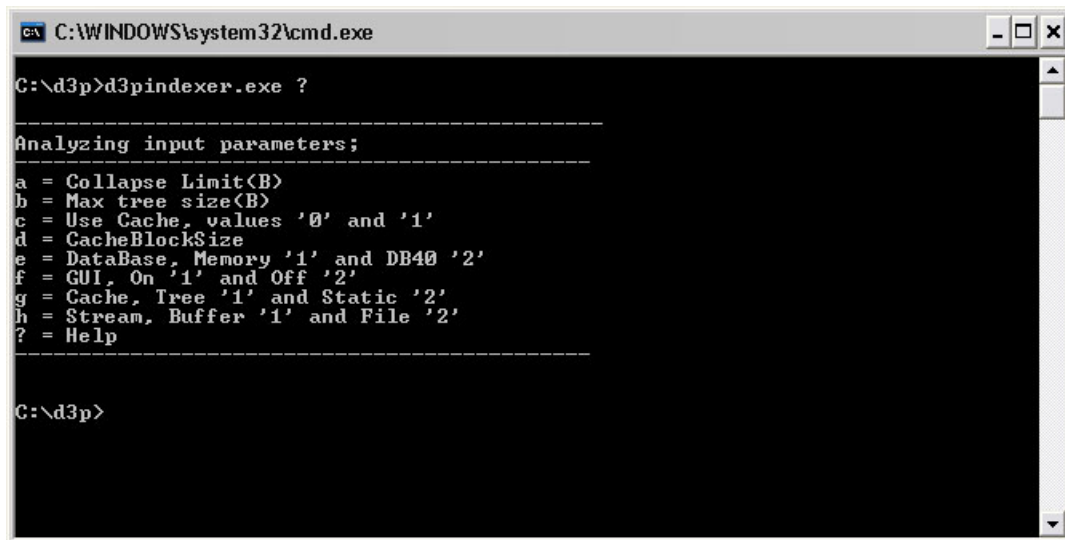


Figure 60: Indexing tool, index tab

There are a number of settings for the indexing, shown in figure 62. Some of these settings can have critical outcome on performance, hence they should be set with care. A simple introduction to the options are presented. For technical specifications, please



```
C:\WINDOWS\system32\cmd.exe
C:\d3p>d3pindexer.exe ?
-----
Analyzing input parameters;
-----
a = Collapse Limit(B)
b = Max tree size(B)
c = Use Cache, values '0' and '1'
d = CacheBlockSize
e = DataBase, Memory '1' and DB40 '2'
f = GUI, On '1' and Off '2'
g = Cache, Tree '1' and Static '2'
h = Stream, Buffer '1' and File '2'
? = Help
-----
C:\d3p>
```

Figure 61: Indexing command line tool

refer to chapter 7 and 8. Further in this chapter, these options and their impact the overall performance is addressed. When using the console or GUI application to index or alter settings, a file with same name as the XML file in question but with the extension *.settings* is stored in the same directory as the XML file. The file containing the both indexes (structural and value) and the symbol table is stored in a file with the same name, but with a *.dat* extension. This file is located in the same directory as well.

9.2 CollapseLimit

The CollapseLimit option is used to select the minimum size of nodes which shall reside in index. A higher value will result in faster parsing and a smaller index. On the other hand, all nodes which are not indexed will be regarded as blocked nodes (as discussed in chapter 7.3.1 and constructed as complete D3P sub trees if accessed. The user can adjust this by how the XML document is to be used. The CollapseLimit simply set the level of granularity to be used in the index and cache. This has impact on performance, as this value affect both the cache behavior and numbers of file accesses when navigating.

Consider a scenario where an XML document has 10 sub nodes to its root node. The file is 50 MB of size. That is, it is estimated for each sub node to be 5 MB, where each sub node of these again has 1000 nodes, each with an estimated node size of 5 bytes. In indexing this particular document, the impact of selecting too high or low CollapseLimit might be crucial.

High CollapseLimit

Choosing a high CollapseLimit will result in a smaller index and lower index construction time, but more nodes have to be parsed as sub trees. The cache will have less control over the memory consumption as it only can dismiss *indexed nodes* and whole *blocked nodes*. However the cache will dismiss more nodes at the time, requiring less work for the cache algorithm.

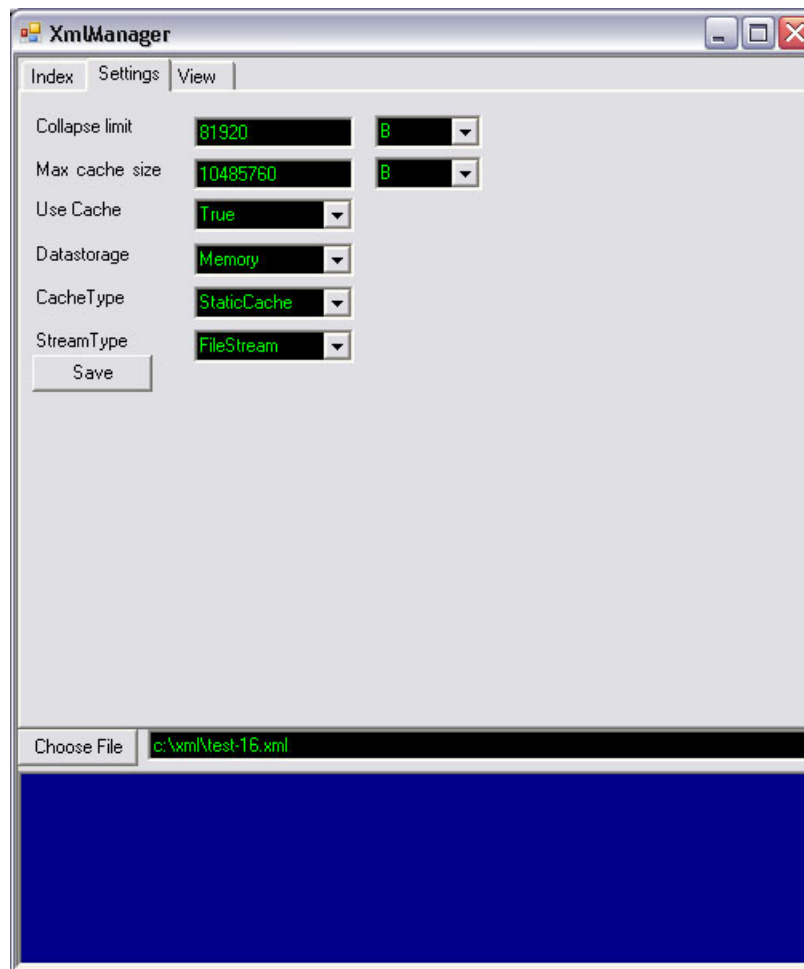


Figure 62: Indexing tool, settings tab

In the presented scenario, a high CollapseLimit over 5 MB, would result in an index only containing one *indexed nodes*, namely the root node, which would contain 10 *blocked nodes*. In this case, the cache can only load and unload entire 5 MB of XML data blocks at a time, resulting in an estimated minimum memory usage of 40 MB (the estimated memory object representation is 8 times the raw XML data, 5 MB * 8 = 40 MB). At the same time, the high CollapseLimit ensures a low number of source accesses, and therefore a low overhead in comparison to traditional approaches.

The advantages with a high CollapseLimit, aside from a smaller index and reduced index construction time, is when whole *blocked nodes* are accessed. This will result in fewer accesses to the source XML source document. The disadvantage is when only a small portion (*Indirect Blocked Node*) of the XML document is to be accessed (1 off the 1000 sub nodes), as this would require to build the entire *blocked node* (1001 nodes), consequently requiring more processing and memory space.

Low CollapseLimit

Choosing a low CollapseLimit will produce a large index, but less nodes have to be completely parsed as D3P sub trees. Also the time spent indexing the file is increased due to more entries to be inserted in the index. The cache would have more control over the memory consumption, but the cache would have more entries and more work to do.

In the presented scenario, a low CollapseLimit under 5 MB would result in an index containing 11 *indexed nodes* and 10000 *blocked nodes*. In this case, the cache can load and unload potentially only 5 bytes of XML data blocks at a time, resulting in an estimated minimum memory usage of 40 bytes.

The advantage of having a low CollapseLimit is when small parts document is accessed. This implies that small nodes are represented as *Blocked Nodes* in the index. This decreases the chance of having to parse and build unnecessary nodes, saving both time and memory space. The disadvantage is if many nodes are accessed, requiring many accesses to the XML source document. Estimated, an entire tree traversal would require 10011 file accesses, making the application a performance bottleneck in any system.

9.3 Max Cache Size

The *Max Cache Size* value is the maximum amount of memory the total application is allowed to use. This includes nodes, index, buffers, temporary object and other data that is required by the D3P library. It is important to select a reasonable amount of memory. For example, when using the XmlDataBufferReader as data source (see chapter 9.6, the total amount of allowed memory can not be below the size of the XML document. Also the Max Cache Size has connections to the CollapseLimit value, discussed in chapter 9.2. Basically, the Max Cache Size must be at least a magnitude of the CollapseLimit. If a chosen CollapseLimit results in blocked nodes of memory size larger than the Max Cache Size, the node will be attempted to be removed once it enters the memory.

The performance hit of choosing size of the tree is very dependent on how the D3P library is used. Consider a scenario where only 10% of a XML document is being used and the Max Cache Size is set to accommodate these 10%. When adjusting this value further, this will happen:

Higher cache size

A higher value for the cache size would not affect the performance, because the memory consumption would never exceed the max cache size and the nodes would always be available in memory.

Lower cache size

If the max tree size was to be lowered, the Cache will have to dismiss the least recently used nodes from memory. And when dismissed nodes are requested they have to be rebuilt, resulting in a decreased performance.

The rule of thumb is that a higher value is better, as long as the system can handle it, as this lower value the chance of nodes being dismissed and rebuilt.

9.4 Use Cache

This option turns on and off the caching in D3P. Without the cache activated, all nodes will be retained in memory. With the caching turned on, nodes which is least recently used will be removed from memory when the limit of *Max Cache Size* is reached. In practice, setting the “use cache” option to off will have the same effect as setting the *Max Cache Size* to infinite (or Int32.MAX).

9.5 Data Storage

This option selects the Data Storage for the index to be used. As for now, one can choose between the db40 Open source Database [41], and a in memory index based on the .NET Hashtable class. When much data is to be indexed, the embedded database is preferred, due to higher memory usage in the in memory index. The in memory index is serialized to disk when the session ends.

Memory Based Index

Choosing a memory index would always be faster than a disk based index however, this comes at a cost of more memory consumption. The structural index is often queried when navigating (and building). A memory index would not require a lot of disk accesses, making this an favorable option.

Disk Based Index

Choosing a disk based index would be slower than a memory based index, but has the advantage of consuming a lot less memory for many index entries. The downside is that the structural index would require a lot of disk access even for simple navigation purposes, making the memory based index favorable. However, the value based index behaves somewhat differently than the structural index. This index only requires one access for each value query (*getElementsByTagName*), and uses the structural index for additional building. The performance difference would be minimal of using memory or disk based value index.

This implementation has either both indexes (value and structure) in memory or in the embedded database.

9.6 Xml Reader

This option selects between direct access to file and full buffer to memory before usage. Copying the file to memory will use considerable more memory. A memory buffer is expected to always be faster than a file on disk. However this advantage is minimal, and only under special circumstances is the buffered approach expected to be significant faster. This happens only when the collapse limit is set low and traversing the node tree, resulting in many disk accesses (I/O operations). The penalty for having a buffer is that it occupies a lot of memory, something the D3P implantation would like to avoid.

9.7 Cache Algorithm

Selects what cache algorithm is to be used in D3P. Currently, a *Static Cache* with pre sized buckets and a *Tree Based Cache* which takes parent-child considerations are available.

Static Cache

The static cache takes no consideration to relationships and dismisses nodes based on their addresses within a predefined interval. This can potentially lead to relevant nodes (parents, siblings, children) often being dismissed from memory and rebuilt when traversing the node tree. When retrieving nodes based on their name (using `getElementsByTagName`), the nodes do not require being nearby related, making the static cache a favorable choice.

Tree Cache

The tree based cache would normally be favorable when traversing. The reason is that the *Tree Cache* takes node relationships into consideration, making it a bit more intelligent than the *Static Cache*. This cache will not dismiss parents of last recently used nodes, thus minimizing the chance of relevant nodes being dismissed from memory, making the tree cache well suited when dealing with hierarchical structured data.

Part III

Evaluation and Discussion

In the former chapters 4 5 and 6 we have been discovering features of lazy processing. The technology and research supported our work in developing a demand driven DOM parser (D3P). A parser making it possible to adjust the memory consumption to be used by the implementation. This would in theory make it efficient when using large XML documents. This implementation of the W3C DOM, is during the next chapter tested with several input files and scenarios. It is designed to look at the D3P pros and cons. Later on we will address these results in the discussing part of the thesis, chapter 11.

10 Testing

When using DOM in an application, questions regarding performance come naturally. The navigation, retrieval and handling of the XML should not take too long or consume huge amounts of memory. Tests like [47, 48, 49] are performed simply to show which product is to be preferred. In particular, implementations of the W3C DOM in java, all share the same interfaces, and many of the parsers are free (licensed under GPL or similarly), making it easy to switch to a “better” parser as requested. This makes it important to show how a W3C DOM implementation performs compared to other implementations. In this chapter, we will test the performance of the D3P compared to other parsers. However, the D3P is no traditional parser. To lower the memory consumption, some other resource has to suffer. In this case, this is I/O and CPU time. In this chapter, we test the implementation not only with acknowledged techniques, but we also introduce some additional tests to show the strength and weakness of Demand Driven Dom Parser (D3P). This chapter start with a simple introduction to software performance testing, moving on to existing DOM performance testing frameworks. Before testing the D3P performance compared to other, internal tests are performed, finding the optimal configuration for the D3P to be used in further testing.

10.1 Software Performance Testing

Software Testing is a crucial part of software development. There are several methods for performing testing, depending on the actual intentions of the test. Unit testing is primarily for functionality testing, that is, to verify that a component/class does what it is suppose to do. The W3C, for instance has many Conformance Test Suites for their recommendations. This is to verify that the implementation supports all functionality of the recommendation. Performance testing is used as another common evaluation criteria. Performance testing has been defined as: **In software engineering, performance testing is testing that is performed to determine how fast some aspect of a system performs under a particular workload.** In this thesis, it is desired to see how the solution performs compared to existing products. XML products strongly depend on high performance in every context of use, making this kind of comparison important. The computer has a number of resources, which all have limitations. These resources are [46]:

CPU time

A measure for the time the application spend in the Central Processing Unit (CPU). A fast CPU makes the calculations faster than slower. A badly designed and written application is potential for producing binary code which makes more calculations than needed, making the application use more CPU time.

Main Memory size

Represents the capability of the internal memory, RAM (Random Access Memory). Everything that needs to be fed to the CPU has to reside in RAM, thus all programs running, has some data in memory. Main Memory is much faster than secondary storage, so it is always desired to keep as much of the currently running programs in memory at any given time. Many running programs or low amount of hardware RAM, causes less available memory.

Secondary Storage (I/O)

I/O¹¹ operations does not only have limitations in its size, but also its speed. This is the slowest resource. The start-up cost when reading a new stream from a hard drive is high, so when reading data, as much as possible should be read at once for less disk accesses. The disk drive is only one of many types of I/O (In-Out) devices used by a computer to store or communicate data. Most I/O devices have performance issues compared with the internal RAM and CPU.

The goal of every performance demanding program will after this be: An application that uses minimal calculations, minimal memory and reads everything it needs from the hard drive at once, results in the fastest runtime. In every task the computer makes, some or all of these resources are used. If one of the resources is empty, that is, there are no more of that resource available, that resource is a bottleneck, because the other resources can still deliver higher performance. It is however, crucial to set test conditions to be similar to the expected actual use. This is often difficult to arrange, because of the variety in pattern of usage.

10.2 Testing DOM Performance

Several tests has been developed and performed to determine which W3C DOM implementation is the best [47, 48, 49], both independent and testing done to emphasize the producers own product. The latter type of testing is often done in the owner's interest, choosing test scenarios where the specific product is superior. To get an independent test result of the D3P, most tests are adopted by independent sources. The particular test in [47] concentrated on these factors:

Document Build Time

Measures the time spend to build a document representation from a text document. The definition of a document representation is a bit vague. Some parses will build entire object models in memory, while others can do preparations, making the file ready to use.

Tree Walk Time

Measures the time spend to traverse the constructed document representation. When traversing, all data is visited: Elements, Attributes and Text nodes, all preorder traversal from the document root element. When starting the traversal, the document is already considered "built".

Modification Time

Measures the time spend traversing the whole document in preordered traversal doing modifications along the way, and write the document back to persistent storage (hard disk). The time does not include the build time of the document.

Text Generation Time

Measures the time spend to get the content of the entire node tree. That is, for an unchanged node tree, the text generation output will be more or less equal to the original XML document. The DOM Parser can format the output in various format however, resulting in slightly various output files. The time does not include the build time of the document.

¹¹I/O (In - Out) represents kommunications between CPU/Memory and som external unit

Document Memory Usage

Shows the memory consumption the complete program has before and after the complete traversal of the DOM node tree. The memory before walk is also the initial memory used after Document Build. This memory consumption includes the entire application. This means that if the DOM library has memory consuming components, these will also be taken into account.

Serialization Out time, In time and Size

Measures the time spend to serialize, de serialize and the memory used for this representation. This is not a part of the W3C DOM recommendation; consequently not all parsers support this.

Also, other tests use some or all of these criteria's when running performance testing. In [36], where lazy XML processing is the issue, the walk time and memory consumption are measured for how much of the document is covered. In result, this test is constructed to accentuate some particular aspect of the DOM implementation.

The test of DOM implementations in [47] does not feed the parser from hard drive, but uses a stream from memory. Using memory buffers to avoid any external timing variables is a common approach of isolating the test to just concern the implementation itself. Also [47] is using a number of tests as data basis. Tests are run ten times, and an average result is calculated, except when large files are used.

10.3 Test Harness

The implementation of DOM described in this thesis, is intended to be used with large files, typically in the size of 10 - 300 MB. XML documents are not the ideal data storage when the file size reaches upwards, hence most DOM parsers are not intended to be used with very large XML documents. The tests described in chapter 10.2 all uses test file with size of magnitude from 400 bytes to 1400 KB. The latter is regarded as a very large file by the testers.

The basic advice on reading large XML documents is: Do not do it! Rather use a pull parser, SAX or an XML database. The basic idea behind this thesis is to make large XML documents available through the W3C DOM interface without the drain of memory resource usually experienced when using other established W3C DOM implementations; hence some tests are made as a special case, to show particular aspects of this implementation, both with it's pros and cons.

10.3.1 Compared W3C DOM Parsers

This implementation is done under the Microsoft .Net framework V2.0. Most other W3C DOM parsers are implemented to run under the Java VM. In fact, no other alternative W3C DOM implementation running on .Net framework could be found. This means that comparisons have to include implementations in other programming languages as well, to get a basis for discussion. Java parses is used because this framework has an architecture and memory handling similar to MS Net's. The parses to be tested in this thesis are:

- Xerces2 (Java), from now on referred to as Xerces

TestFile Number	XML structure	Attributes	Depth	Text Length	Size (MB)
Test: 1-6	flat	0	3	20 - 180B	0.5 - 330
Test: 7-12	deep	0	12-22	20B	0.5 - 390
Test: 13-18	medium	8	5-8	20B	0.5 - 303
Test: 19-24	medium	0	5	20 - 15KB	0.5 - 345

Table 2: Test files

- Xerces Deferred mode (Java)
- MS .Net library XML parser programmed in C#. From this point referred to simply as C# XML DOM.
- D3P (programmed in C#.net)

However, in preliminary test rounds, Dom4J, JDOM and Crimson (all discussed in chapter 6.4) was a part of the test harness, but it was unveiled that these implementations did not contribute to any new findings, either compared to others or themselves. Xerces was chosen to represent all of these parsers. This choice was done due to that Xerces also has a deferred mode, which utilizes on demand traversal and other memory saving techniques.

The Xerces Deferred mode is the Xerces parser with the deferred mode turned on. Deferred mode results in partial building of the document representation as the node tree is traversed. This is also the basics of D3P, consequently D3P and the C# XML DOM can be compared in the same manner. The results is partially but not completely comparable because of language differences. And for correct testresults, comparisons within the same programming language and platform is preferred.

10.3.2 Test Files

The XML structure has the ability to have a number of features, which can have impact on performance results. Some of these are:

- Number of children under each element (XML structure).
- Number of levels (depth).
- Number of attributes per elements.
- Number of bytes in text nodes and/or PCDATA.

To keep control over the features of the documents, 24 documents have been produced for the soul purpose of this evaluation. The 24 XML documents are divided into four groups, each with respect to one of the features above. The files are shown in table 2 Each of the four groups have six files, with a size from 500 KB to just over 300 MB.

Specifications of each test file are available in appendix F.1. All files are considered relatively simple. They have neither namespace, PCDATA, Xlink or DTD schemas attached to them. In the manner the documents are constructed, deep documents will have a smaller amount of actual XML data per MB than flat files. This is because the D3P

parser does not support these operations. For each level, white spaces are added in front of the nodes to make the document appear “pretty”. This is a common approach for XML output, hence this is also done with these files. The files are made by a “random” XML generator made for the sole purpose of making these files. The generator takes input parameters for how the file is to be constructed in sense of children, attributes, text size and element name. When constructing elements, the tag name is selected from a collection of strings, making the same element name appear random in the document.

10.3.3 The Tests

Most tests described here use the tests from [47] as a basis to test the performance of the demand driven approaches used in D3P. However, some test in [47] can not be carried out, because some features of D3P are not fully implemented. The test to be carried out as described in [47] is the *Tree Walk Time* and the *Document Memory Usage* test. The latter test is altered to contain document memory usage before and after walk. This tests measures the memory consumption of the object representation before actual usage of the node tree and after total traversal of the tree. The tests are referred to as *general tests* and are issued for all 24 test files described in chapter 10.3.2.

Serialization is not implemented in the D3P, so all test regarding serialization are skipped. The *Text Generation Time* is dismissed due to the unrealistic scenario of outputting entire XML text for the file size D3P is constructed for.

In addition to these tests, three other tests are issued. These tests shows scenarios where the differences for demand driven versus traditional W3C DOM implementations are easier revealed:

Partial Traversal Test

Measures the time and consumed memory for some part of the document to be accessed. The partial test traverses every N node in the tree in preorder traversal, regardless of level. The resulting test will access approximately a 1/3 of all nodes in the node tree. The test is only issued for test file 16.

Partial Traversal Test Repeat

Measures the time and memory for a specific assembled part of the document. This test will traverse, in preorder traversal 1/10 of a document 10 times. In total, only 10% of the document is ever accessed. For simplicity, file number 16 is used with this test. The root node has 10 sub nodes of equal size, making it easy to control that only 10% of the document is accessed. D3P is configured to have a cache large enough to keep this part in memory without having to remove nodes from memory as traversed. The test is only issued for test file 16.

GetElementsByTagNameTime

The `GetElementsByTagName` command in W3C’s Document Object Model returns a nodelist with every node matching the exact name, in preorder traversal of the document. This can result in holding elements from various parts of the document. The test measures the time spend to issue this command for a known element name, return the node list and access each node in it. This special test is only issued for file number 16. The test runs the D3P DOM parser under several

settings in terms of CollapseLimit. This is due to that for any element with the specific name inside a blocked node, the entire blocked node has to be parsed, and D3P nodes constructed. The CollapseLimits are:

- 1 B
- 1 KB
- 10 KB
- 80 KB
- 1 MB

The element name of the document node is used as criteria in the `getElements-ByTagName`. The method is run from the Document Node because it is not implemented on Element level. A total number of 5081 elements are expected in the returned nodelist.

RemoveTest

Measures the time spent and memory consumed removing every second node from the root element of a specific document (test file 16). Measures are taken for the build, removal of nodes and the writing of the modified node tree to XML. File 16 is known to have ten sub nodes to the document root. This test should remove five of them and write the document back. The size of the modified file should be approximately half of the original file.

Document Walk Time and Memory, Percentage Coverage

Measures the time spent to reach a percentage coverage of the document when traversing in preorder traversal. Reading the document from the start, time and memory is measured for every 10% of the file traversed. For simplicity, a file with 10 sub nodes of the document root is used. Each sub node has the same length. This test measures the total time and memory at end of every sub node read. I.e. when the third sub node is traversed 30% of the document is covered. The initial parsing and index construction is not included for D3P in this test.

10.3.4 Test Technique

Test computer is a Dell Inspiron 9300 Windows XP Service Pack 2 PC with the specifications:

- CPU: Intel Dothan 1.87 Ghz.
- RAM: DDR2, 533Mhz Dual Channel, 1024 MB.
- Harddrive: 5400 RPM, 8 MB cache.

Framework libraries used in the test are sun java 1.5 and .Net framework 2.0. All tests are run on the Windows operation system. This is due to the fact that the Microsoft .NET framework only runs on this operation system. Another .NET library implementation, *mono* [45] exists, and runs on various operation systems, but the XML parser performance on this implementation is known to have performance problems compared with the Microsoft Implementation. When the tests were run, no other processing, I/O or memory consuming application was running. This was necessary for reducing uncertainty in the test results. Below, an overview of how measurements are reached is shown:

Measuring Times

Time measuring is done by using the systems timer. A timestamp is collected both before and after the test, and the difference is the test result. That is, the time taken to perform the operation.

Measuring Memory Consumption

Memory measurement are done before and after the test, and the difference is the test result. To make the test “fair” for all the implementations, the garbage collector is issued before reading the memory consumption.

Profiling tools gives the opportunity to get the time spent in each routine in the application. However, profiling takes time and gives unrealistic results compared to regular runtime. Therefore, profiling was rejected as a possible alternative to get accurate timing and memory measurements.

The *Document Build Test* described in chapter 10.2 refer to the building process as a bit vague. The D3P has an indexing process prior to usage of the node tree. But the index process is done once for a static document and never again. On document load, the index is loaded and used. It can be questioned whether or not the indexing time is a part of the document build process, therefore in the test presented later in this thesis, both approaches are presented. That is, both D3P without indexing referred to as *D3P No Parse* and D3P with indexing simply referred to as *D3P*.

D3P and *D3P No Parse* is expected to differ in all measurements concerning time, but when measuring consumed memory, there is no difference between the two. After all, the parsing and indexing is not a part of the document access, and the index will be of same size. Hence, in all memory measurements, the test result is simply referred to as *D3P*.

During internal testing of the D3P, it was observed that the Microsoft .NET framework buffers data internally. When multiple tests were performed in the same run, test results became unrealistic, because it seemed like data could be fetched from secondary storage extremely fast. Since Java tests did not behave like this, it was decided to run tests without any pre cached document data. That is, each test was run once for every binary, and all tests was run in a sequence. All binaries were run once without recording test results, due to the JIT¹² compilation both for Java and .NET framework.

10.4 Test Preparations and Configuration

This implementation states a number of configuration options which can affect the test result. Therefore, a test unveiling which approaches are generally the most performance efficient in the sense of both timing and memory consumption. There are four components the user can affect in selecting the specific approach to be used. The four components are:

Index Storage: In memory or embedded object database.

XML Data Reader: Filestream from disk or stream from buffered data in memory.

¹²Just In Time compilation. Frameworks like Java and .NET utilizes script files compiled at runtime to be optimized for the current OS and hardware configuration.

Cache Algorithm: Static or adaptive to the Node Tree.

Memory cache size Can be set from 1 byte to infinitely.

The three first tests have been run on 4 files, with the parser being tuned to show the characteristics of the components, but not necessary with the configuration that is the fastest. Complete test results can be found in appendix F. The last test is used to find the performance influence of the memory cache size. This test will unveil the consequence of having a low memory cache versus a high memory cache for D3P nodes.

10.4.1 Index Storage

In terms of *indexstorage* the choice is between the *embedded-* and the *memory-database* option. In the embedded object database the data is mainly stored on the hard drive, using less memory when the index grows big. On the other hand the latter only accesses the hard drive on opening and closing the index, serializing and deserializing data. The frequently invokes on the database data causes the DB4O to be relatively slow compared to the in memory approach. It also have a larger startup overhead in terms of time. The test in appendix F.2 reveals that the DB4O is approximately 35 times slower on small files. The only drawback of the in memory database is a small memory overhead, as shown in appendix F.2. But this is not of notice relative to the total memory consumption of the application.

The test shows that the memory database is considerable faster than the embedded object database. The object database uses more load time than the in memory storage. Although the in memory index approach will use more memory, the number of entries in the index would normally be too few to be essential, hence the memory index will be selected for further testing.

10.4.2 XML Data Reader

The test results in appendix F.2 shows that performance differences can be considerable between the two readers. This is a result of how many data reader accesses who are issued. In test 6 and 7, the low *CollapseLimit* causes many file accesses for the *FileReader*. For the *FileStream*, this is no issue. In test 8 and 9, where the *CollapseLimit* is raised, resulting in less individual file accesses and more resources spent parsing and making D3P nodes. Here, the result for total traversal are approximately equal. The build-time (initial time before using the DOM tree) is always lower for the *FileReader*, caused by the *BufferedReader* which always copies the entire XML data to memory.

In the example of test 6 and 7, where the “user” set an unfavorable *CollapseLimit*, the result is very high walk time for both approaches. In test 8 and 9, where *collapseLimit* is set reasonable, walk time is lower for both of them. Considering the fact that the buffered method uses additional memory the size of the XML source, *FileReader* is considered to be the most favorable as when the *CollapseLimit* is reasonable.

10.4.3 Cache Algorithm

The test results in appendix F.2 shows the difference in performance as the allowed memory consumption varies. The dynamic tree cache outperforms the static cache in

all situations. Also the performance is better when the allowed memory consumption decreases. A possible reason for this performance gain is that the TreeCache gets less objects to compare to in the LRU algorithm when there are fewer nodes present. As the node tree size increases, more time will be spent finding the right node to release. In the future test harness, the dynamic tree cache will be used. For the tests which utilize the `getElementByTagName` function, the static cache is used. This command makes a list of nodes from different parts of the node tree. If these tests are used with tree cache, all ancestors of the nodes in the list is also constructed, resulting in a unnecessary high memory and performance overhead

10.4.4 Cache Size Impact on Performance

This test is constructed to get an estimate of the performance impact of the memory cache size. As described in chapter 9.3, the memory cache size sets the total amount of memory the application is allowed to use and therefore sets constrains how much of the node tree can be kept in memory. In this test, test file 5 is used. This file is indexed with a 5 MB `CollapseLimit`. The cache configurations settings for the cache size are as follows:

- D3P-1: 1 MB cache size.
- D3P-2: 100 MB cache size.
- D3P-3: No caching restriction is used. The DOM tree is build partially, but all nodes are kept in memory, as with unlimited cache size. The implementation will work in the same manner as Xerces Deferred.

Since the `CollapseLimit` of test file 5 is 5 MB, the memory cache limit of 1 MB in D3P-1 is unrealistic, and the memory consumption will obviously not reach to the level of 1 MB. This is done solely to stress the algorithm to always try to remove nodes from memory. Note that the node tree is traverse, so no nodes leaved by postorder traversal is ever referred to again.

Figure 63 and 64 shows the measured results. The graphical representation shows that with the cache turned off, the least amount of time spent are attained. Note that for setting D3P-2, the time spent are the same until the memory limit of 100 MB is reached. In situations where nodes have to be removed from memory for the traversing to proceed, some extra time is spent. The conclusion is that the penalty for using a low memory cache size on traversal is not overwhelming, hence low memory cache limits are encouraged in these situations.

10.5 Test Result

Detailed test results are available in appendix F, and only some parts are extracted for deeper analysis. Some parsers did not succeed in completing all test within reasonable time. In particular this was the case for very large files which was close to a third of the internal memory. These files (Test files 6, 12, 18, 24) made the memory requirement exceed the limits of the internal RAM and made the application break. The most basic tests are also carried out on Crimson, DOM4J and JDOM, but these results are not discussed in this chapter. Test results for all implementations are available in appendix F.

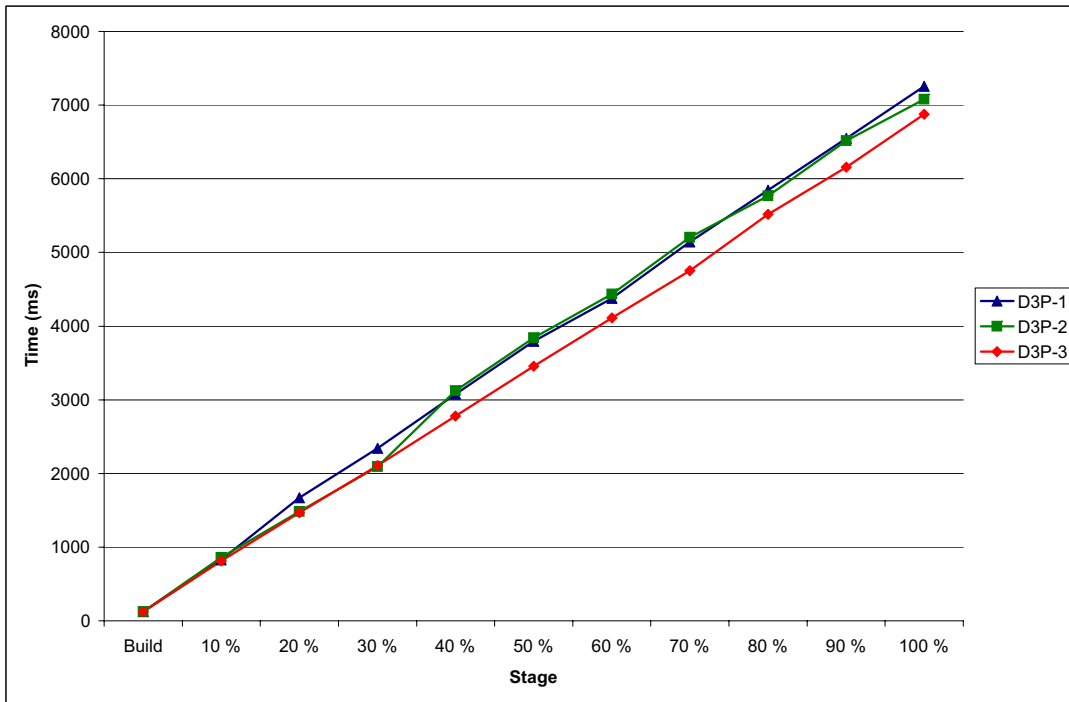


Figure 63: Cache size impact, time usage

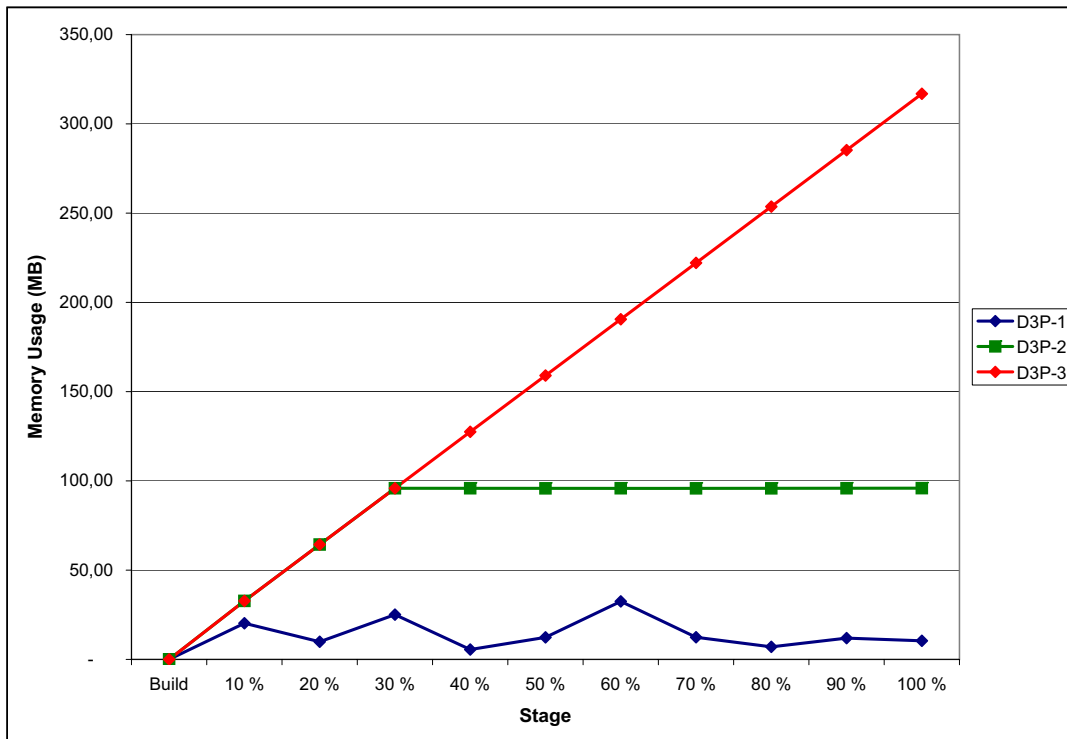


Figure 64: Cache size impact, memory usage

Parser	Time in ms
XercesDeferred	4000
Xerces	8296
C#	5078
D3P - No parse	15
D3P	10016

Table 3: Document build time. test file 5

Parser	Time In Milliseconds
XercesDeferred	485
Xerces	781
C#	266
D3P No Parse	16
D3P	563

Table 4: Document build time. test file 2

In this result summary, only parts of the test results are shown. Tests with noticeable results are emphasized and further discussed in chapter 11. Each test is supplemented with a comment to stress points of interests in an objective matter.

10.5.1 Document Build Time

Figure 65 shows the test results for the build time of test file number 5. The results are also reflected as numeric values in table 3. Test results for file number 2 is also included in numerical values in table 4. The size of test files 5 and 2 is 118 MB and 4684 KB respectively.

As observed in both tests, C# outperforms the Java parsers. Also, the Xerces parser has significant lower build time with deferred mode turned on. Having less startup cost is a normal behavior with lazy and demand driven parsers. This is also reflected when comparing the D3P No Parse and the C# parser. The D3P with indexing is the slowest of all, due to overhead in the analyzing of the XML document.

With the D3P No Parse, the time used in the build process is not used building any nodes, but to prepare indexes and open a file stream. Hence the build time is not depending on how large the source XML document is, or how it is structure, but merely of loading indexes.

10.5.2 Tree Walk Time

Figure 65 shows the measured results for traversing file number 5 in preorder traversal. The results are also reflected as numerical values in table 5.

Results in figure 65 and table 5 shows that D3P, D3P No Parse and Xerces Deferred uses more time than the others. That is, all demand driven parsers uses more time for a full traversal of the node tree. This is also acknowledged by the other tests listed in

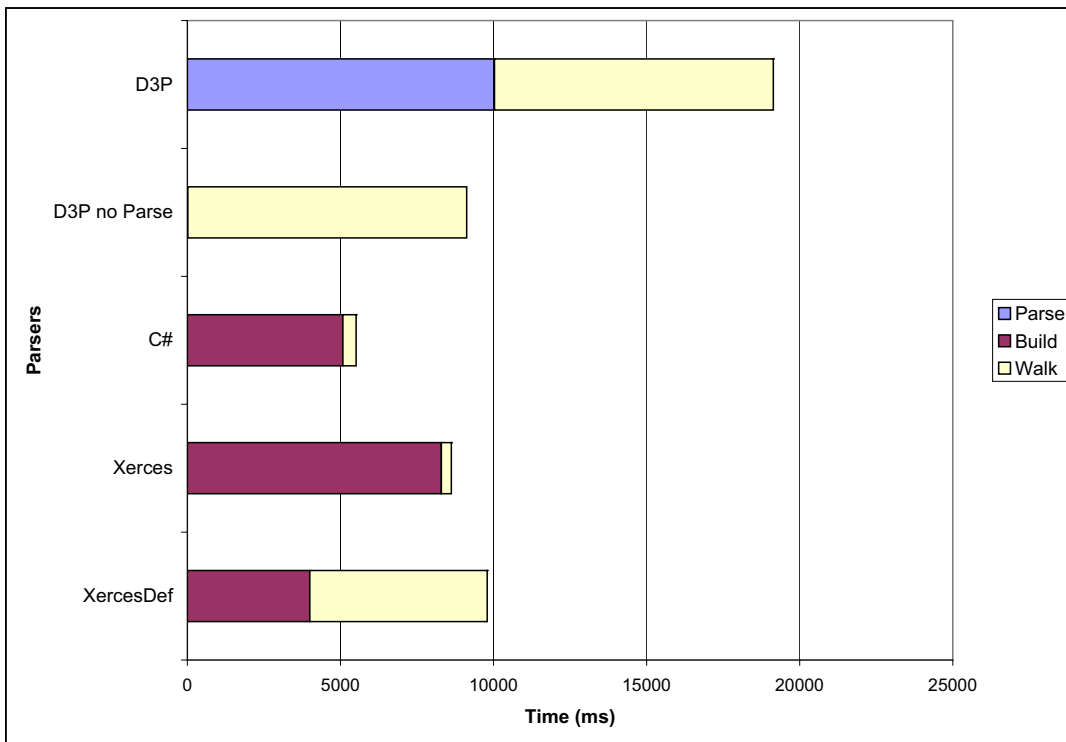


Figure 65: Full traversal, file 5 time usage

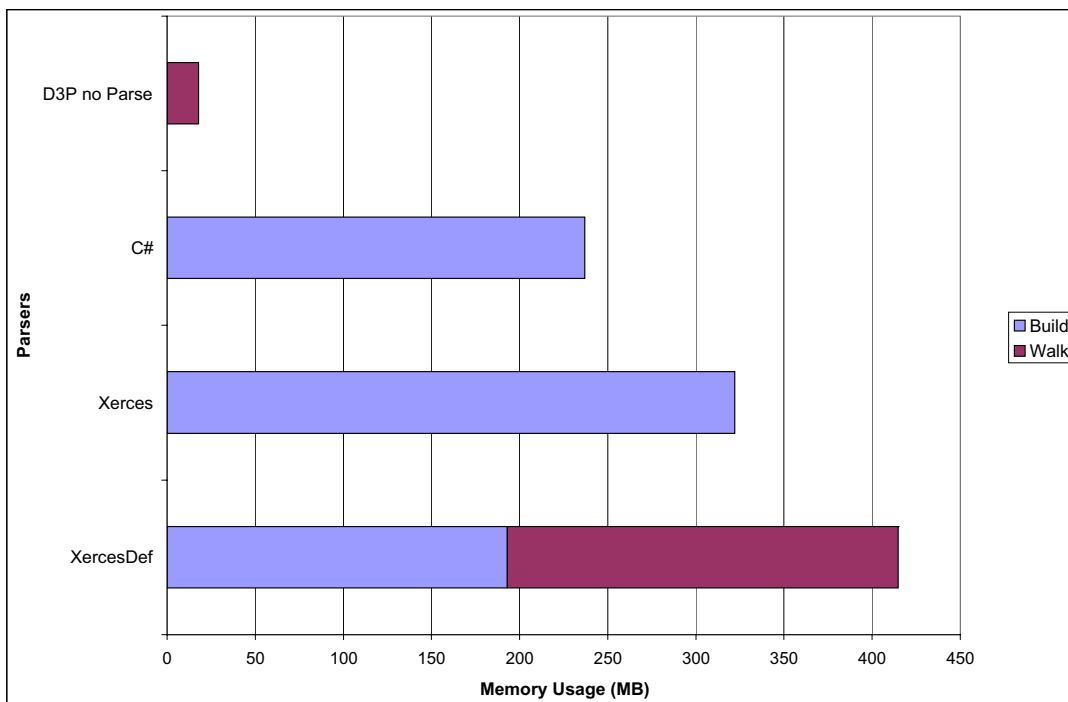


Figure 66: Full traversal, file 5 memory usage

Parser	Time in ms
XercesDeferred	5797
Xerces	328
C#	438
DP3	9110

Table 5: Document walk time. test file 5

appendix F.3.

10.5.3 Document Memory Usage Before and After Walk

The D3P has the ability to adjust the memory consumption at any time. That is, how much node tree data is to be in memory cache. To compare this solution with others might seem odd, because the memory consumption can be lowered to outperform the others. Even though, a reasonable cache limit is set for this test.

Figure 66 shows the memory usage both before and after walk for the implementations. Test file 5 of size 118 MB is used. Both D3P and Xerces Deferred uses memory during traversal in contradiction to Xerces and C# parser. The latter actually uses less memory when the traversal is done than before. This is acknowledged by test results in appendix F.3. Observe that Xerces Deferred uses more memory than Xerces after the full traversal. This was expected from the Xerces specifications listed in chapter 6.2.

10.5.4 Partial Traversal Test

Figure 67 and 68 shows the measured results for time used and memory consumed when only partial traversing the node tree. The test uses test file 16 of size 46 MB. Observe that Xerces Deferred uses less time than Xerces in normal mode, which is expected when only parts of the document actually is visited. However, comparing D3P No Parse and C#, D3P uses more time. D3P uses less memory, due to the ability to adjust memory consumption. Comparing the Xerces and Xerces Deferred unveils that the latter surprisingly is using more initial memory than Xerces in normal mode. This is not correct according the technical specifications in [42]. Additional test results in appendix F.3 reveals that this only happens for XML documents with attributes.

10.5.5 Partial Traversal Test Repeat

Figure 69 and 70 shows measured results for time spend and memory consumed in the test respectively. The test is performed on test file 16, with a size of 46 MB. Note that only 10% of the document is accessed. The graphical representation shows the first traversal and the final nine separately. This is to show the measured difference in the first and remaining traversal. Note that both D3P and Xerces Deferred build the nodes in this first traversal, but traverse in memory data in the remaining. With the Xerces as the highest measured time and D3P not far behind, the D3P no parse uses the less time in this test. Note that D3P (normal and no parse) uses 31 MB of memory. This is 20% of the C# implementation which uses 150 MB.

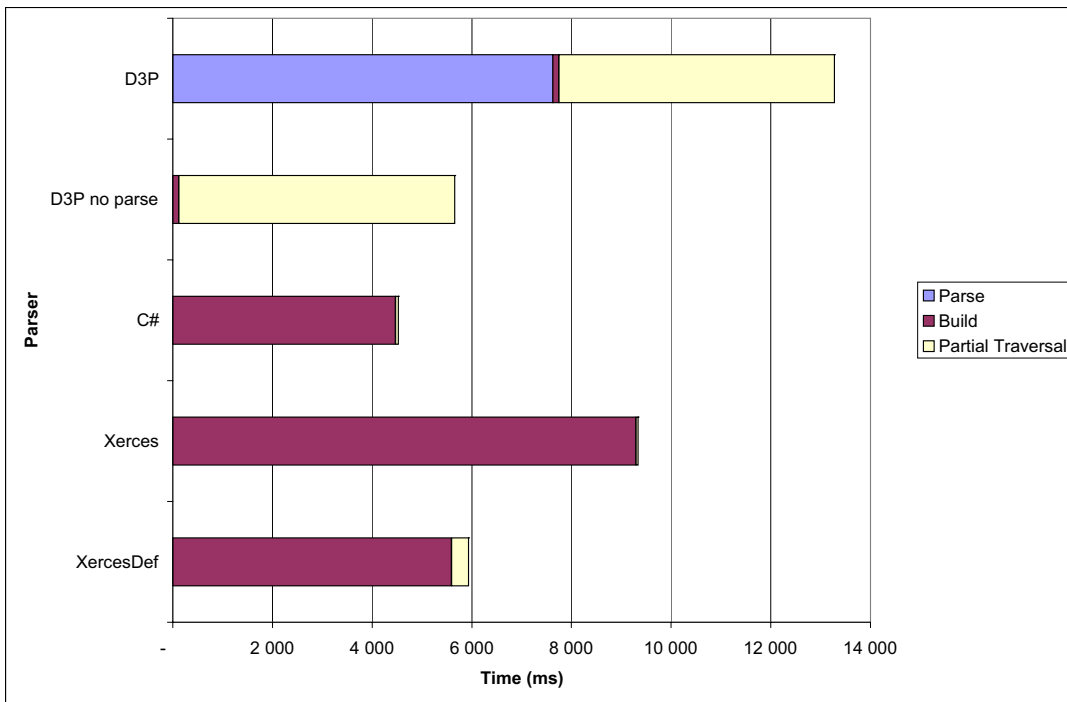


Figure 67: Partial traversal, file 16 time usage

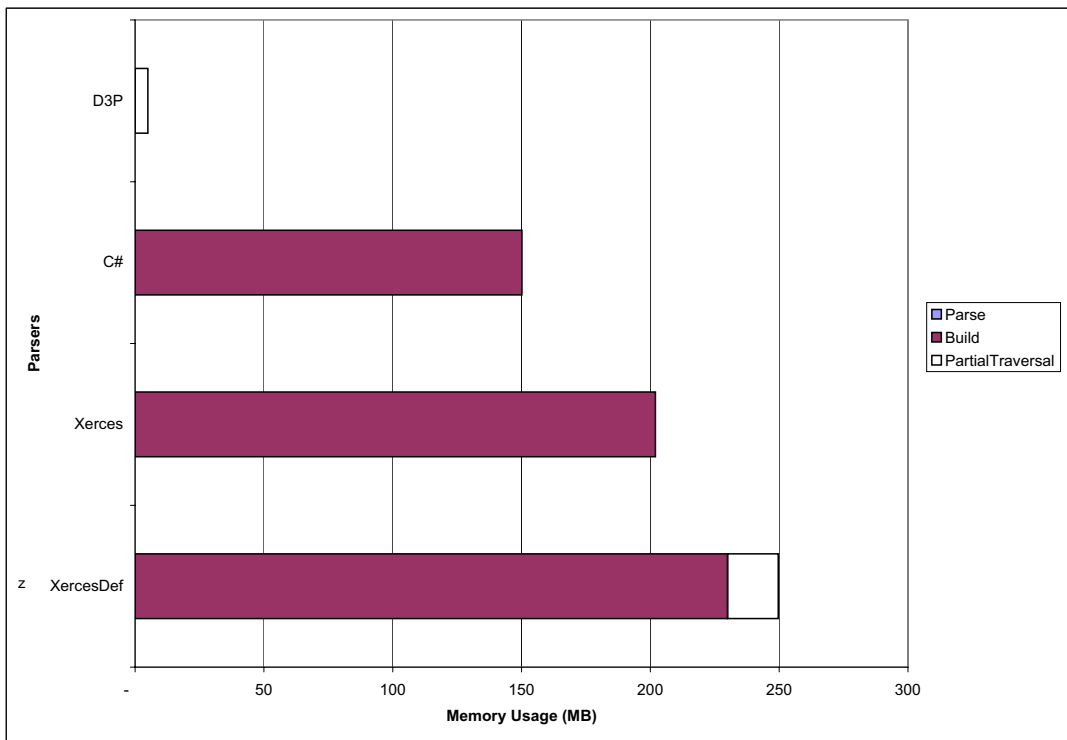


Figure 68: Partial traversal, file 16 memory usage

TESTING

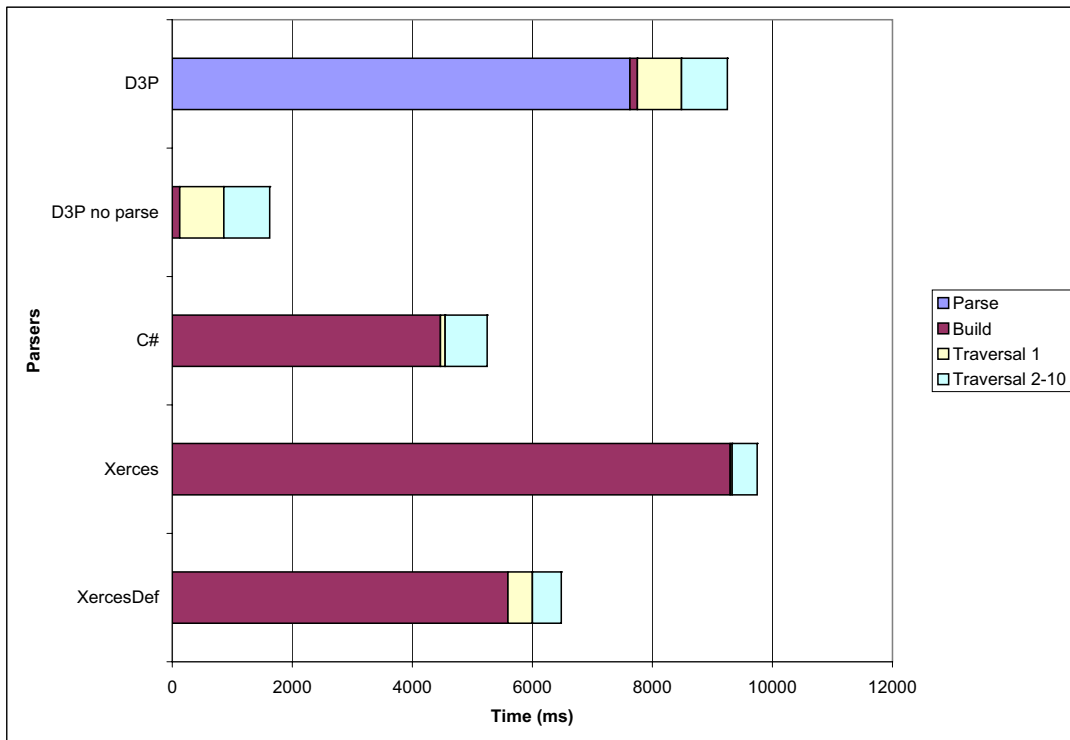


Figure 69: Partial traversal repeat, file 16 time usage

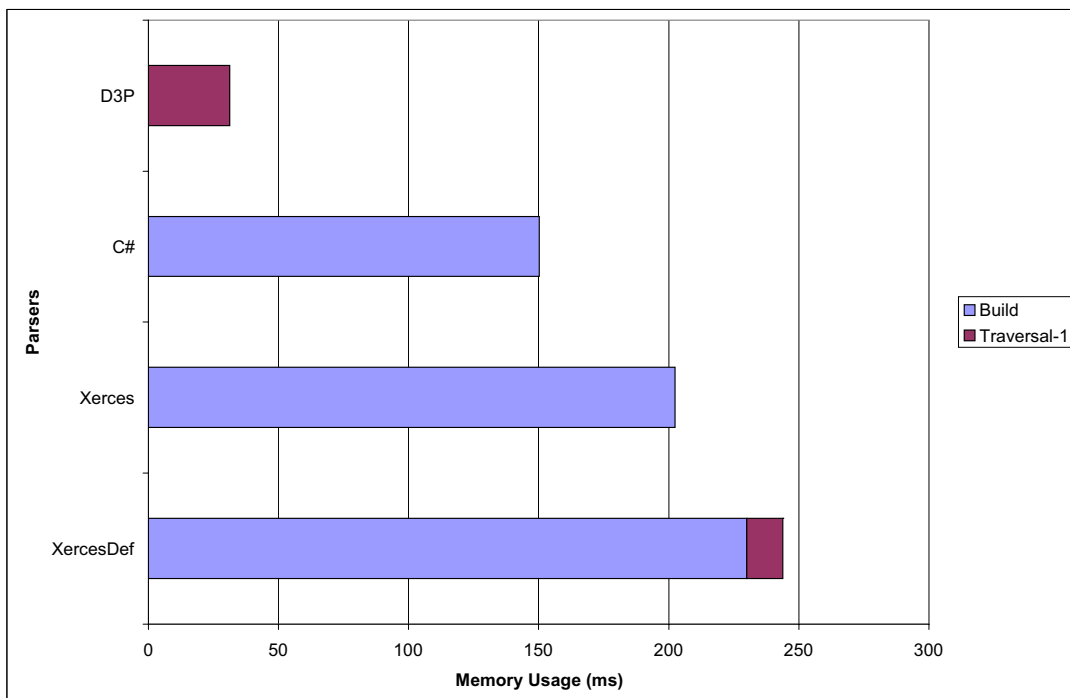


Figure 70: Partial traversal repeat, file 16 memory usage

10.5.6 Remove Test

Figure 71 and 72 shows the D3P produce a very good result. This is due to the fact that it does not build more than the document element. And when running output it just reads and writes the delta unmodified area directly from one stream to another. Observe that Xerces and C# uses only half of the memory after the remove test. This indicates that the nodes simply are removed both from the node tree and from memory. In the write operation, Xerces Deferred uses more memory compared to the other implementations. D3P uses less memory in this test, almost unnoticeable in the graphical presentation.

10.5.7 GetElementsByTagName Test

Figure 73 and 74 shows the measured results for the `getElementsByTagName` test. The D3P and D3P No Parse have a number of entries due to different `CollapseLimits` used. The figure shows that the performance this function is highly dependent on the `CollapseLimit`. With D3P No Parse, the best measured setting for test file 16 was 1 KB. This particular setting gave the measured lowest both time and memory consumption in this test. Also observe that both parsing and document build takes longer for a lower `CollapseLimit`.

10.5.8 Document Walk Time and Memory, Percentage Coverage

Figure 75 shows the measured test results using XML test file 16. The test includes load time for all parsers. The D3P with initial indexing included is not included in the test. The Xerces Deferred has lower measured time than the Xerces normal mode for document coverage under 83%. The same observation can be done with D3P No Parse versus the C# implementation. Here the crossover is at 65%. Note that the memory usage of the two implementation using demand driven approaches. Xerces reaches many times the document size; while D3P No Parse's highest measured memory consumption is 40 MB, under the size of the document (test file number 16 is 46 MB).

TESTING

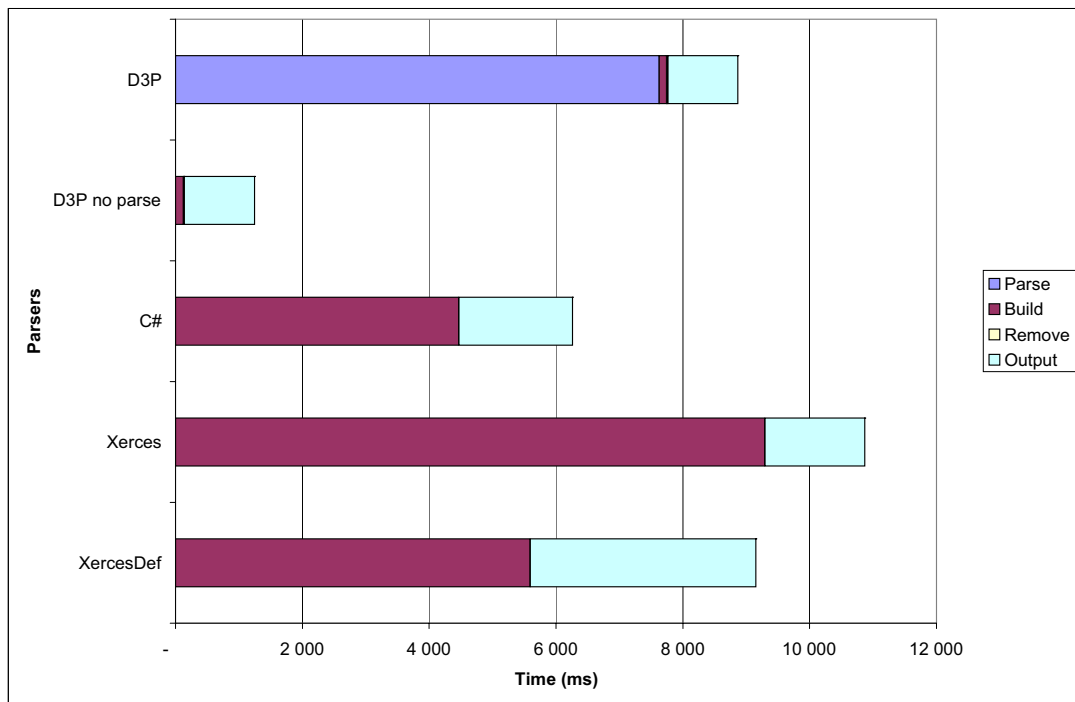


Figure 71: RemoveTest, File 16 time usage

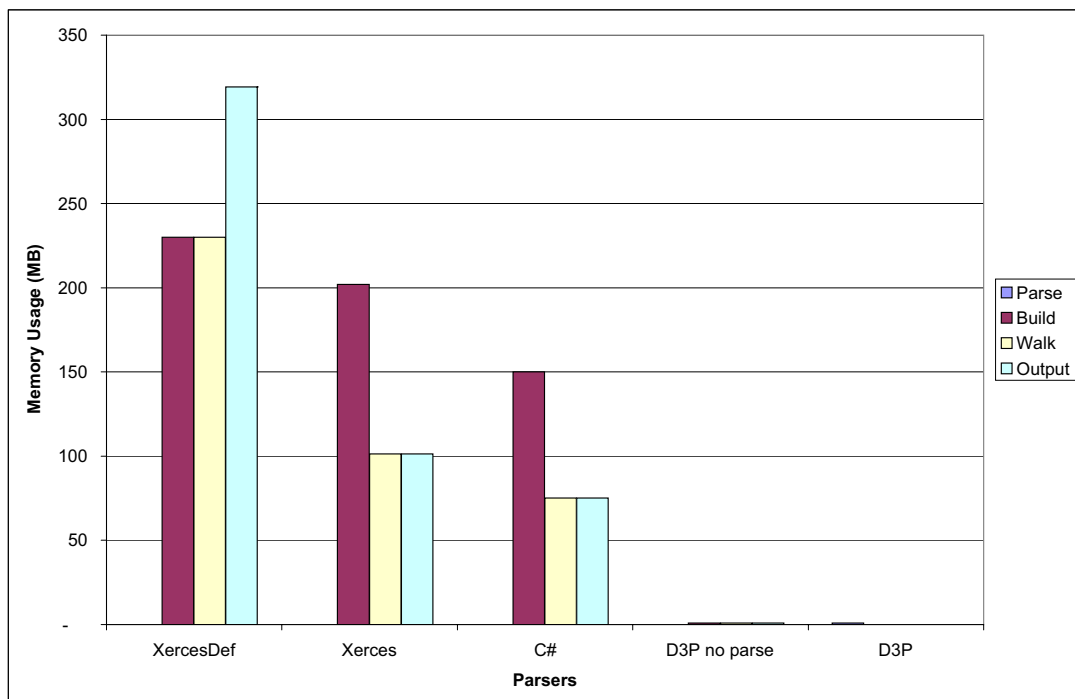


Figure 72: RemoveTest, File 16 memory usage

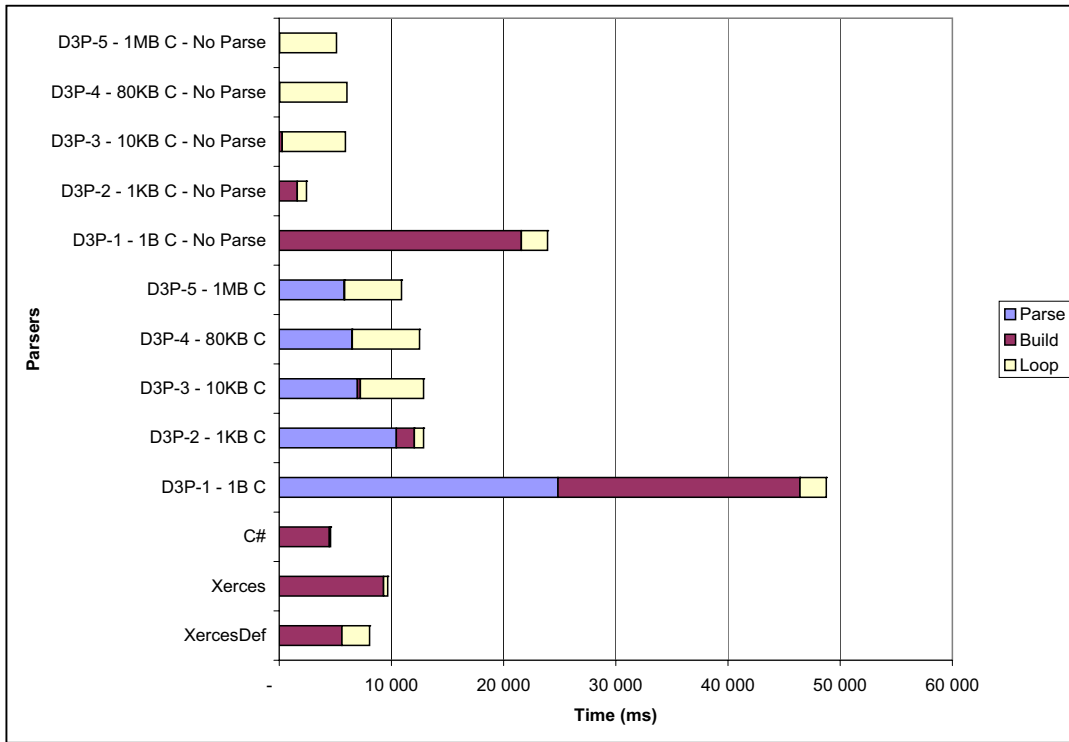


Figure 73: GetElementByTagName, file 16, measured time

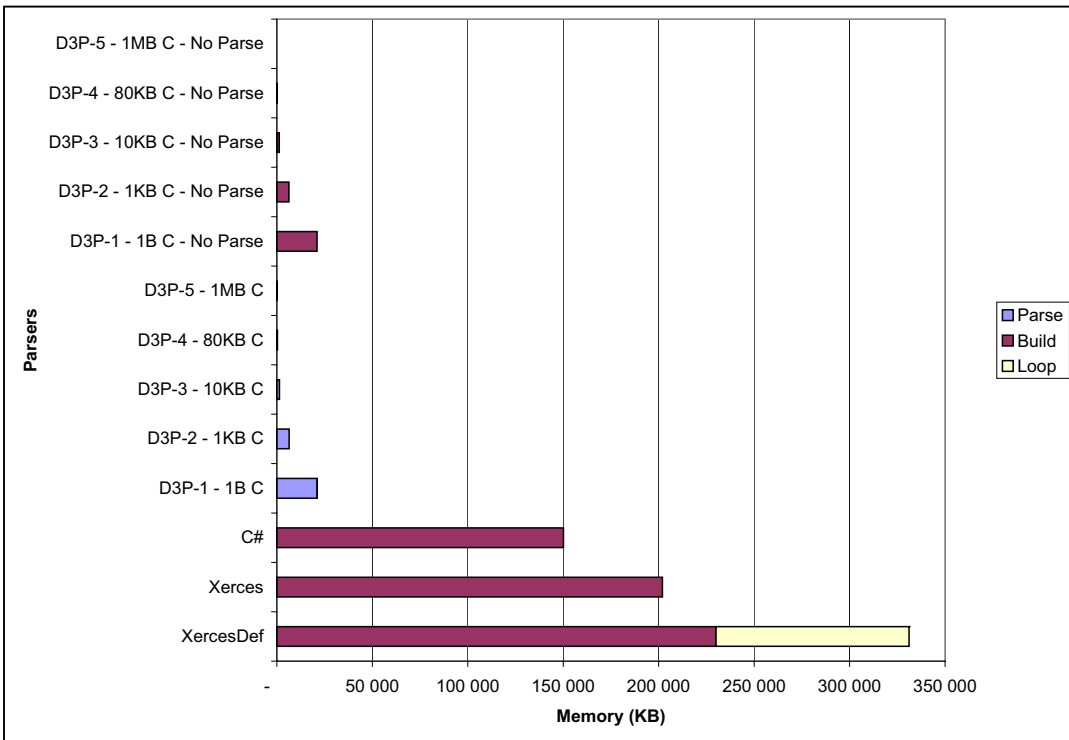


Figure 74: GetElementByTagName, file 16 memory consumed

TESTING

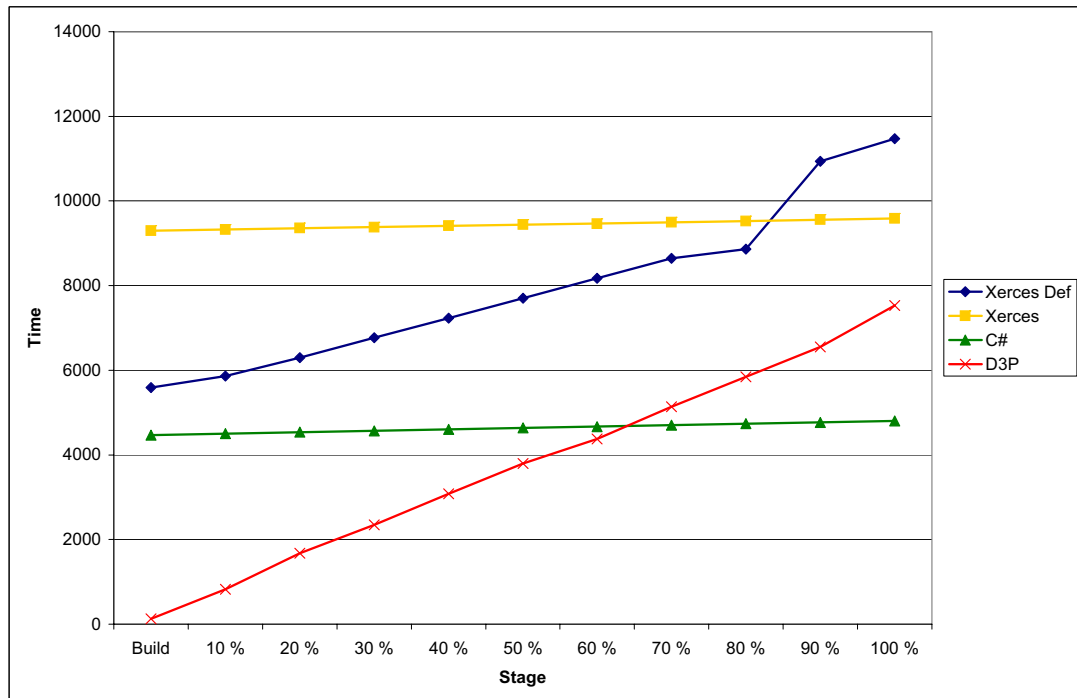


Figure 75: Traverse time for percentage document cover

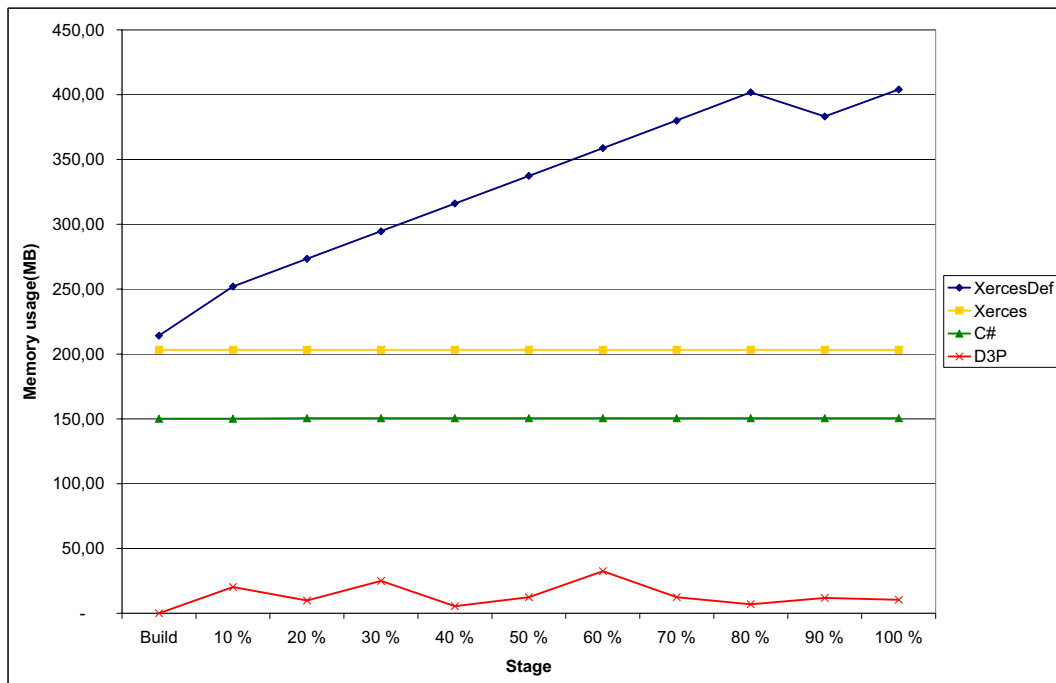


Figure 76: Memory consumed for percentage document cover

11 Discussion

The test results confirm what is expected for an on demand XML parser: Slower because it needs more source data accesses and processing, but uses less memory. The parse and indexing time uses a lot of time for the D3P. Since the indexing is a “parse once, use many times” operation for static documents, this index time can be disregarded in many use cases. To show remarks with the test results, some results are emphasized:

Tree Walk Time

Test results are as expected for total traversal using a demand driven parser; D3P and Xerces Deferred is slowest. This is verified by theory in chapter 3.2. The cache size of D3P has limited impact on total traversal time when the entire document is traversed. This is due to that nodes are not referred to again once leaved behind by the postorder traversal.

Partial Test

D3P is only just beaten in terms of time used compared to the C# parser. This is also as expected. The test uses nodes from different parts of the document. With the parser settings used, the value of CollapseLimit is set to 80 KB the traversing will result in building many nodes not used, not fulfilling the full potential of D3P. Having a lower CollapseLimit, makes the parser index all of the nodes, would solve this problem but introduce another, namely the number of disk accesses. With over 250 000 element nodes and traversing 68 000 of them this would lead to 68 000 disk accesses. A solution to this problem could be having the whole document in memory using a *MemoryStream* (discussed in chapter 8.4). An approach is also used by the VTD-XML parser. Doing this amount of file accesses would make the I/O a bottleneck in the system. But this also involve having the whole file in memory, resulting in a large memory overhead, which can be avoided with the approach used in this example.

Partial Traversal Test Repeat

This test basically shows the D3P performance when only a part of the document is in memory and the caching is turned off. This is the perfect scenario for D3P. All nodes which are build are used several times and none is discarded. In test file 5, as shown in the test result, D3P parser uses approximately 30% (1625 ms versus 5249 ms) of the time used by the C# parser. Also this test shows that when keeping all of the nodes in memory the D3P parser it is only marginal slower than the C# parser; Traversing the last 9 nodes in 765 ms versus 702 ms by the C# parser. The D3P can be tuned exactly on how much memory to be used, making the initial tuning of D3P important considering the scenario to be used in.

Get Elements By Tag Name

The test shows the importance of choosing the right CollapseLimit to the purpose of usage. The task of finding the “magic limit” for the CollapseLimit is not easy. Even though it can be argued that for the `getElementsByTagName` test, a CollapseLimit of 1 byte (indexing all nods) would be ideal, tests shows the opposite. This is due to many disk accesses. One for each element actually holding attributes.

RemoveTest

D3P is superior in this test. This is simply because no nodes are actually made, except for the root node. When writing, has to build to nodes to write and therefore uses much memory in this operation compared to the others. D3P uses approximately zero memory except the application data and indexes, because it does not have any nodes build except the rootnode. When writing it takes advantage of this, writing from the new file directly from the source file. This method is described in chapter 3.1.2.

Document Walk Time and Memory, Percentage Coverage

The result shows how well the D3P perform when a percentage of the document is covered. The D3P performs better when only parts of the document is covered. This can be used as an indication for when to use the D3P and when to use a traditional implementation. If the user has a large XML document, but knows the total usage will only cover a smaller part of it, then D3P can be a lucrative option.

In all tests regarding memory consumption, D3P is superior. This is simply because D3P can adjust its memory consumption. No other W3C DOM implementation does this, making the D3P unique.

11.1 Evaluation of the Implementation

The proposed prototype described in chapter 8, is implemented and working. The discussion makes no doubt that the implementation has it strengths, but also some weaknesses. To simplify the evaluation, strength and weakness of the proposed prototype implementation and the approaches used are structured in separate chapters. Chapter 11.2 and chapter 11.3 deals with the two different aspects respectively.

11.2 Application Strength

The D3P handels XML data as W3C DOM nodes in a way not done before. In terms of speed and memory management the implementation works as predicted. In every test, D3P allocates less total memory than any other W3C DOM based implementation. The D3P also has the ability to use a custom configuration to improve performance.

The measured time results shows that D3P No Parse generally is approximately 40 % slower than the C# parser in full traversals of the document. When coverage decreases, D3P catches up. As the percentage document coverage test in chapter 10.5.8 shows, the D3P is faster in terms of combined build and walk time when it comes to approximately 65 % or less document coverage. D3P is also faster on traversing a pre defined parts of a document several times discussed in chapter 10.5.5. When keeping all of the data in memory, D3P is marginal slower than the .NET parser, as expected.

11.2.1 API

The API supports the W3C DOM, making D3P compatible with applications using the DOM interface. Although it does not build on specific interfaces, since none such are supported by the Microsoft .NET framework, a user will fully recognize the W3C DOM interfaces used in this library.

11.2.2 Memory Consumption

This implementation has the ability to adjust memory consumption, even dynamically at runtime, making it an extremely powerful W3C DOM library for large XML documents. Most W3C DOM implementations does not have this ability, limiting the size of XML data being accessed through DOM. The application has a little overhead for index, symbol table and cache algorithm, but even so, memory consumption is considerable lower than any other W3C DOM implementation today.

11.3 Application Weakness

The application weakness is the same as its strength, namely the memory usage. When the memory consumption is forced down only small parts of the document is kept in memory. This is described in chapter 11.2.2. This can result in performance issues in some cases.

11.3.1 Performance

D3P will have performance penalties as the coverage of the XML document increases. In theory, techniques like lazy loading and demand driven processing will always have performance penalties over eager approaches when all data is used as shown in chapter 3.2. This is due to the overhead of having more individual accesses to the XML source. In addition, this implementation also removes nodes from memory as it goes. That is, time is spent removing nodes from the node tree and running the garbage collector. D3P can to a certain degree adjust how lazy it should be. In principle, only the root node in the XML document has to be loaded, and each children of the root node can be loaded in an eager approach. This can be accomplished by setting the CollapseLimit to Int32.MAX.

11.3.2 Finding Optimal Configuration

Finding the optimal configuration to use with the D3P is not an easy task. There are many considerations to take, and a mistake can result in lower performance. This particularly concerns the selection of the CollapseLimit for a document. This is absolutely a drawback of the application.

11.4 Deferred Overhead

Based on the general test results (available in appendix F.3), the overhead in terms of time and memory consumption was analyzed. The results are shown in appendix F.6. When comparing Xerces and Xerces Deferred, the deferred variant normally has a memory overhead of 25 % to 90 % depending on the document structure. The overhead in terms of processing time is large on small documents. Up to 70 %, but less significant as the document increases, probably because of larger initial startup costs. The result is as expected and also shown in earlier results. When doing full coverage of XML documents, deferred parsers like Xerces Deferred always has an overhead. Such a parser would be favorable when document coverage is under a certain limit. As the percentage document coverage test in chapter 10.5.8 shows, the Xerces Deferred is the

DISCUSSION

fastest parser when document coverage is less than 83 84 %.

When comparing D3P and the C# parser, a different result appears than in the Xerces/Xerces Deferred comparison . The configuration has a large influence on the behavior of the parser. With the final test settings, the parser normally uses approximately 40 % more time than the C# parser. This is quite acceptable due to overhead caused by the caching, indexes, I/O accesses etc. The main advantage of the D3P parser is its low memory consumption, using only 1-30 % of the C# parser's. The consumption is more or less independent of the XML document size. The granularity of the index has impact on the memory consumption. Large granularity i.e. a high CollapseLimit value makes the D3P consume more memory, while small granular indexing will make the parser use less memory, but more processing.

Part IV

Conclusion And Further Work

12 Conclusion

12.1 Comparison with Others

The test results from chapter 10.5 and the further discussion in chapter 11 indicates the differences between the implementations. Generally, the implementations on the Microsoft .NET platform perform better than the implementations on the Java platform. But however, the Xerces in deferred mode can be compared to D3P as Xerces in normal mode is compared to the MS .NET library implementation.

12.2 Summary

In this thesis, the aim has been to develop and evaluate a W3C DOM implementation for large XML documents, by utilizing demand driven approaches and caching to decrease the memory consumption. We started by gathering comprehensive background information regarding the basics of XML, how to handle XML and parsing techniques. Also, memory handling and indexing techniques for XML have been examined to find the best solution. Although no similar W3C DOM implementation could be found, two XML processors with untraditional approaches has been carefully examined, namely the Xerces2 Deferred and VTD-XML. Both these implementations utilize techniques for lower memory usage. We wanted to take memory consumption to a bottom level, even lower than the existing solutions for object models.

This resulted in the prototype implementation of D3P (Demand Driven Dom Parser). It utilizes preparsing, making an index later used for navigation of the node tree. A memory cache is used for the XML nodes currently used, and a replacement algorithm selects nodes to be removed from cache, when the limited cache size is exceeded. The cache size can be changed dynamically at runtime. The D3P implements W3C DOM Core level 1 with minor modifications, and the implementation partially supports write back of changes to secondary storage.

Compared to other solutions, D3P uses less memory than any W3C DOM implementation currently known. As document coverage is low, the D3P is advantageous, but as document coverage increases, a traditional approach is preferred. When altering and saving an XML document, smaller changes in a large document will emphasize D3P as advantageous. It uses less time in the operation than other tested implementations.

12.3 Conclusion

We have through this thesis solved the problem of memory consumption for W3C DOM implementations. All nodes are retained in a memory cache, which gives the application full control of the nodes. To conclude the scenarios presented in the introduction (chapter 1.1.1), the solutions for these are presented here:

- Size of the XML document is of no importance. Huge XML documents are available through the W3C DOM interface.
- Faster document loading as document coverage decreases. If only a small portion of the document is loaded, the memory consumption is no higher than the data representing this part of the document.

The proposed solution always has an advantage when the memory consumption is considered. This is due to that every node in the node tree is seldom referred to all at once. Hence, not all nodes has to be located in memory. The D3P makes the decision of which nodes to be allowed in memory.

Total processing time is always higher for lazy implementations, with the proposed solution as no exception. Doing a full traversal of the XML document, makes the traditional approaches to be preferred. As total document coverage decreases, the suggested approach are advantageous because it only uses resources reading the data in question. These results was expected through theory about lazy loading, and are reflected in the results of this thesis.

We hope this thesis can be an inspiration for further work of handling XML. The proposed prototype shows how lazy loading can be combined with caching to control memory handling, even for tree structured data like XML.

13 Further Work

In this section we present points in this thesis which could be improved. The implementation in this theses is considered a prototype, and during the limited time available for the master thesis, all plans for implementation and evaluating were carried through. The following chapters will highlight some of these plans, which can be used as a basis for further research, development and evaluation.

13.1 Analysis Tool

As discussed in chapter 11.3.2, the optimal configuration is hard to find, especially for an inexperienced user of the D3P. In particular the value of CollapseLimit is hard to evaluate. For the user to find value which gives a good balance between memory consumption and number of data accesses, knowledge about the document structure is essential. For D3P to be a generic W3C DOM library, an analysis tool should be proposed. This tool could, given criteria's for user pattern and knowledge about the document, structure select a CollapseLimit for the document to be indexed with. The tool has to analyze the entire document or take samples of it to get a structure overview.

13.2 Update Index

Indexes which do not alter in its lifetime are referred to as static indexes. This prototype utilizes static indexes due to the non extractive approach of mapping start and end offset in the XML document. The start and end offsets gives D3P the ability to extract data from the XML document on request, without having to search whole or parts of the document.

The static index approach leads to a forced re index of the XML document when a new file is written. A new approach is desired, which update the index when writing the new file. Further development and research in the field of solving the index update problem might determine the actual utility value of this demand driven W3C DOM approach.

13.3 Extended DOM Support

Full support for W3C DOM recommendations is desired. The current implementation has support for all functionality within Core 1 which is considered the most important, but some interfaces remains. Core 2 have support for namespace, a functionality which is extremely important as the collection of XML files grows. The parser also do not have support for DTD validation, a desired functionality in future versions. The DTD could also possibly be used to reveal the structure of the document and could be considered when reaching decisions in the proposed analysis tool discussed in chapter 13.1.

13.4 Perfect Writing

W3C DOM implementations output XML formatted differently. The XML recommendation does not define how the resulting text file containing the XML data should be formatted, only that the content of this file should be well formed XML. Hence, XML documents written to file as output from various implementations vary. Some even has an option for selecting "pretty XML", which means that line feeds at end of nodes and a

number of white spaces or tabulators before nodes is used to make the document more human readable. Typically, a fixed number of white spaces are used before each node or element.

In the D3P prototype implementation, write back utilizes non extractive approaches to merge the old source document with new and altered nodes which reside in memory. When copying pieces of the source document using only start and end offset, there is no way of knowing at what node level the copied data is located, hence when new or altered nodes are written, it has no knowledge how to format its data. This means that the output sometimes looks odd. By having knowledge of the current level of the node (either by introducing yet another index parameter or analyzing existing index / node tree) and how the legacy XML is formatted, the integration between legacy XML and new XML will seem seamless.

13.5 Switch Memory Model

The managed memory model in frameworks like Java and .NET uses much resources cleaning removing objects from memory. And as discussed in this thesis, the operation of removing D3P nodes from the memory is tricky: All references to an object have to be removed for the garbage collection to remove the object. In addition, the garbage collection uses resources; checking references and reorganizing the heap.

Languages like C++, has another memory model, which put all responsibility for the memory handling on the shoulders of the developer. Memory are considered to released when the *delete* operator is used on data. Data is removed manually, with no expensive garbage collection. It could be interesting to see how the techniques used in this implementation would perform under another memory model. Even though this master thesis describes a prototype implementation in C#, principles used here, could also be migrated to other languages.

13.6 Serialize Dirty Nodes

As the node tree is altered and new nodes are added or deleted, everything works as with traditional implementations. But when a new or altered node has to be removed from memory due to full memory cache, all these nodes are still retained in memory. These nodes have to be removed from memory, but they can not be discarded as other nodes, because they can not be retrieved from the source XML document. Functionality for saving these nodes to persistent storage as a temporary file is desired. A suggestion is to use already established techniques of PDOM (Persistent DOM). Many native XML databases use PDOM as their internal storage, also with a complete index over the DOM nodes.

13.7 Save State

A common approach for W3C DOM implementations is to alter the node tree while it is in memory and write the entire node tree to a new document when a command is issued. This is also supported by the D3P prototype, but since the index uses non-extractive information, it is recommended by Zhang [1] to save the state of the file, rather than updating it. With the index already in persistent storage, only the update

log and the in memory dirty nodes have to be written. This will be significant faster than updating the source file, in particular when the size of the file increases. Also, discussed in chapter 13.2, index update when writing an altered XML node tree to file is not implemented. With the *save state*, function, the index does not have to be re indexed.

A proposed approach is to write the file back in intervals, i.e. when the cost of use or size of serialized nodes is increasing to a critical point, or when the file is needed by some other application.

13.8 More Test Results for Evaluation

The evaluation of the demand driven approach used by D3P shown in chapter 10 is mostly based on approaches for reading data from the node tree. On altering the node tree, only one test is issued, namely the "Remove Test" discussed in chapter 10.3.3. Even though the D3P has no problems with running test which also add or alter arbitrarily nodes, these tests has not been included in this thesis. The reason is the lack of serialization of dirty nodes discussed in chapter 13.6. As the number of altered and new nodes increases, so will the memory consumption, eventually wasting the point of a demand driven DOM parser. For further study, when the serialization of dirty nodes has been completed, another set of test can unveil the performance of demand driven approaches for altering a W3C DOM node tree.

References

- [1] Jimmy Zhang - Non-Extractive Parsing for XML. [23.05.06]
<http://www.xml.com/pub/a/2004/05/19/parsing.html>
- [2] Aleksander Andrzej Slominski - Push and Pull: complementary sides of XML parsing [23.05.06]
http://www.extreme.indiana.edu/xgws/papers/xml_push_pull/node3.html
- [3] XQuery : the Xml Query Language. [10.02.06] <http://www.w3.org/TR/xquery/>
- [4] World Wide Web Consortium. [24.01.06] <http://www.w3.org>
- [5] Extensible Markup Language (XML). [25.01.06] <http://www.w3.org/XML>
- [6] Hunter, Watt m.fl - *Beginning XML 3.rd Edition* (2004) Wiley Publishing Inc.
- [7] Bourret Ronald - XML and Databases. [08.05.06]
<http://www.rpbourret.com/xml/XMLAndDatabases.htm>
- [8] Akmal Chaudhri m.fl - *XML Data Managment: Native XML and XML-enabled Database Systems*. Addison Wesley (2003)
- [9] Anatomy of a native XML base management system. [08.03.06] *The VLDB Journal* nr 11/2002
- [10] Sleepycat Software - Whitepaper on Anatomy of a Native XML Database [04.05.06] <http://www.sleepycat.com/>
- [11] Robin Cover - Technology Reports - W3C Document Object Model (DOM). [04.05.06] <http://xml.coverpages.org/dom.html>
- [12] DOM origion and the Level. [24.02.06]
<http://www.quirksmode.org/index.html?js/dom0.html>
- [13] W3C DOM CORE 1 Specifications. [24.02.06]
<http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html>
- [14] W3C DOM Core 2 Specification. [23.02.06] <http://www.w3.org/TR/DOM-Level-2-Core/>
- [15] Arnaud Le Hors,Elena Litani - Discover key features of DOM Level 3 Core. [04.05.06] <http://www-128.ibm.com/developerworks/xml/library/x-keydom.html>
- [16] Standard Generalized Markup Language (SGML). [15.05.06]
<http://xml.coverpages.org/sgml.html>
- [17] This is the official website for SAX. [26.05.06] <http://www.saxproject.org/>
- [18] Catania B, Ooi B, et.al. - Lazy XML Updates: Lazyness as a Virtue of Update and Structural Join Efficiency. [05.05.06] <http://www.comp.nus.edu.sg/ooibc/sigmod386.pdf>
- [19] XML:DB Initiative XUpdate: XML update Working Draft. [05.05.06]
<http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html>
- [20] XML:DB - The XML Database Initiative. [27.05.06] <http://xmldb-org.sourceforge.net/>

REFERENCES

- [21] Bic, Shaw - *Operating Systems Principles*. Prentice Hall 2002
- [22] Xerces2 Symbol Table. [15.02.06]
<http://xerces.apache.org/xerces2-j/xni-xerces2.html#symbol-table>
- [23] Kjell Brasbergengen *Lagring og behandling av store datamengder*, Tapir (2003)
- [24] Connolly Thomas, Begg Carolyn *Database Systems*, 3rd edition, 2002 pp. 998 1044.
Addison Wesley
- [25] Baeza-Yates, Ribeiro-Neto *Modern Information Retrieval Addison Wesley; 1st edition*
(May 15, 1999)
- [26] Dietz, Paul F. Dietz. Maintaining order in a linked list. *In Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pp 122 - 127
- [27] Quanzhong Li, Bongki Moon: Indexing and Querying XML Data for Regular Path Expression. [26.02.06] <http://www.vldb.org/conf/2001/P361.pdf>, in proceedings of VLDB, 2001
- [28] Gongzhu Hu, Chunxia Tang. Indexing XML Data for Path Expression Queries *Lecture Notes in Computer Science Volume 3026 / 2004* pp. 332 - 348. Springer-Verlag GmbH
- [29] Chun Zhang, Jeffrey Naughton et.al: On Supporting Containment Queries in Relational Database Systems. [15.03.06]
<http://www.cs.wisc.edu/niagara/papers/ZND+01.pdf>, SIGMOD 2001
- [30] Al-Khalifa, Jagadish m.fl - Structural Joins: A Primitive for Efficient XML Query Pattern Matching. [12.03.06]
<http://www.eecs.umich.edu/jignesh/pub/xmljoin-ICDE.pdf>
- [31] Duda, Kossmann - Adaptive XML storage or the importance of being lazy. [25.02.06] http://www.dbis.ethz.ch/research/publications/xime_p2005.pdf
- [32] Chung, Min, Shim - APEX: An Adaptive Path Index for XML data. [06.03.06]
<http://ee.snu.ac.kr/shim/sigmod02-apex.pdf>
- [33] Catania, Maddalena - XML Document Indexes: A Classification. [01.03.06]
<http://oswinds.csd.auth.gr/papers/ic05b.pdf>
- [34] Ponce, Vila, Hersch - Indexing and selection of data items in huge data sets by constructing and accessing tag collections [02.03.06]
<http://diwww.epfl.ch/tw3lsp/publications/gigaserver/iasodiihdsbcaatc.pdf>
- [35] Jeffrey Richter - Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework. [13.02.06]
<http://msdn.microsoft.com/msdnmag/issues/1100/gci/>
- [36] Noga M., Schott S., Löwe W. - Lazy XML Processing. [20.02.06]
<http://www.cs.uwm.edu/classes/cs790/digdoc-s2003/papers/p88-noga.pdf>
- [37] Jimmy Zhang - Improve XML Processing with VTD-XML. [07.01.06]
<http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/code/211657.htm>

- [38] Project Homepage of VTD-XML. [07.01.06] <http://vtd-xml.sourceforge.net/>
- [39] Jason Hunter - JDOM: How it Works and how it opened the Java Process. [07.01.06] www.jdom.org/downloads/oraoscon01-jdom.pdf
- [40] Dom4J homepages. [20.01.06] <http://www.dom4j.org/>
- [41] db4Objects Open Source embeded database Homepage. [03.02.06] <http://www.db4o.com/>
- [42] Project homepage of the Xerces2 DOM parser. [29.03.06] <http://xerces.apache.org/xerces2-j/>
- [43] Project homepage of the Apache Crimson parser. [31.05.06] <http://xml.apache.org/crimson/>
- [44] Mircosoft homepage - Process XML Data Using the DOM Model [31.05.06] [http://msdn2.microsoft.com/en-us/library/t058x2df\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/t058x2df(VS.80).aspx)
- [45] The mono project homepage. [07.06.06] <http://www.mono-project.com/>
- [46] What Is Performance? [28.04.06] http://developer.apple.com/documentation/Performance/Conceptual/PerformanceOverview/DevelopingForPerf/chapter_2_section_2.html
- [47] XMLBench Document Model Benchmark. [28.04.06] <http://www.sosnoski.com/opensrc/xmlbench/index.html>
- [48] Comparing XML Performance (.NET 2.0, .NET 1.1, and Sun Java 1.5). [28.04.06] <http://msdn.microsoft.com/vstudio/java/compare/xmlperf/default.aspx>
- [49] XMark - An XML Benchmark Project. [03.05.06] <http://www.xml-benchmark.org/>

Part V

Appendix

A Glossary

API Application Programming Interface

Blocked Node A node indirect reachable from the D3P index. A Indexed Node has references to it.

Cache A system for making local copies of data, for faster retrieval

Collapse Limit The maximum byte size of a node which is to be treated by the D3P index.

CPU Central Processing Unit

Content Object VTD-XML object. Holds the complete path to the current cursor (VTD record)

DHTML Dynamic HTML

DOM Document Object Model

DTD Document Type Definition

FIFO First In First Out algorithm

DR Data Retrieval

GC Garbage Collector

HTML HyperText Markup Language

I/O In/Out. Access to external elements (Disk/Network)

IDL Interface Definition Language.

Indexed Nodes Nodes which are directly reachable form the D3P index.

Indirect Blocked Node Nodes which are not reachable from the D3P index. They retain within a blocked node, and the blocked node has to be parsed for the Indirect Blocked Nodes to be revealed.

IR Information Retrieval

JIT Just In Time compilation. Frameworks like Java and .Net utilizes script files compiled at runtime to be optimized for the current OS and hardware configuration.

Location Cache VTD-XML object. Holds child and sibling structure information for the VTD-XML processor.

LRU Least Recentlry Used replacement algorithm

GLOSSARY

Managed Memory Model A Memory model relieving the developer from reclaiming allocated memory. Memory are allocated of a heap, in which all objects are located, and a Garbage Collector (GC) are used for removing objects from memory which are not referred to.

MRU Most Recently Used replacement algorithm

MS Microsoft

Name Tag Index Inverted Value Index in D3P. Gets identifier for element nodes which have a specific name.

OOP Object Orientated Programming

PDOM Persistent Document Object Model. Serialized DOM objects.

Page The amount of data adressed by a virtual memory management system

RAM Random Access Memory. The main memory in a computer.

Relational Index Index which holds information regarding navigation.

RDBMS Relational DataBase Managment System

SAX Simple API for XML

SGML Standard Generalized Markup Language

Trashing Unloading memory pages due to be used in recent future

XSL XML StyleSheet Language

XSLT XSL Transformations

Vitrual Memory Memory managment system controlling a virtual address space

VM Virtual Machine (in sense of java and .net)

VTD Vitrual Token Descriptor

XPath An XML Query language often used with XSLT

XQuery The XML Query language

W3C World Wide Web Consortium

Weak Refernce A reference not taken into concern when issuing the garbage collector in the Managed Memory Model

B Document Object Model

Logical View

DOM (Core) Level One

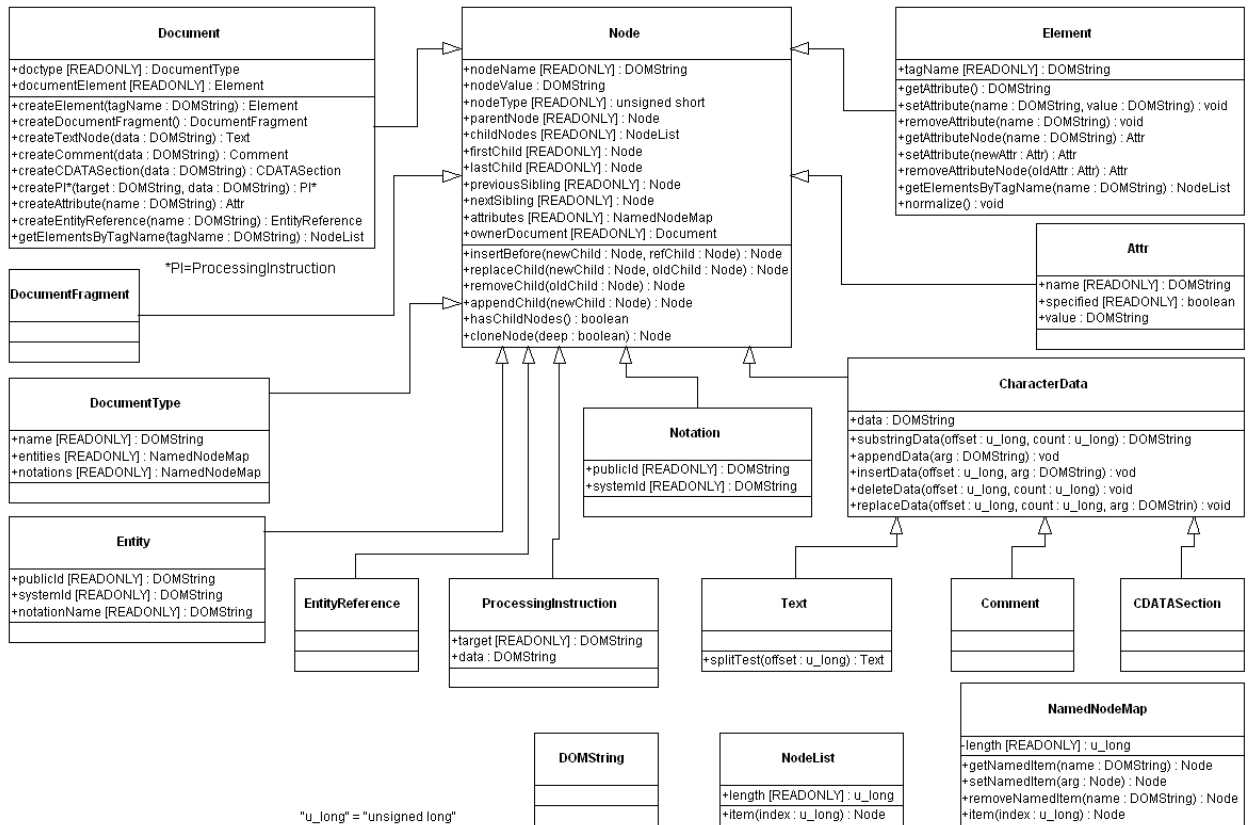


Figure 77: XML DOM core level one

C Sample Documents

C.1 Sample XML Document 1

```
<?xml version="1.0" encoding="utf-8"?>
<CD type="music">
  <Artist type="group">
    <name>
      U2
    </name>
    <title >
      Best of U2
    </ title >
  </Artist>
</CD>
```

Figure 78: Xml sample document #1

C.2 Sample XML Document 2

```
<?XML version="1.0" encoding="UTF-8"?>
<books>
  <bookstore>
    <book>
      <title lang="eng">Harry Potter</title>
      <price>29.99</price>
    </book>
    <book>
      <title lang="eng">Learning XML</title>
      <price>39.95</price>
    </book>
  </bookstore>
  <storage>
    <book>
      <title lang="eng">XML for real programmers</title>
      <price>34.50</price>
    </book>
    .....
    <book>
      ....
    </book>
  </storage>
</books>
```

Figure 79: Xml sample document #2

C.3 Sample XML Document 3

```

<Music-Store>
  <Group>
    <Title>Live8</Title>
    <CD>
      <Title>Perfect Day</Title>
      <Songs>
        <Song>
          <Title>Perfect Morning</Title>
          <Duration>2.19</Duration>
          <Collaborations>
            <Group>U2</Group>
            <Group>Queen</Group>
          </Collaboration>
        </Song>
        <Song>
          <Title>Perfect Afternoon</Title>
          <Duration>2.19</Duration>
          <Collaborations>
            <Group>Smashing pumpkin</Group>
          </Collaborations>
        </Song>
        <Song>
          <Title>Perfect Nightbp!</Title>
          <Duration>2.19</Duration>
          <Collaborations>
            <Group>Metallica</Group>
            <Group>Green Day</Group>
          </Collaboration>
        </Song>
      </Songs>
    </CD>
  </Group>
  <Group>
    <Title>Live9</Title>
    <CD>
      <Title>Unperfected day</Title>
      <Songs>
        ....
      </Songs>
    </CD>
  </Group>
  <Group>
    ....
  </Group>
</Music-Store>

```

Figure 80: Xml sample document #3

D D3P UML Diagrams

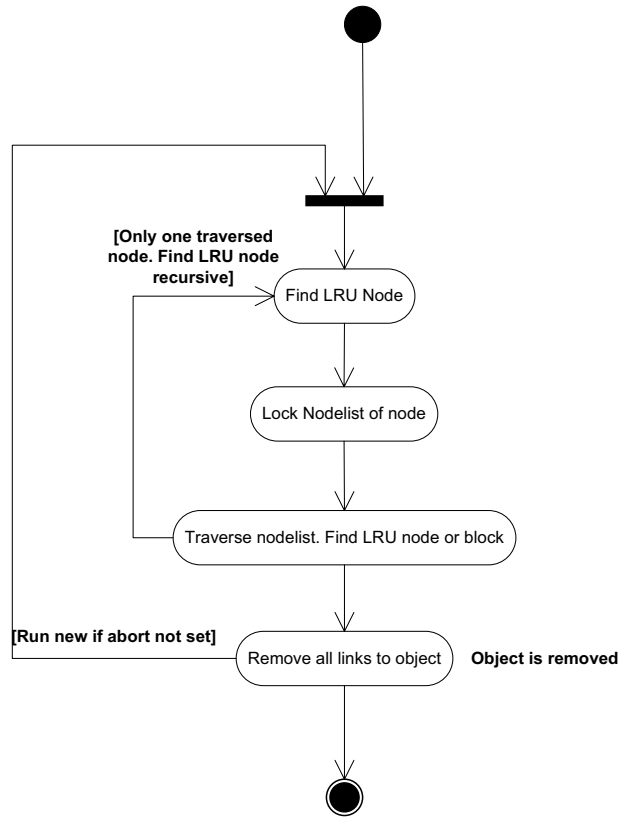


Figure 81: UML cache diagram

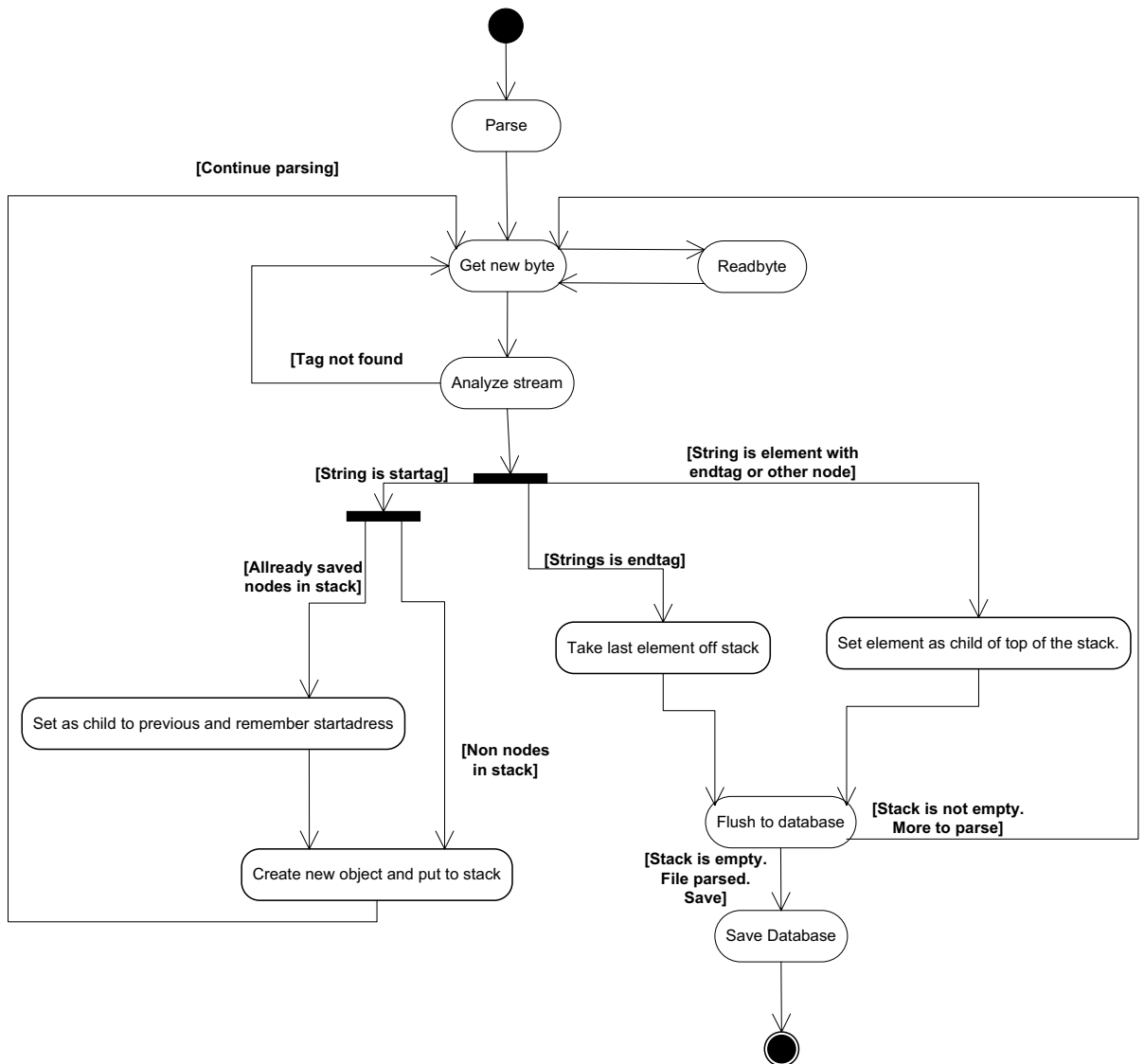


Figure 82: UML parser diagram

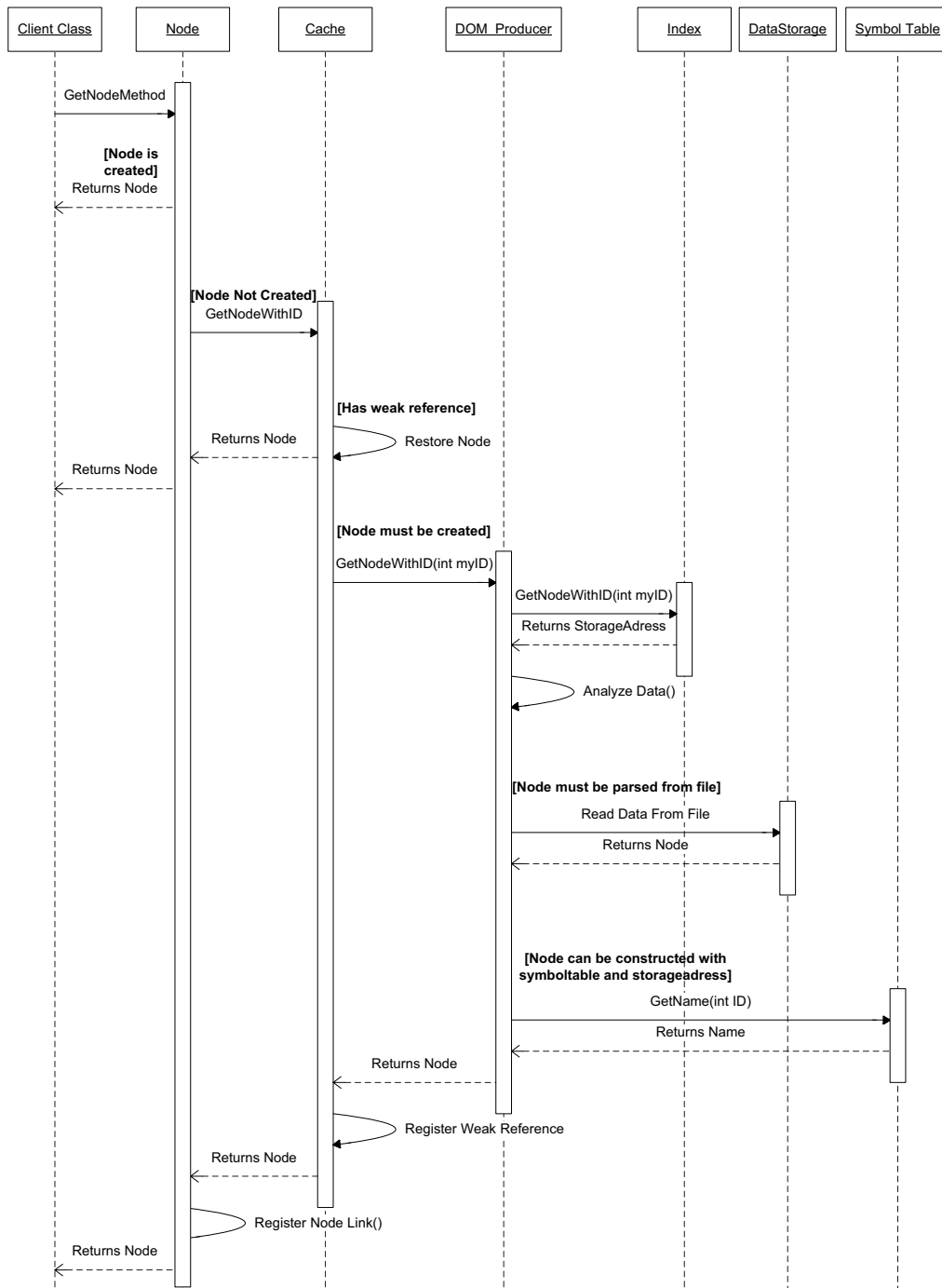


Figure 83: Generic example to get a node in the hierarchy

E W3C Standard

This appendix shows the extent of implementation the W3C DOM Core 1 recommendations. The listing below shows all interfaces in Core 1. For interfaces fully implemented, no comment are added. For interfaces where D3P has additional or missing functionality a + or - are used respectively to indicate this. Remaining interfaces are commented as needed. Interfaces which are not implemented in the D3P are simply commented with "not supported by D3P".

- Attr
 - + ownerElement : introduced in level 2
- CDATASection : not supported by D3P
- CharacterData : the .NET String object is used for Characterdata
- Comment
- Document
 - + Load(string filename) : load a file.
 - + Save(string filename) : save the current XML document to a file.
 - + ownerDocument - introduced in level 2
 - createCDATASection
 - createEntityReference
 - DocumentType
 - DOMImplementation
- DocumentFragment : not supported by D3P
- DocumentType : not supported by D3P
- DOMException
 - + INVALID_STATE_ERR : introduced in level 2
 - + SYNTAX_ERR : introduced in level 2
 - + INVALID_MODIFICATION_ERR : introduced in level 2
 - + NAMESPACE_ERR : introduced in level 2
 - + INVALID_ACCESS_ERR : introduced in level 2
- DOMImplementation : not supported by D3P
- Element
 - + bool hasAttributes : introduced in Level 2
 - getElementsByTagName : this function is only available from Document
- Entity : not supported by D3P
- EntityReference : not supported by D3P

- NamedNodeMap
- Node
 - + bool hasAttribute(string name) : introduced in level 2
 - + String innerXML(bool pretty) : Output data of child nodes
 - + String outerXML(bool pretty) : Output data of this and child nodes
 - + innerText : Write text content of all children
- NodeList
- Notation : not supported by D3P
- ProcessingInstruction
- Text

F Testing

F.1 Test Preliminary Settings

Filename	Elements	Attributes	Text node length	Depth	Size (KB)
Test-1.xml	1 * 5 * 2000	0	20	3	479
Test-2.xml	1 * 9 * 6000	0	60	3	4694
Test-3.xml	1 * 18 * 6000	0	60	3	9388
Test-4.xml	1 * 30 * 12000	0	100	3	52384
Test-5.xml	1 * 40 * 16000	0	160	3	118127
Test-6.xml	1 * 90 * 18000	0	180	3	330648
Test-7.xml	1 * (2^12)	0	20	12	544
Test-8.xml	1 * (2^15)	0	20	16	4928
Test-9.xml	1 * (2^16)	0	20	17	10240
Test-10.xml	1 * (2^18)	0	20	19	44032
Test-11.xml	1 * (2^19)	0	20	20	91136
Test-12.xml	1 * (2^21)	0	20	22	389120
Test-13.xml	1*10*(8^2)*4	8	20	5	569
Test-14.xml	1*10*(8^3)*4	8	20	6	4623
Test-15.xml	1*10*(8^4)	8	20	6	9005
Test-16.xml	1*10*(8^4)*5	8	20	7	46308
Test-17.xml	1*10*(8^4)*11	8	20	7	99325
Test-18.xml	1*10*(8^5)*4	8	20	8	303797
Test-19.xml	1*10^4*4*2	0	2000	5	644
Test-20.xml	1*10^4*4*2	0	15000	5	4706
Test-21.xml	1*10^8*4*2	0	15000	5	9411
Test-22.xml	1*30^8*4*4	0	15000	5	56428
Test-23.xml	1*60^8*4*4	0	15000	5	97817
Test-24.xml	1*180^8*4*4	0	15000	5	346175

Table 6: Testfiles for the XML parser test harness

Test Individual Settings		
File	Collapselimit	Maxtreesize
Test-1.xml	100KB	2MB
Test-2.xml	800KB	3MB
Test-3.xml	2MB	3MB
Test-4.xml	2MB	6MB
Test-5.xml	5MB	15MB
Test-6.xml	10MB	30MB
Test-7.xml	500KB	1MB
Test-8.xml	500KB	2MB
Test-9.xml	500KB	3MB
Test-10.xml	500KB	3MB
Test-11.xml	500KB	15MB
Test-12.xml	10MB	30MB
Test-13.xml	15KB	700KB
Test-14.xml	15KB	700KB
Test-15.xml	15KB	1MB
Test-16.xml	80KB	1MB
Test-17.xml	80KB	1MB
Test-18.xml	1MB	3MB
Test-19.xml	5KB	350KB
Test-20.xml	50KB	350KB
Test-21.xml	50KB	350KB
Test-22.xml	1MB	5MB
Test-23.xml	1MB	7MB
Test-24.xml	1MB	7MB

Table 7: Test settings

F2 Internal Performance Tests

Internal tests:

Test #	File	Reader	Database	Collapselimit	Use Cache	Cache alg.	Tree size	Parse	Build	Walk
LDXP-1	Test-1.xml	BufferReader	DB40	40KB	false	-	-	1281	78	1579
LDXP-2	Test-1.xml	BufferReader	Memory	40KB	false	-	-	109	16	671
LDXP-3	Test-1.xml	BufferReader	DB40	500KB	false	-	-	1265	63	62
LDXP-4	Test-1.xml	BufferReader	Memory	500KB	false	-	-	109	15	31
LDXP-5	Test-4.xml	BufferReader	DB40	1MB	false	-	-	7828	250	137391
LDXP-6	Test-4.xml	BufferReader	Memory	1MB	false	-	-	5813	578	69523
LDXP-7	Test-4.xml	FileReader	Memory	2MB	false	-	-	6123	422	141016
LDXP-8	Test-4.xml	BufferReader	Memory	2MB	false	-	-	4422	171	2907
LDXP-9	Test-4.xml	FileReader	Memory	2MB	false	-	-	4765	16	2969
LDXP-10	Test-4.xml	FileReader	Memory	15MB	true	Static	60	5594	15	7750
LDXP-11	Test-4.xml	FileReader	Memory	15MB	true	Tree	60	5625	16	7734
LDXP-12	Test-4.xml	FileReader	Memory	2MB	true	Static	25MB	6171	16	4922
LDXP-13	Test-4.xml	FileReader	Memory	2MB	true	Tree	25MB	6406	15	4579
LDXP-14	Test-4.xml	FileReader	Memory	2MB	true	Static	12MB	6235	16	6109
LDXP-15	Test-4.xml	FileReader	Memory	2MB	true	Tree	12MB	6297	16	3406

Database memory handling:

Database	File	Collapselimit	Initial	Parse (B)	Build	Walk
Memory	Test-1.xml	40B	305248	1	3276	4065100
DB40	Test-1.xml	40B	318952	619944	17544	4103280
Memory	Test-1.xml	500KB	321220	514748	3172	3172044
DB40	Test-1.xml	500KB	316684	616624	18068	3176880
Memory	Test-4.xml	1MB	322372	57986060	3728	219022656
DB40	Test-4.xml	1MB	323528	53799044	250	220469584

Figure 84: Internal tests

TESTING

F.3 General Tests

When examining the test tables, observe that a blank field means non-measured because the test was not finished. A “-” means that the digit was too small to be shown with two decimals.

CRIMSON

FILENAME	Build Time	Walk Time	Modify Time	Output Time	Build Mem	Walk Mem
Test-1.xml	203	78	63	-	1,77	1,52
Test-2.xml	578	328	203	-	11,13	10,18
Test-3.xml	828	844	406	-	22,38	22,89
Test-4.xml	2 260	2 562	1 250	-	94,79	100,14
Test-5.xml	4 594	4 594	2 110	-	193,90	200,97
Test-6.xml					-	
Test-7.xml	250	78	47	-	1,55	1,41
Test-8.xml	1 688	375	156	-	12,66	11,74
Test-9.xml	922	765	297	-	26,08	26,39
Test-10.xml	2 875	3 219	1 157	-	105,52	111,67
Test-11.xml	6 109	6 547	2 390	-	214,45	224,41
Test-12.xml					-	
Test-13.xml	313	125	32	-	3,05	1,87
Test-14.xml	875	625	78	-	23,26	15,78
Test-15.xml	1 407	1 188	125	-	45,41	33,07
Test-16.xml	6 031	5 984	547	-	233,69	214,86
Test-17.xml					-	
Test-18.xml					-	
Test-19.xml	141	16	16	-	0,92	0,53
Test-20.xml	328	15	16	-	13,17	-3,79
Test-21.xml	1 641	31	15	-	26,24	-7,52
Test-22.xml	2 313	94	47	-	156,58	-45,16
Test-23.xml	4 110	156	47	-	271,55	-78,45
Test-24.xml					-	

XERCES DEFERRED

FILENAME	Build Time	Walk Time	Modify Time	Output Time	Build Mem	Walk Mem
Test-1.xml	188	94	78	93	1,79	1,56
Test-2.xml	485	516	266	468	11,16	10,39
Test-3.xml	687	703	703	922	22,19	20,81
Test-4.xml	2 094	3 156	1 234	4 937	94,81	101,40
Test-5.xml	4 000	5 797	2 562	12 844	193,84	207,09
Test-6.xml						
Test-7.xml	359	93	62	344	1,57	1,44
Test-8.xml	656	422	188	250	12,68	12,00
Test-9.xml	953	797	344	438	25,80	26,92
Test-10.xml	2 891	3 391	1 266	1 672	105,54	112,48
Test-11.xml	5 218	6 594	2 672	4 047	214,07	230,82
Test-12.xml						
Test-13.xml	312	141	31	94	3,07	1,82
Test-14.xml	906	594	63	328	23,21	14,13
Test-15.xml	1 328	1 093	109	640	45,43	33,38
Test-16.xml	5 594	5 812	516	2 938	229,60	139,35
Test-17.xml	11 000				497,28	
Test-18.xml						
Test-19.xml	187	15	15	390	0,94	0,53
Test-20.xml	469	16	15	407	13,20	-3,79
Test-21.xml	1 656	31	16	781	26,26	-7,52
Test-22.xml	2 313	219	47	5 437	156,60	-45,14
Test-23.xml	4 281	344	46	8 765	271,59	-78,43
Test-24.xml						

XERCES

FILENAME	Build Time	Walk Time	Modify Time	Output Time	Build Mem	Walk Mem
Test-1.xml	187	-	63	78	2,45	-0,00
Test-2.xml	781	16	218	406	16,93	-0,00
Test-3.xml	1 187	47	406	828	33,91	-0,08
Test-4.xml	4 531	172	1 422	4 344	153,86	-
Test-5.xml	8 296	328	2 469	9 703	322,39	-
Test-6.xml						
Test-7.xml	250	16	63	79	2,28	-0,00
Test-8.xml	891	32	203	250	19,12	-
Test-9.xml	1 484	47	375	406	38,95	-0,08
Test-10.xml	5 641	235	1 422	1 500	161,38	-
Test-11.xml	11 203	422	2 937	4 156	329,45	-
Test-12.xml						
Test-13.xml	329	15	31	78	2,58	-0,00
Test-14.xml	1 157	31	110	344	20,47	-
Test-15.xml	2 015	62	187	641	39,70	-0,08
Test-16.xml	9 297	344	890	3 000	202,39	-
Test-17.xml	21 985	688	1 969	6 813	432,08	-
Test-18.xml						
Test-19.xml	156	-	-	78	1,37	-0,00
Test-20.xml	438	-	16	359	9,31	-
Test-21.xml	1 640	-	16	719	18,57	-0,00
Test-22.xml	2 438	-	31	4 000	110,99	-
Test-23.xml	3 735	-	47	7 453	192,36	-0,00
Test-24.xml	13 125	31	1 094	45 485	680,72	-0,08

JDOM

FILENAME	Build Time	Walk Time	Modify Time	Output Time	Build Mem	Walk Mem
Test-1.xml	296	31	312	171	2,19	0,52
Test-2.xml	1 141	94	4 578	765	15,73	2,87
Test-3.xml	1 844	500	9 000	1 469	31,59	5,63
Test-4.xml	4 250	531	58 265	5 813	146,45	22,53
Test-5.xml	7 968	906	137 719	11 172	310,23	38,63
Test-6.xml						
Test-7.xml	328	15	31	125	2,11	0,31
Test-8.xml	938	109	156	547	17,09	3,52
Test-9.xml	1 500	218	297	1 047	34,97	6,86
Test-10.xml	5 344	875	1 172	4 063	145,34	28,41
Test-11.xml	10 734	1 782	2 360	8 234	297,47	56,82
Test-12.xml						
Test-13.xml	375	16	16	141	2,33	0,05
Test-14.xml	1 281	47	94	750	18,88	0,32
Test-15.xml	2 359	94	172	1 422	36,75	0,19
Test-16.xml	10 250	438	844	7 015	186,92	2,93
Test-17.xml	23 891	828	1 875	15 109		
Test-18.xml						
Test-19.xml	172	-	-	78	1,32	0,02
Test-20.xml	313	-	-	187	9,26	0,02
Test-21.xml	1 671	-	-	375	18,51	0,05
Test-22.xml	2 032	16	15	3 203	110,83	0,26
Test-23.xml	3 297	16	32	8 516	192,12	0,46
Test-24.xml						

TESTING

DOM4J

FILENAME	Build Time	Walk Time	Modify Time	Output Time	Build Mem	Walk Mem
Test-1.xml	313	-	47	94	1,94	-
Test-2.xml	797	47	1 047	563	28,29	-0,07
Test-3.xml	2 536	141	1 547	4 735	135,52	-
Test-4.xml	3 954	141	6 188	3 610	135,52	-
Test-5.xml	7 594	250	14 234	9 969	290,62	-
Test-6.xml					-	-
Test-7.xml	312	-	16	47	2,02	-
Test-8.xml	938	31	110	187	16,89	-
Test-9.xml	1 422	62	235	375	34,46	-0,06
Test-10.xml	1 109	16	516	297	14,14	-
Test-11.xml	10 672	562	1 938	3 344	293,46	-
Test-12.xml					-	-
Test-13.xml	391	-	16	47	2,33	-
Test-14.xml	1 172	16	78	250	18,40	-
Test-15.xml	2 110	31	140	469	35,51	-0,06
Test-16.xml	9 094	188	703	3 000	181,12	-
Test-17.xml	20 485	391	1 547	8 765	386,57	-
Test-18.xml						
Test-19.xml	172	-	-	32	1,37	-
Test-20.xml	313	-	-	141	9,31	-
Test-21.xml	1 703	-	-	266	18,55	-
Test-22.xml	2 063	-	15	2 968	110,80	-
Test-23.xml	3 297	-	15	7 703	192,02	-
Test-24.xml	11 468	16	47	21 766	679,47	-0,07

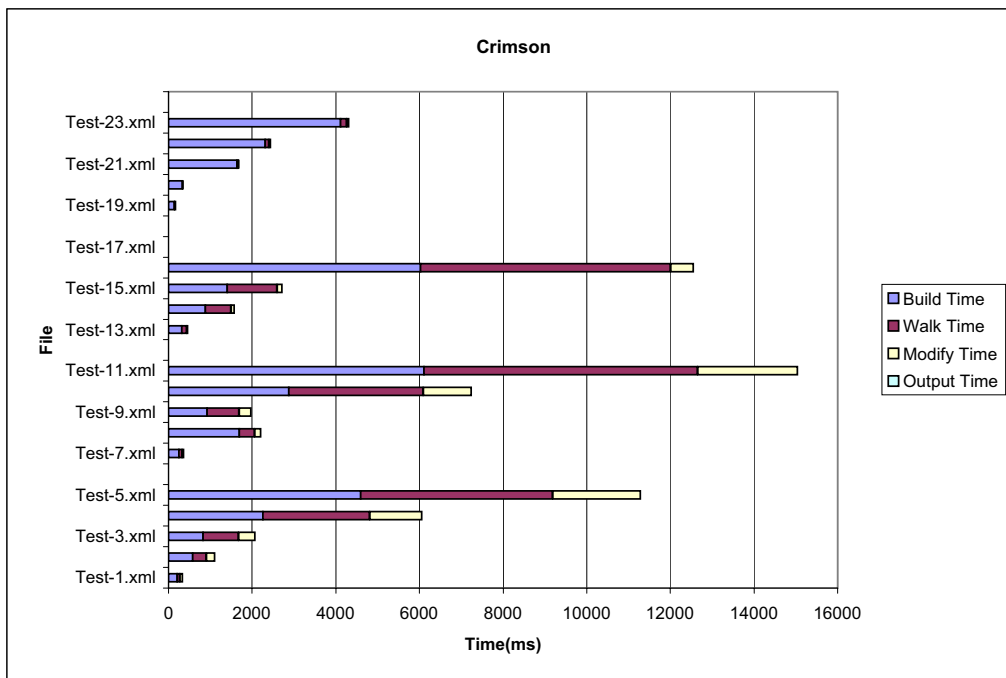
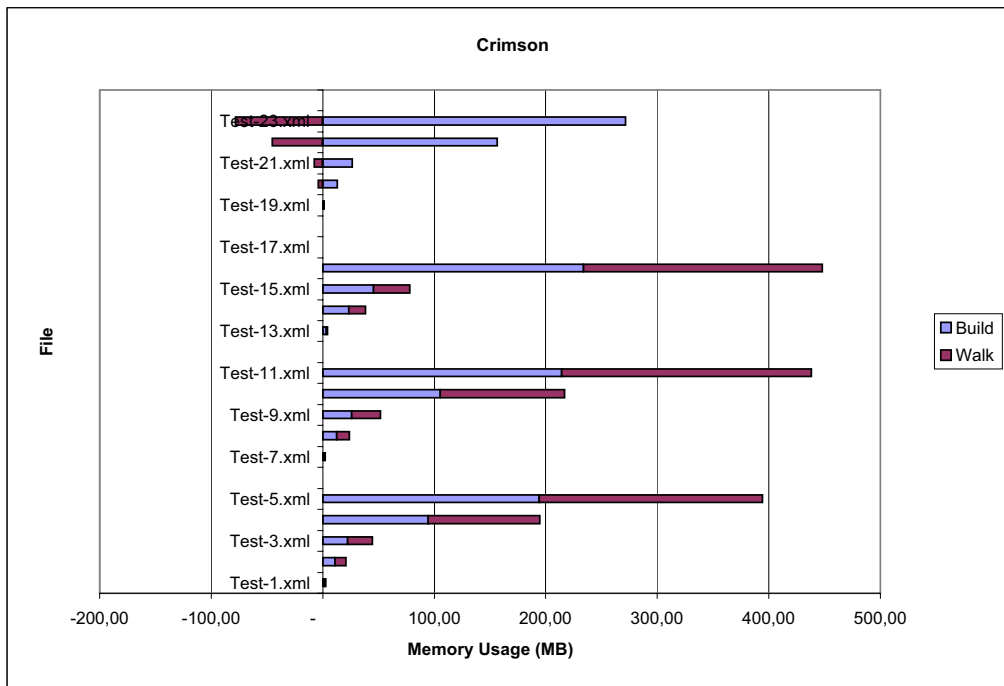
C#

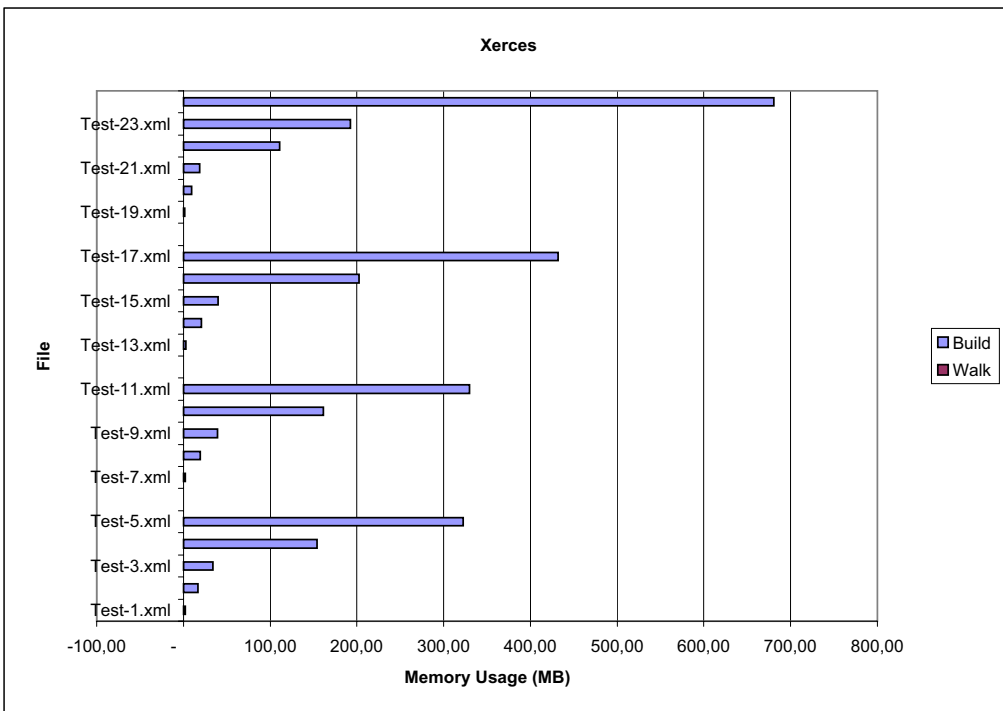
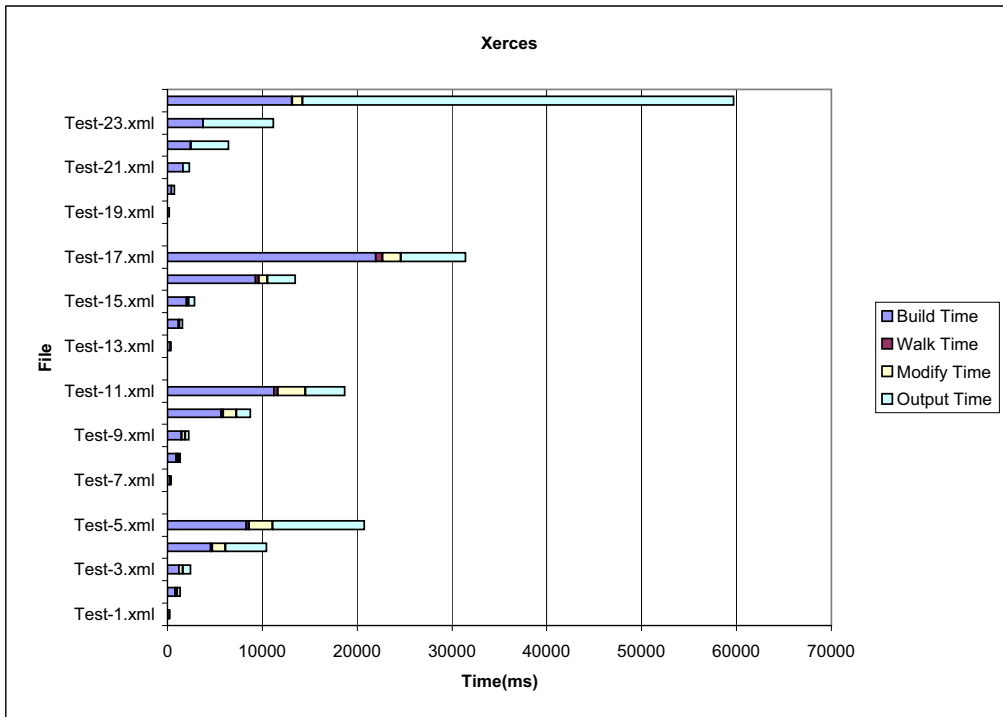
FILENAME	Build Time	Walk Time	Modify Time	Output Time	Build Mem	Walk Mem
Test-1.xml	63	16	31	47	1,04	0,00
Test-2.xml	266	31	234	266	9,73	0,00
Test-3.xml	390	47	515	532	19,44	0,00
Test-4.xml	2 203	219	1 938	5 687	107,57	0,00
Test-5.xml	5 078	438	3 828	13 422	236,83	0,00
Test-6.xml	81 312	1 078			661,25	0,00
Test-7.xml	484	16	31	110	0,54	0,00
Test-8.xml	328	17	156	266	4,26	0,00
Test-9.xml	437	47	406	531	8,51	0,00
Test-10.xml	3 047	172	1 594	5 156	34,01	0,00
Test-11.xml	6 453	328	4 344	16 187	68,01	0,00
Test-12.xml	15 750	1 562	13 938		272,01	0,00
Test-13.xml	594	15	16	47	1,89	0,00
Test-14.xml	531	16	172	265	15,06	0,00
Test-15.xml	1 265	63	281	719	29,96	0,00
Test-16.xml	4 469	297	1 890	5 563	150,17	0,00
Test-17.xml	10 156	656	6 625	16 265	328,84	0,00
Test-18.xml						
Test-19.xml	94	-	-	16	1,55	0,00
Test-20.xml	204	-	-	46	9,94	0,00
Test-21.xml	328	-	-	94	19,86	0,00
Test-22.xml	1 937	16	15	4 453	119,08	0,00
Test-23.xml	3 125	15	46	12 047	206,39	0,00
Test-24.xml	18 469	94	12 688	79 375	730,28	0,00

D3P

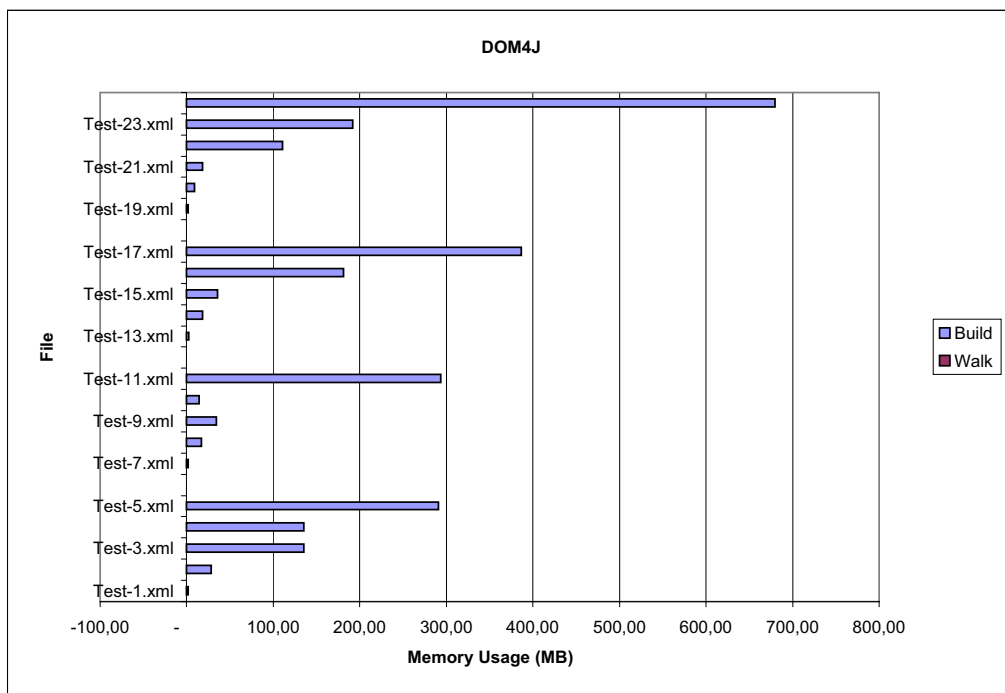
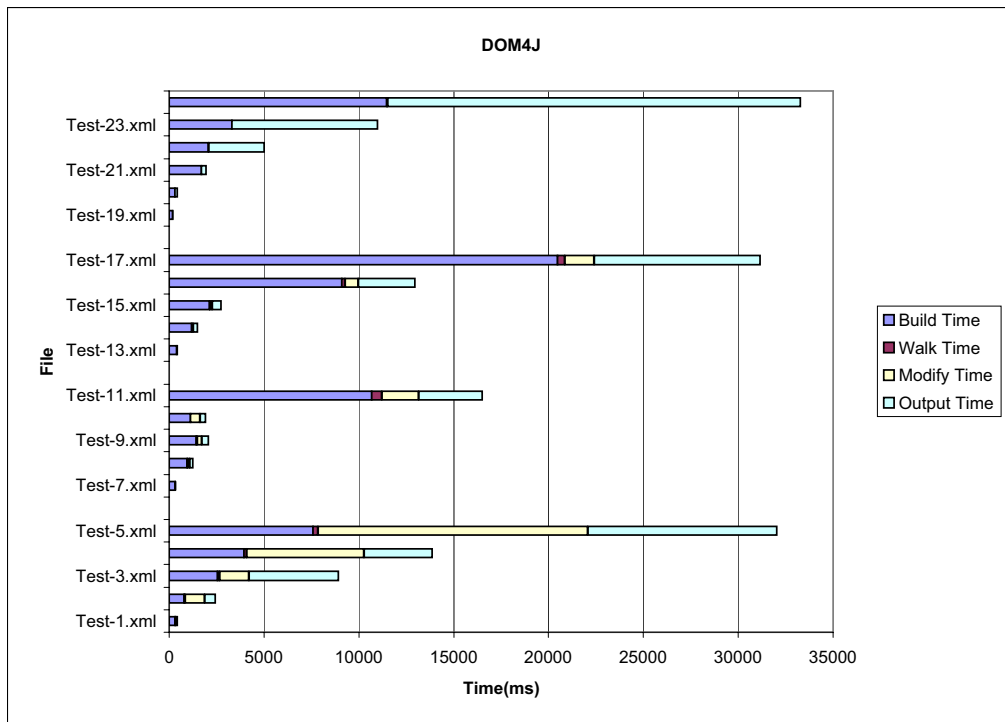
FILENAME	Parse Time	Build Time	Walk Time	Parse Mem	Build Mem	Walk Mem
Test-1.xml	109	16	47	0,03	-0,00	3,18
Test-2.xml	563	16	359	0,03	-0,00	16,58
Test-3.xml	1 359	16	1 172	0,03	-0,00	14,21
Test-4.xml	4 875	15	5 438	0,04	-0,00	18,48
Test-5.xml	10 016	15	9 110	0,04	-0,00	37,37
Test-6.xml	26 063	31	24 141	0,06	-0,00	22,51
Test-7.xml	171	16	203	0,03	-0,00	2,02
Test-8.xml	672	15	516	0,03	-0,00	1,01
Test-9.xml	1 344	15	1 032	0,04	-0,00	1,01
Test-10.xml	5 844	16	3 484	0,08	-0,00	5,06
Test-11.xml	11 922	31	7 469	0,14	-0,00	18,20
Test-12.xml	47 391	31	72 500	0,05	-0,00	16,13
Test-13.xml	219	47	187	0,04	-0,00	3,27
Test-14.xml	703	62	735	0,17	-0,00	3,28
Test-15.xml	1 172	94	1 687	0,22	-0,00	0,13
Test-16.xml	7 625	125	6 640	0,27	-0,00	0,53
Test-17.xml	14 442	297	16 750	1,76	-0,00	0,41
Test-18.xml	43 797	78	70 031	0,27	-0,00	3,19
Test-19.xml	172	63	62	0,03	-0,00	0,02
Test-20.xml	313	47	342	0,03	-0,00	8,09
Test-21.xml	657	75	469	0,04	-0,00	20,19
Test-22.xml	2 281	32	1 718	0,06	-0,00	36,20
Test-23.xml	3 547	16	2 484	0,09	-0,00	4,04
Test-24.xml	14 250	63	10 266	0,24	-0,00	8,11

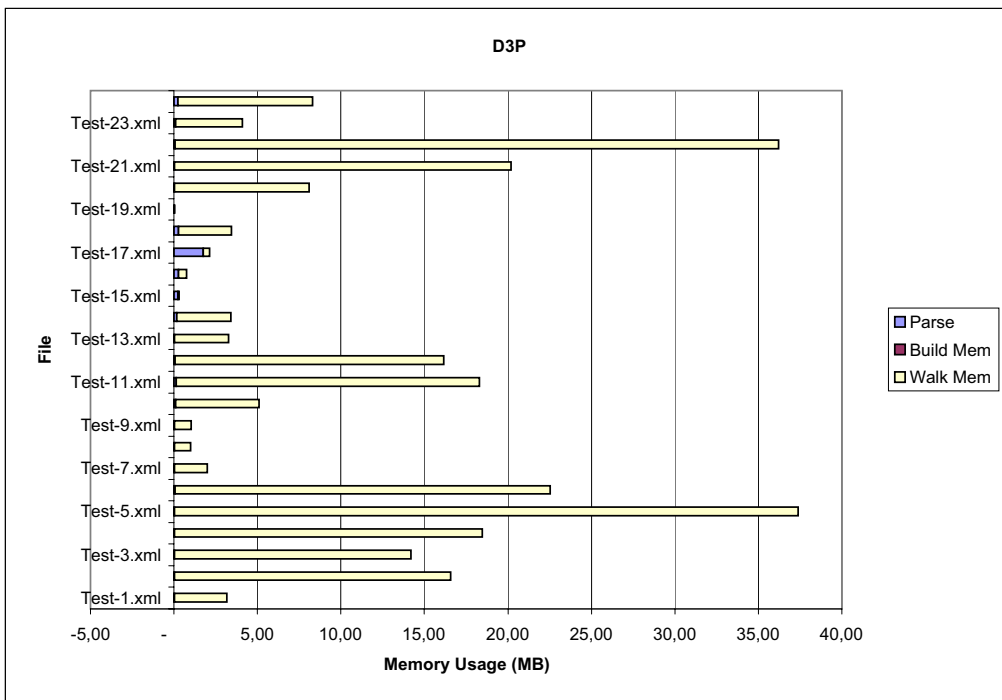
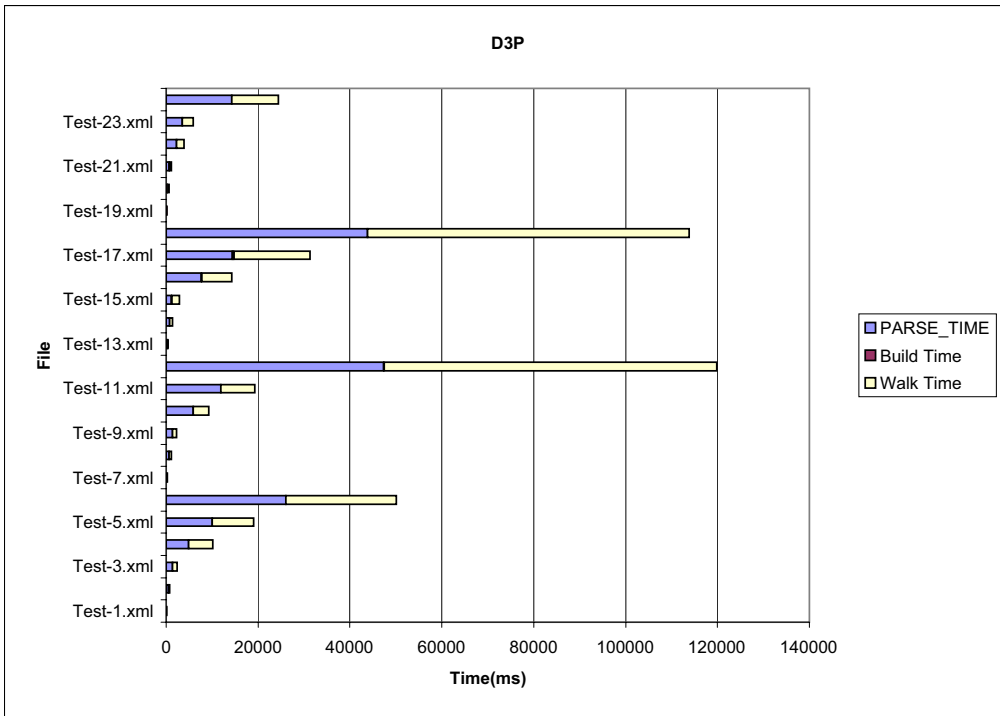
TESTING



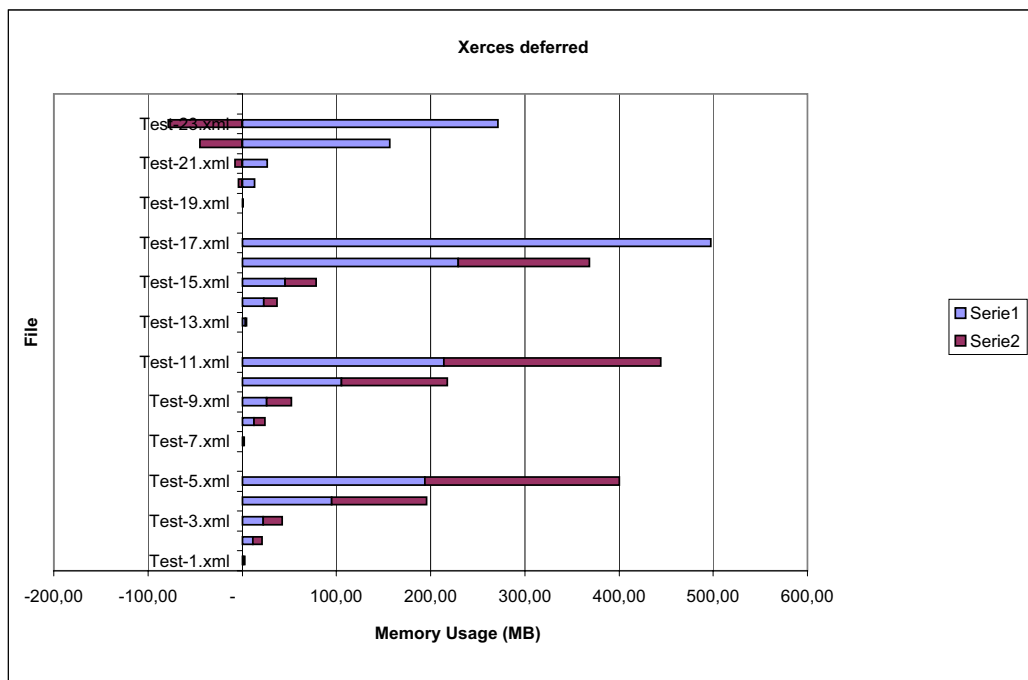
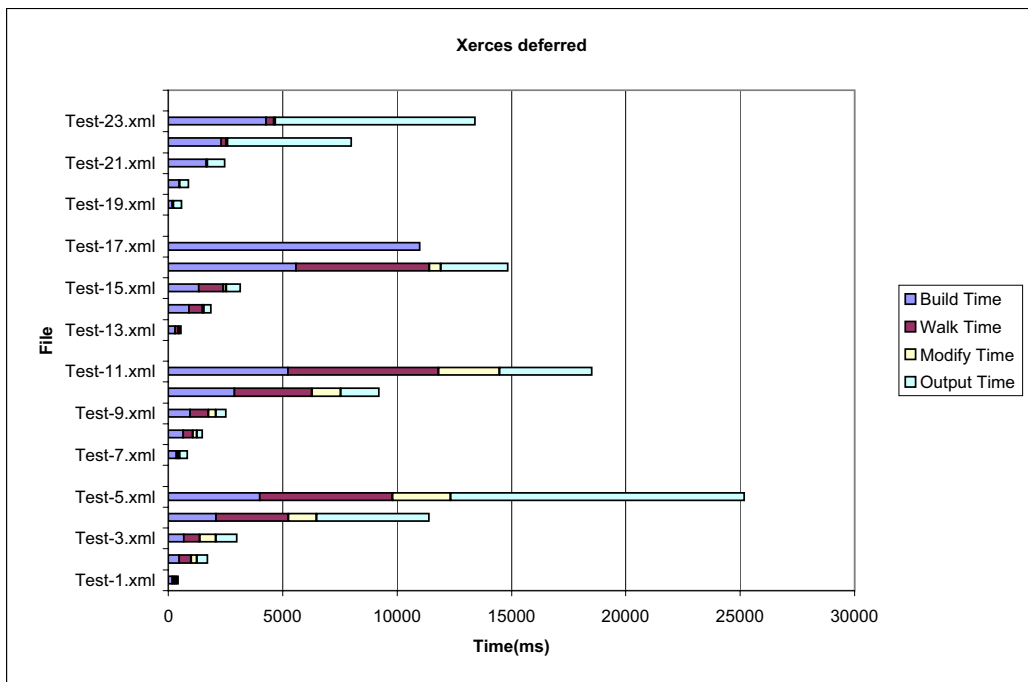


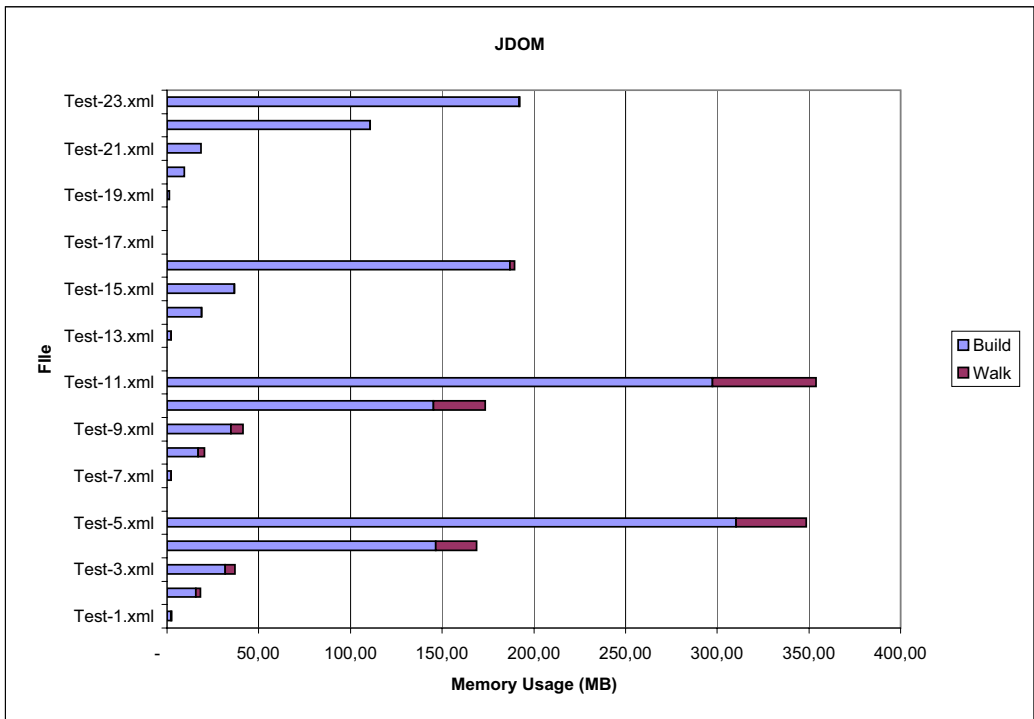
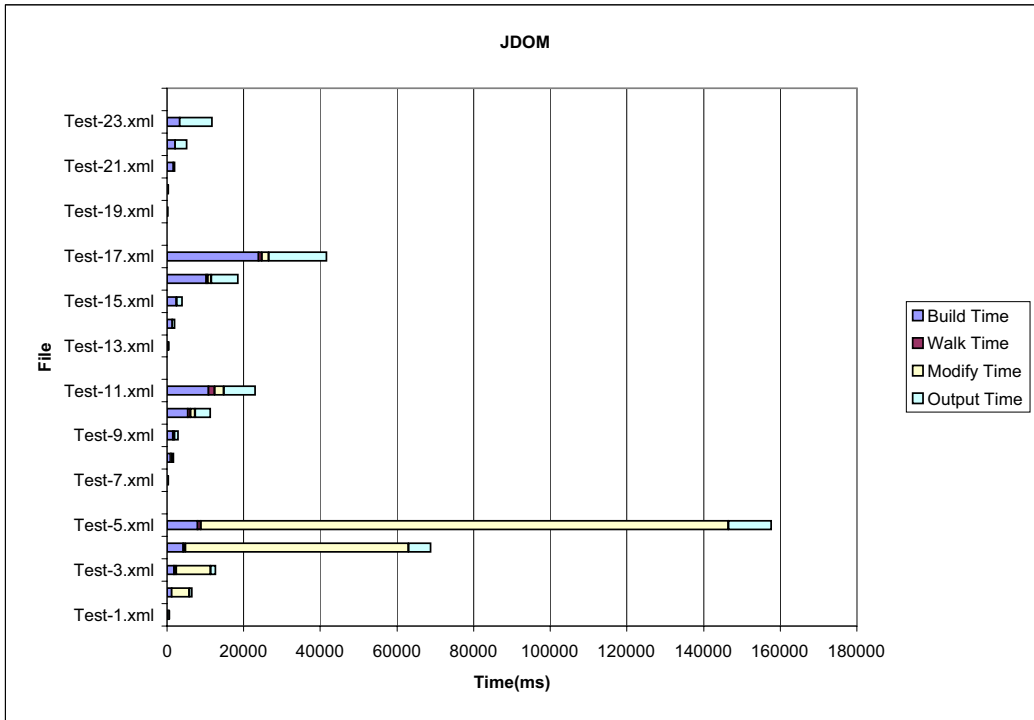
TESTING



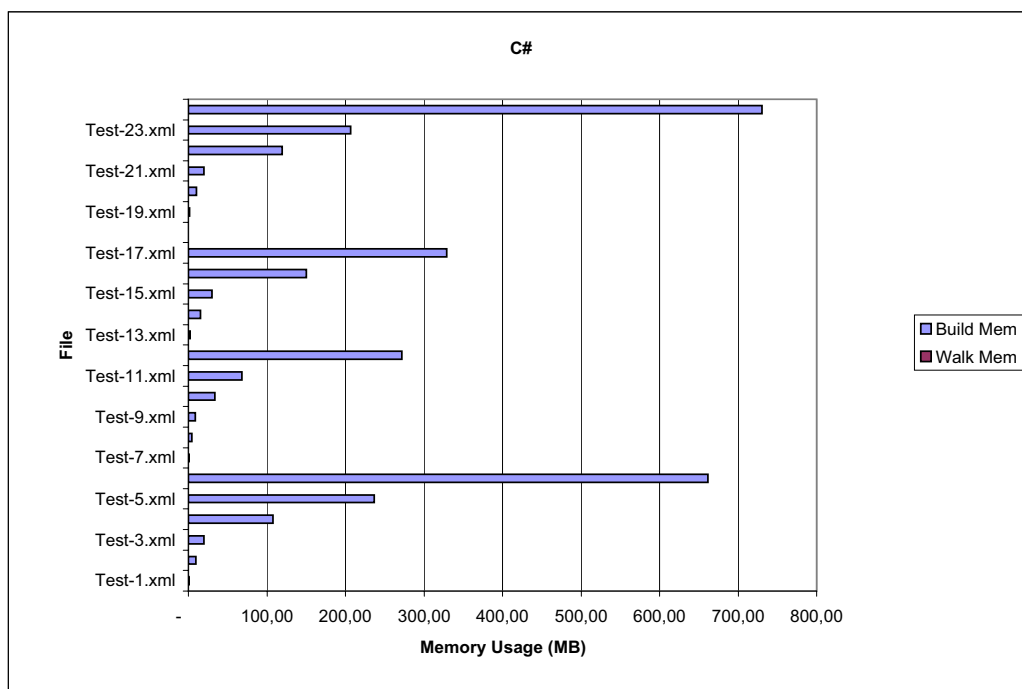
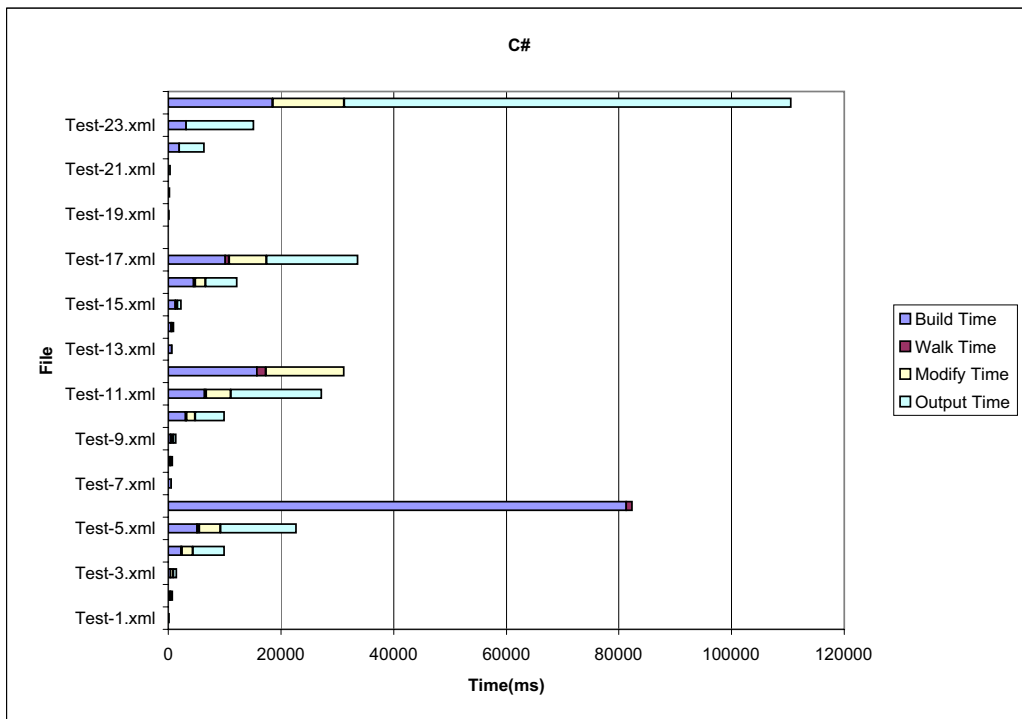


TESTING





TESTING



F.4 Special Tests

Partial Traversal Test:

	Parse		Build		Partial Traversal	
	Time(ms)	Memory(MB)	Time(ms)	Memory(MB)	Time(ms)	Memory(MB)
Parsed	-	-	5 594	230	344	20
XercesDef	-	-	9 297	202	47	-
Xerces	-	-	4 469	150	62	0
C#	-	-	125	-	5 538	5
D3P no parse	7 625	0	125	-	5 538	5
D3P						

Get Elements By Tagname Test:

	Parse		Build		Tag Test	
	Time(ms)	Memory(KB)	Time(ms)	Memory(KB)	Time(ms)	Memory(KB)
C=Collapselimit	-	-	5 594	230 000	2 469	101 169
XercesDef	-	-	9 297	202 000	375	-
Xerces	-	-	4 469	150 170	109	-
C#	-	-	21 578	1	2 344	1
D3P-1 - 1B C	24 844	21 119	1 609	1	828	1
D3P-2 - 1KB C	10 437	6 411	266	1	5 640	1
D3P-3 - 10KB C	6 969	1 355	47	1	6 000	1
D3P-4 - 80KB C	6 469	286	31	1	5 082	1
D3P-5 - 1MB C	5 797	62	21 578	21 109	2 344	1
D3P-1 - 1B C - No Parse			1 609	6 391	828	1
D3P-2 - 1KB C - No Parse			266	1 306	5 640	1
D3P-3 - 10KB C - No Parse			47	266	6 000	1
D3P-4 - 80KB C - No Parse			31	42	5 082	1
D3P-5 - 1MB C - No Parse						

F.5 Percentage Tests

Percentage Test				Xerces				C#				D3P					
XercesDef		Xerces		C#		D3P		XercesDef		Xerces		C#		D3P			
Time(ms)	Sum(ms)	Memory(B)	Time(ms)	Sum(ms)	Memory(B)	Time(ms)	Sum(ms)	Memory(B)	Time(ms)	Sum(ms)	Memory(B)	Time(ms)	Sum(ms)	Memory(B)	Time(ms)	Sum(ms)	Memory(B)
Parsed	5594	5594	214,07	9297	9297	203,25	4469	150,17	125	125	-	703	828	20,23	703	828	20,23
Build	265	5859	252,16	31	9328	203,25	32	4501	150,17	150,17	150,44	844	1672	9,90	844	1672	9,90
10 %	438	6297	273,47	37	9365	203,25	35	4536	150,44	150,44	150,45	671	2343	25,08	671	2343	25,08
20 %	469	6766	294,66	16	9381	203,25	31	4567	150,45	150,45	150,45	735	3078	5,52	735	3078	5,52
30 %	468	7234	316,12	31	9412	203,25	32	4599	150,45	150,45	150,45	719	3797	12,36	719	3797	12,36
40 %	469	7703	337,49	31	9443	203,25	36	4635	150,45	150,45	150,45	578	4375	32,47	578	4375	32,47
50 %	469	8172	358,72	21	9464	203,25	35	4670	150,45	150,45	150,45	766	5141	12,39	766	5141	12,39
60 %	469	8641	379,90	31	9495	203,25	33	4703	150,45	150,45	150,45	703	5844	7,01	703	5844	7,01
70 %	218	8859	401,85	31	9526	203,25	34	4737	150,45	150,45	150,45	703	6547	11,92	703	6547	11,92
80 %	2079	10938	383,25	31	9557	203,25	32	4769	150,45	150,45	150,45	984	7531	10,44	984	7531	10,44
90 %	531	11469	404,03	32	9589	203,25	37	4806	150,45	150,45	150,45						
100 %																	

Percentage Test With Different Cache Size

D3P-1				D3P-2				D3P-3			
Time(ms)	Sum(ms)	Memory(B)	Time(ms)	Sum(ms)	Memory(B)	Time(ms)	Sum(ms)	Memory(B)	Time(ms)	Sum(ms)	Memory(B)
125	125	-	125	125	-	125	125	-	125	125	-
703	828	20,23	734	859	32,76	688	813	32,81	688	813	32,81
844	1672	9,90	625	1484	64,33	656	1469	64,38	641	2110	95,92
671	2343	25,08	609	2093	95,88	641	2110	95,92	672	2782	127,52
735	3078	5,52	1031	3124	95,87	672	2782	127,52	672	3454	158,99
719	3797	12,36	719	3843	95,83	672	3454	158,99	656	4110	190,61
578	4375	32,47	594	4437	95,86	656	4110	190,61	641	4751	222,16
766	5141	12,39	766	5203	95,81	641	4751	222,16	766	5517	253,71
703	5844	7,01	563	5766	95,84	641	6158	285,32	641	6158	285,32
703	6547	11,92	750	6516	95,88	641	6158	285,32	718	6876	316,91
708	7255	10,44	563	7079	95,94	718	6876	316,91			

F.6 Deferred Overhead

File	Deferred time overhead : Build + Walk				Time Overhead	Memory Overhead
	Xerces		XercesDef			
	Time(ms)	Memory(M)	Time(ms)	Memory(MB)		
Test-1.xml	187	2,45	282	3,35	51 %	36 %
Test-2.xml	797	16,93	1001	21,55	26 %	27 %
Test-3.xml	1234	33,83	1390	42,99	13 %	27 %
Test-4.xml	4703	153,86	5250	196,21	12 %	28 %
Test-5.xml	8624	322,39	9797	400,92	14 %	24 %
Test-6.xml						
Test-7.xml	266	2,28	452	3,01	70 %	32 %
Test-8.xml	923	19,12	1078	24,67	17 %	29 %
Test-9.xml	1531	38,88	1750	52,71	14 %	36 %
Test-10.xml	5876	161,38	6282	218,02	7 %	35 %
Test-11.xml	11625	329,45	11812	444,89	2 %	35 %
Test-12.xml						
Test-13.xml	344	2,58	453	4,89	32 %	90 %
Test-14.xml	1188	20,47	1500	37,33	26 %	82 %
Test-15.xml	2077	39,62	2421	78,81	17 %	99 %
Test-16.xml	9641	202,39	11406	368,95	18 %	82 %
Test-17.xml	22673	432,08				
Test-18.xml						
Test-19.xml	156	1,37	202	1,48	29 %	7 %
Test-20.xml	438	9,31	485	9,41	11 %	1 %
Test-21.xml	1640	18,57	1687	18,75	3 %	1 %
Test-22.xml	2438	110,99	2532	111,46	4 %	0 %
Test-23.xml	3735	192,36	4625	193,15	24 %	0 %
Test-24.xml	13156	680,64				

File	Deferred time overhead : Build + Walk				Time Overhead	Memory Overhead
	C#		D3P			
	Time(ms)	Memory(M)	Time(ms)	Memory(MB)		
Test-1.xml	79	1,04	63	3,18	-20 %	205 %
Test-2.xml	297	9,73	375	16,57	26 %	70 %
Test-3.xml	437	19,44	1188	14,21	172 %	-27 %
Test-4.xml	2422	107,57	5453	18,48	125 %	-83 %
Test-5.xml	5516	236,83	9125	37,37	65 %	-84 %
Test-6.xml	82390	661,25	24172	22,51	-71 %	-97 %
Test-7.xml	500	0,54	219	2,02	-56 %	272 %
Test-8.xml	345	4,26	531	1,01	54 %	-76 %
Test-9.xml	484	8,51	1047	1,01	116 %	-88 %
Test-10.xml	3219	34,01	3500	5,06	9 %	-85 %
Test-11.xml	6781	68,01	7500	18,20	11 %	-73 %
Test-12.xml	17312	272,01	72531	16,13	319 %	-94 %
Test-13.xml	609	1,89	234	3,27	-62 %	73 %
Test-14.xml	547	15,06	797	3,28	46 %	-78 %
Test-15.xml	1328	29,96	1781	0,13	34 %	-100 %
Test-16.xml	4766	150,17	6765	0,52	42 %	-100 %
Test-17.xml	10812	328,84	17047	0,41	58 %	-100 %
Test-18.xml			70109	3,19		
Test-19.xml	94	1,55	125	0,02	33 %	-99 %
Test-20.xml	204	9,94	389	8,09	91 %	-19 %
Test-21.xml	328	19,86	544	20,19	66 %	2 %
Test-22.xml	1953	119,08	1750	36,20	-10 %	-70 %
Test-23.xml	3140	206,39	2500	4,04	-20 %	-98 %
Test-24.xml	18563	730,28	10329	8,11	-0,4435705	-99 %

Figure 85: Deferred overhead

G CD content

Complete listing of files on CD

G.1 D3P library

The Complete Demand Driven DOM Parser(D3P) source code, with MS Visual Studio 2005 project files.

D3P

- DocumentFactory

D3P.Common

- Cache.cs
- Common.cs
- Constants.cs
- DocumentRoot.cs
- DomProducer.cs
- Hashlist.cs
- NameTag.cs
- Parser.cs
- processingInstruction.cs
- Serializer.cs
- SpecType.cs
- StorageAdress.cs
- SymbolTable.cs

D3P.Common.DataStorages

- db40DataStorage.cs
- IIndexDataStorage.cs
- indexStorageProvider.cs
- memoryDataStorage.cs
- simpleStorage.cs

D3P.Common.FileStorages

- DomWriter.cs
- DomXmlWriter.cs
- fileStorageProvider.cs
- IxmlDataReader.cs
- simpleXmlDataReader.cs
- XmlDataBufferReader.cs
- XmlDataReader.cs

D3P.Common.GUI

- indexGUI.cs
- indexGUI.designer.cs
- indexGUI.resx
- Options.cs
- Settings.cs
- Settings.designer.cs

- Settings.resx
- XmlManager.cs
- XmlManager.designer.cs
- XmlManager.resx
- XmlViewerGUI.cs
- XmlViewerGUI.designer.cs
- XmlViewer.resx

D3P.Common.LruCache

- cacheAlgProvider.cs
- Cleaner.cs
- ICache.cs
- nodeTimedCache.cs
- SimpleCacheAlg
- StaticCache.cs
- TimedCache.cs

D3P.DOM

- AttrImpl.cs
- CDATASectionImpl.cs
- Comment.cs
- DocumentFragment.cs
- DocumentImpl.cs
- DocumentType.cs
- DOMExceptionImpl.cs
- Element.cs
- Entity.cs
- EntityReference.cs
- NamedNodeMap.cs
- NodeImpl.cs
- NodeListImpl.cs
- NodeTypes.cs
- Notation.cs
- ProcessingInstruction.cs
- TextImpl.cs

D3P.Interfaces

- Attr.cs
- CDATASection.cs
- CharacterData.cs
- Comment.cs
- DocumentFragment.cs
- Document.cs
- DocumentType.cs
- DOMException.cs
- DOMImplementation.cs
- Element.cs
- Entity.cs
- EntityReference.cs

- NamedNodeMap.cs
- Node.cs
- NodeList.cs
- Notation.cs
- ProcessingInstruction.cs
- Text.cs

D3P.Simple

- SimpleCharacterData.cs

G.2 .NET Test Suite

Test suite for the D3P and C# parsers, with MS Visual Studio 2005 project files.

TestApplication

- CSharpTest.cs
- D3PTest.cs
- simpleTest.cs
- DocumentSummary.cs
- Program.cs

G.3 Java Test Suite

Test suite for the Java parsers.

default package

- Constants.java
- Crimson.java
- Xerces.java
- DOM4J.java
- JDOM.java
- Crimson.jar
- Xerces.jar
- DOM4J.jar
- JDOM.jar
- DocumentSummary.cs
- TESTBENCH.cs

G.4 D3P Console Parser

Console application to start a console indexer. Look in Release catalog for sample .bat files to run. Run HELP.bat for introduction and look at run.bat for a general index. Comes with MS Visual Studio 2005 project files.

D3P Console Parser

- Program.cs

G.5 D3P GUI Parser

The GUI parser used to adjust settings and index files. It does the same as the console parser in appendix refDCP but with a graphical user interface. The MS Visual Studio 2005 project files are attached, and the binary is located in the bin/Release folder.

D3P GUI Parser
Program.cs

G.6 XML File Generator

XML filegenerator for the generated XML files

TestFileGenerator
Program.cs

G.7 D3P Library Documentation

D3P Documentation in PDF and HTML format.

- doc.pdf
- doc.html

G.8 Testfiles

Folder containing all testfiles used.
Testfiles Test-(1-24).xml