University of Cincinnati

Date: 6/26/2012

I, Neha Gujarathi, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:
A Performance Based Comparative Study of Different APIs Used for Reading and Writing XML Files

Student's name: Neha Gujarathi

This work and its defense approved by:

Committee chair: Carla Purdy, PhD

Committee member: Raj Bhatnagar, PhD

Committee member: George Purdy, PhD
A Performance Based Comparative Study of Different APIs Used for Reading and Writing XML Files

A Thesis submitted to the

Graduate School at the

University of Cincinnati

In Partial Fulfillment of the

requirements for the Degree of

MASTER OF SCIENCE

In the School of Electronics and Computing Systems

of the College of Engineering and Applied Sciences

By

Neha Gujarathi

Bachelor of Engineering (B. E.), 2009

University of Pune, India

Committee Chair: Dr. Carla Purdy
ABSTRACT

Recently, XML (eXtensible Markup Language) files have become of great importance in business enterprises. Information in the XML files can be easily shared across the web. Thus, extracting data from XML documents and creating XML documents become important topics of discussion. There are many APIs (Application Program Interfaces) available which can perform these operations. For beginners in XML processing, selecting an API for a specific project is a difficult task. In this thesis we compare various APIs that are capable of extracting data and / or creating XML files. The comparison is done based on the performance time for different types of inputs which form different cases. The codes for all the different cases are implemented. Two different systems, one with Windows 7 OS and another with Mac OS are used to perform all the experiments. Using the results found we propose a suitable API for a given condition. In addition to the performance, programming ease for these APIs is taken into consideration as another aspect for comparison. To compare the programming ease, aspects such as number of lines of code, complexity of the code and complexity of understanding the coding for the particular API are considered. Thus, we are also able to suggest an appropriate API based on programming ease.
ACKNOWLEDGMENTS

I would like to first and foremost thank God, for granting me the opportunity to do this research. I am very much thankful to my advisor Dr. Carla Purdy who gave me the timely advice and suggestions I needed. She helped and motivated me throughout the research which has made this thesis a success. I would also like to thank Dr. George Purdy and Dr. Raj Bhatnagar who agreed to be a part of the thesis committee. I am grateful to my friends and colleagues at the University of Cincinnati for their constant help and support.

The idea for this dissertation was conceived during an internship and I would like to extend my gratitude to my colleagues at Cummins Inc. Especially Mr. Kyle Fath and Mr. Ken Lanteigne, without whose guidance and valuable assistance, I would not have been able to run various experiments and helping in the preparation and completion of this study.

Last but not the least I would like to thank my husband, my family and friends who kept me motivated and without their support this thesis would not be possible.
Table of Contents

1. INTRODUCTION ............................................................................................................. 01
   1.1 Motivation ................................................................................................................. 01
   1.2 Thesis goals ............................................................................................................... 01
   1.3 Document organization ............................................................................................ 02

2. BACKGROUND .............................................................................................................. 03
   2.1 What are XML files? .................................................................................................. 03
   2.2 XML parsing and its importance .............................................................................. 07
   2.3 XML parsing APIs ................................................................................................... 07
      2.3.1 SAX API ........................................................................................................... 08
      2.3.2 DOM API ....................................................................................................... 12
      2.3.3 StAX API ....................................................................................................... 15
      2.3.4 DOM4J API ................................................................................................. 19
      2.3.5 JDOM API ................................................................................................. 21
      2.3.6 XPath ........................................................................................................... 24
   2.4 Work related to the thesis done in past ..................................................................... 26

3. METHODOLOGY .......................................................................................................... 30
   3.1 Which programming platform should be used? ...................................................... 31
   3.2 Which APIs should be considered? .......................................................................... 31
   3.3 Which experiments should be performed? .............................................................. 34
      3.3.1 Find the size of XML documents to be considered ........................................... 34
      3.3.2 Find the important elements in an XML file that are extracted often ............... 35
      3.3.3 Find the XML files used as an input to the programs ....................................... 36
      3.3.4 Using the Cummins Inc. project XML files as input ......................................... 37
   3.4 What other factors should be considered other than the experiments to compare the
       APIs? .......................................................................................................................... 38
   3.5 Experimental setup .................................................................................................. 38
4. RESULTS

4.1 Experiments in case A and B performed on Windows 7 system

4.1.1 Case A1

4.1.2 Case A2

4.1.3 Case A3

4.1.4 Graph plotted for average execution time against file size for all the APIs

4.1.5 Case A4

4.1.6 Case B

4.2 Experiments in case A and B performed on Mac OS system

4.2.1 Case A1

4.2.2 Case A2

4.2.3 Case A3

4.2.4 Graph plotted for average execution time against file size for all the APIs

4.2.5 Case A4

4.2.6 Case B

4.3 Comparing the performance of two systems

4.4 Comparison based on programming ease and understanding the API

4.5 Summary of experimental results

4.6 Comparing the work in this thesis to related previous work

5. CONCLUSIONS AND FUTURE WORK

REFERENCES
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example XML document</td>
<td>05</td>
</tr>
<tr>
<td>2.2</td>
<td>Tree representation of the XML document</td>
<td>06</td>
</tr>
<tr>
<td>2.3</td>
<td>The parsing process is controlled by the parser and code listens for events, responding as they occur [6]</td>
<td>09</td>
</tr>
<tr>
<td>2.4</td>
<td>SAX APIs [10]</td>
<td>11</td>
</tr>
<tr>
<td>2.5</td>
<td>DOM document tree structure [6]</td>
<td>13</td>
</tr>
<tr>
<td>2.6</td>
<td>DOM APIs [10]</td>
<td>14</td>
</tr>
<tr>
<td>2.7</td>
<td>XMLStreamWriter Interface [5]</td>
<td>18</td>
</tr>
<tr>
<td>2.8</td>
<td>UML model of core JDOM classes [6]</td>
<td>20</td>
</tr>
<tr>
<td>2.9</td>
<td>DOM4J core interfaces [6]</td>
<td>22</td>
</tr>
<tr>
<td>2.10</td>
<td>Node access methods on Branch, Document, and Element interfaces [6]</td>
<td>23</td>
</tr>
<tr>
<td>2.11</td>
<td>JAXP XPath Interfaces [6]</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Case A1 for Windows 7 system</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Case A2 for Windows 7 system</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Case A3 for Windows 7 system</td>
<td>47</td>
</tr>
<tr>
<td>4.4</td>
<td>Chart plotted for average execution time against file size for all the APIs combining cases A1, A2 and A3 for Windows 7 system</td>
<td>48</td>
</tr>
<tr>
<td>4.5</td>
<td>Case A4 for Windows 7 system</td>
<td>50</td>
</tr>
<tr>
<td>4.6</td>
<td>Case B for Windows 7 system</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>Case A1 for Mac OS system</td>
<td>54</td>
</tr>
<tr>
<td>4.8</td>
<td>Case A2 for Mac OS system</td>
<td>56</td>
</tr>
<tr>
<td>4.9</td>
<td>Case A3 for Mac OS system</td>
<td>58</td>
</tr>
<tr>
<td>4.10</td>
<td>Chart plotted for average execution time against file size for all the APIs combining cases A1, A2 and A3 for Mac OS system</td>
<td>59</td>
</tr>
<tr>
<td>4.11</td>
<td>Case A4 for Mac OS system</td>
<td>61</td>
</tr>
<tr>
<td>4.12</td>
<td>Case B for Mac OS system</td>
<td>63</td>
</tr>
<tr>
<td>4.13</td>
<td>Sample XML document</td>
<td>65</td>
</tr>
<tr>
<td>4.14</td>
<td>SAX API example code</td>
<td>66</td>
</tr>
</tbody>
</table>
List of Tables

Table 2.1: StAX API- Event Types [6].................................................................17
Table 2.2: Comparison of XML parser APIs [1].................................................27
Table 2.3: XML processing performance characteristics [20]...............................29
Table 3.1: APIs and types....................................................................................33
Table 4.1: List of experimental cases and the corresponding APIs used..............41
Table 4.2: Case A1 for Windows 7 system.........................................................42
Table 4.3: Case A2 for Windows 7 system ..........................................................44
Table 4.4: Case A3 for Windows 7 system ..........................................................46
Table 4.5: Case A4 for Windows 7 system ..........................................................49
Table 4.6: Case B for Windows 7 system ............................................................51
Table 4.7: Case A1 for Mac OS system...............................................................53
Table 4.8: Case A2 for Mac OS system ...............................................................55
Table 4.9: Case A3 for Mac OS system ...............................................................57
Table 4.10: Case A4 for Mac OS system .............................................................60
Table 4.11: Case B for Mac OS system ...............................................................62
Table 4.12: Comparison of systems 1 vs. system 2............................................64
Table 4.13: APIs and corresponding programming properties.............................72
1. INTRODUCTION

1.1 Motivation

In this work, we are comparing different Application Program Interfaces (API) that are used for reading and creating eXtensible Markup Language (XML) documents. This idea originated from an internship experience at Cummins Inc. One of the projects at Cummins Inc. was to extract information from multiple XML files and write that information into a text file. For this project, a comparison of the performance of the most commonly occurring APIs would be really useful.

XML files have recently become very important in terms of data exchange via the web [1]. In many industries, XML files are used to store data. So it becomes important to retrieve this data for various operations. This retrieval can be performed by various APIs available in the market. It becomes difficult for beginners in XML processing fields to choose a particular API for their application. Hence a good comparison of the commonly occurring APIs can help people new to this field.

1.2 Thesis goal

The goal of this thesis is to choose a list of most commonly occurring APIs for comparison. Then we devise a set of experiments which will help in comparing these APIs to get precise results. After coming up with the set of cases, experiments will be performed. Thereafter evaluation of the results is done to conclude which API behaves best for a given scenario.
1.3 Thesis organization

The thesis is organized into 5 chapters.

- Chapter 1 contains the motivation for coming up with this topic and the goals of the thesis.

- Chapter 2 comprises the background study for understanding this topic. It starts with explaining the XML documents, basic components of the XML documents etc. It then describes the XML parsing process. After that different APIs are discussed in detail. The basic working of APIs is also explained.

- In Chapter 3 all the questions that we wish to answer are stated and the experimental design is explained.

- Chapter 4 contains all the experimental results along with charts which help in analyzing the results.

- Chapter 5 contains the conclusions and describes what future work can be done.
2. BACKGROUND

There are a number of Application Program Interfaces (API) existing in Java which support interaction with the eXtensible Markup Language (XML). XML is a format that is not easily readable. Hence many applications need to retrieve data from the XML files to make it usable. There are various APIs available which are capable of performing such tasks. The goal here is to compare the performance of these APIs while retrieving data from the XML files and also to suggest different scenarios in which each will work best.

2.1 What are XML files?

A group originally known as SGML Editorial Review Board developed XML [2]. That group today is known as XML Working Group. It was formed under the umbrella of the World Wide Web Consortium (W3C) in 1996. Jon Bosak of Sun Microsystems led the group with the active participation of an XML Special Interest Group (previously known as the SGML Working Group) [2].

Today the World Wide Web (WWW) has become an important entity which has made searching, manipulation and exchange of information a crucial part of everyone’s lives. Hyper Text Markup Language (HTML) and eXtensible Markup Language (XML) have been developed to meet these needs. XML is a subset of the Standard Generalized Markup Language (SGML). It is a data description language. XML allows users to describe their own set of markup tags depending upon the document content. It also brings structured information to the internet. This makes
XML extensible and supports validation. It is a different methodology compared to fixed markup tags used in HTML [3].

The main purpose of creating XML is to use structured documents via the web. HTML and SGML could be used via the web, but with HTML any arbitrary structure cannot be used and with SGML, although arbitrary structures can be used, using them just for a web browser is difficult [4].

An XML file should always be well formed. But it is not necessary for it to be validated [5]. Definitions of a well-formed and validated XML files are given below.

- **Well-formed XML document**
  A grammar is defined for XML documents, which specifies things like position of tags, element names, how to attach attributes to elements etc. [5]. This grammar supports construction of XML parsers which read the XML documents. If an XML document follows all the rules defined in the grammar, then the XML file is called well-formed. If the document isn’t well-formed, the XML processor will reject it.

- **Valid XML document**
  The structure of a document is described by markup in the XML document. The markup tells us about the association between the elements. For particular applications, there are some constraints related to which markup is permitted. This is usually documented in a schema. XML documents belonging to that application can be compared to that schema. If the document
matches the schema, it is a valid document, else it is invalid. But it isn’t as necessary for an XML document to be valid as it is to be well-formed [5].

```xml
<?xml version="1.0"?>
<Information>
  <Name>
    <First_name>Neha</First_name>
    <Last_name>Gujarathi</Last_name>
  </Name>
  <Department>Electrical and Computer Engineering</Department> <Degree>M.S. Computer Engineering</Degree>
</Information>
```

Figure 2.1 Example XML document

Figure 2.1 shows a well-formed XML document. In this document the first statement is the declaration statement. It displays the version number of XML used and it can have some other information too.

The other basic parts of a XML file are:

- **Element** [6]: Elements are delimited by angular brackets. The name of the element usually gives an idea about the content of it. A non-empty element will begin with `<element>` which is called the start tag and end with `</element>` which is called end tag. An empty element can be shown as `<element/>`. For example, in Figure 2.1, “Name”, “Information”, “First_name”, “Last_name”, “Department” and “Degree” are all elements.
• **Attribute** [6]: These usually occur after the element name in a tag, the attribute also has a corresponding value. In the statement below, First and Last are attributes.

```xml
<Name First="Neha" Last="Gujarathi"/>
```

• **Comment** [5]: In XML files, begin syntax for a comment is `<!--` and the comment ends with `-->`.

Also every XML file can be shown as a tree structure [5]. The tree structure for the XML document in Figure 2.1 is shown below in Figure 2.2.

![Figure 2.2 Tree representation of the XML document](image)

The format of an XML document is very simple, well-documented and forthright. The data in the XML files is portable data. XML documents can be used across any platform. Also an XML document can be read with any tool which reads text since it is in text form. The mark-up tags format is also text. There is nothing complex about the XML files. The tag names tell us what the
data contains. Because of the simplicity of XML documents, they have been very widely used in business enterprises [5].

2.2 XML parsing and its importance

Currently industries use XML files for permanently storing large amounts of data and XML files have not just remained a format for data interchange [7]. And wherever these files are used, data from these files needs to be extracted and manipulated often. An XML parser is used to read the XML document. The document is divided into different parts such as element, attributes, etc., by these parsers. Then this information is passed to the application which needs it. In some cases, if the document is not well formed, the parser will send an error and stop parsing. In other cases, it will parse the whole document and send all the errors together. But if the error is encountered, it will only give information about the error and not the contents of the XML document [5].

XML parsing can seriously affect overall performance for any project it is used in [7]. Hence it is very important to study different APIs which have the capability of XML parsing and to compare the performance of the APIs to choose the best method for a given condition.

2.3 XML parsing APIs

There are various APIs for XML parsing available in the market [7]. XML parsers can be divided into 2 major types:
• Tree based APIs [8]:

These APIs create a tree representation of the XML document in the memory. Data can be accessed by navigating through this tree. Document Object Model (DOM) is a tree based API. There are many other tree based APIs available.

• Event based APIs [8]:

These APIs do not build a tree representation. They report different parsing events like start element, end element using the call back method. Handlers are designed to deal with the events. Simple API for XML (SAX) is the most common type of event-based API.

Here we examine the APIs event-based (SAX and StAX) and the tree based APIs (DOM, JDOM and DOM4J). They are discussed below.

2.3.1 SAX API [6, pp. 41-69][8]

One important XML API is Simple API for XML (SAX). It is public domain software. It was developed by members of the XML-DEV mailing list [9].

The API interacts with the program using a callback model or method which is a type of event based programming [6, pp. 41-69]. This means whenever any event like a tag or text in an XML file occurs while parsing, the code is called back. So the code can be written based on what we want to do depending on the particular event. For example suppose there is a tag ‘Name’ in your XML file and you want to retrieve the contents of the element ‘Name’. You can use a method
characters(). Figure 2.3 shows this process. In this process the java code is not working all the time. It is just working when an event occurs. Hence in this model, the code is passive. But we cannot go back to the part of the document which has been already parsed [6, pp. 41-69].

![Figure 2.3: The parsing process is controlled by the parser and code listens for events, responding as they occur [6, pp. 41-69]](image)

The SAX API is made up of a number of interfaces which define various methods. These APIs are defined in a package org.xml.sax. Figure 2.4 shows interfaces for the SAX parser. Following is the information for those interfaces.

- **SAXParserFactory** [10]: First a SAXParserFactory instance is to be created. This instance is nothing but the instance of a parser.
• **SAXParser** [10]: This interface defines the parse() methods. The XML document and default handlers are given to the parser which will help in parsing of the document. And different methods will be invoked by the handler object.

• **SAXReader** [10]: SAXReader is wrapped by SAXParser. The communication between SAX event handlers is carried out by SAXReader.

• **DefaultHandler** [10]: ContentHandler, ErrorHandler, DTDHandler and EntityResolver are all implemented by DefaultHandler.

• **ContentHandler** [10]: startDocument(), endDocument(), startElement(), endElement() and some other methods which give the text data are all defined in this interface. These methods are invoked when the XML Tags occur.

• **ErrorHandler** [10]: If any fatal errors or warnings occur, they are taken care of by this interface. In the case of a fatal error, an exception occurs, else errors and warnings are handled.
• **DTDHandler [10]**: In case of an unparsed entity during DTD processing, DTDHandler is used.

• **EntityResolver [10]**: If the parser is needed to get data from a uniform resource identifier (URI), an EntityResolver is used.

Unlike others APIs, SAX API cannot be used to create an XML document [6, pp. 41-69].
2.3.2 DOM API [6, pp. 91-124][11]

The **Document Object Model** (DOM) is developed by the World Wide Consortium (W3C). The important flexibility with this model is that it can be used with all programming languages and tools, i.e., DOM is a platform independent object framework. DOM is a standard specification exactly like XML [6, pp. 91-124][5].

The DOM API is organized into different levels which are similar to different versions in other APIs. DOM can also be used with other types of documents such as HTML or other content models [6, pp. 91-124].

For processing the XML documents DOM uses the tree model. This model basically creates in memory a representation of the document in the form of a tree. The DOM interface org.w3c.dom.node is the main interface on which everything in the tree is built [6, pp. 91-124]. The tree structure of a typical XML document is shown in Figure 2.5.
The tree model in Figure 2.5 shows that, for an Element node that has text value, the text value is the child node of the element node rather than being the element node itself. Hence if the text value is needed, getText() method can be used for the Element node. Traversing the tree can be done using getParent() and getChild() methods. So the node types need not be known [6, pp. 91-124].
DOM for reading an XML file

Figure 2.6 shows the interfaces in the DOM API and how they are related. The DocumentBuilderFactory interface will give a DocumentBuilder instance. The DocumentBuilder has a parse method, where the XML file is to be passed as an argument. The parse method will output a Document object which is nothing but the DOM tree. Using such methods as getParent(), getChild() traversing this tree is possible, and this can be useful in manipulating the XML file [10].

The two important packages for using DOM are org.w3c.dom and javax.xml.parsers. The first one defines DOM programming interfaces and the second one defines the DocumentBuilderFactory and DocumentBuilder classes [10].
DOM for creating an XML file

DOM API can also be used in creating an XML file. To create an XML file, first a DOM tree should be created. Two approaches can be used for this purpose. A new instance of org.w3c.dom.Document or org.w3c.dom.DOMImplementation class can be created. The package org.apache.xerces.dom contains both these classes. After any one of the above instances is created, a Document instance can be created using the createDocument() method. Then the root element for the XML document can be created. Child nodes can be formed and then connected to the root or other elements using the append() method. There are many other methods which can be used to create an attribute or set text values, etc., in the created XML file. Using these methods, XML files can be manipulated, which cannot be done in SAX [6, pp. 91-124].

DOM is a basic and important type of API when it comes to dealing with XML files. It can be used in reading, manipulating and also creating XML files. Thus studying DOM is very important when it comes to studying different XML APIs.

2.3.3 StAX API [6, pp. 196-242][12]

StAX stands for Streaming API for XML. This type of API uses the pull parsing approach for XML parsing. The authors Stefan Haustein and Aleksander Slominski of XMLPULL collaborated with BEA Systems, Oracle, Sun, Breeze Factor and James Clark to create StAX. StAX is a Java Specific Recommendation (JSR) 173 [13].
As noted, StAX is a Java "pull parsing" API. This API also acts like a "serial access" protocol, but its processing model is ideal for state dependent processing. With this API, you ask the parser to send you the next thing it has, and then decide what to do with what it gives you. For example, when you’re in a heading element and you get text, you’ll use one font size. But if you’re in a normal paragraph and you get text, you’ll use a different font size [10].

The DOM and SAX APIs described above are tree based or event based parsers. We have little or no control over parsing while using these APIs. The whole data stream is consumed at once using them. But in the StAX API, parsing can be started, continued, paused and resumed by the client which means the client has full control. Using StAX we can read and also write XML files [14].

**StAX for reading an XML file**

StAX consist of 2 APIs, cursor API and event iterator API. Reading and writing is done by the cursor API and the event iterator API is used in case of events. XMLStreamReader and XMLStreamWriter interfaces are used for cursor API whereas XMLEventReader and XMLEventWriter interfaces are for event iterator API. When a document is parsed using StAX, a set of events occurs, and these return an integer value [6].
Table 2.1: StAX API- Event Types [6, pp. 196-242]

Table 2.1 shows the event names and the corresponding event types. To read an XML file XMLStreamReader or XMLEventReader can be used. XMLStreamReader has a method hasNext() and next(). next() method returns a integer value which is nothing but the event code, as shown in Table 2.1 [6, pp. 196-242].
Using this information, an XML document can be read using StAX API.

**StAX for creating an XML file**

Unlike SAX, StAX can also be used in creating XML files. XMLStreamWriter interface can be used.

![XMLStreamWriter Interface](image)

Figure 2.7: XMLStreamWriter Interface [5]
Figure 2.7 shows the methods in the XMLStreamWriter interface that can be used to create an XML document.

Thus StAX which is a pull parsing API, can be used in reading and writing an XML file.

2.3.4. JDOM [15][6, pp. 243-287]

Java Document Object Model (JDOM) is developed for accessing an XML document in java by converting the XML file in the tree structure model [6, pp. 243-287]. In 2000, Jason Hunter along with Brett McLaughlin founded the JDOM project. The mission of the founders is “We want to provide a solution for using XML from Java that is as simple as Java itself [15]”. The main difference between JDOM and DOM is that, JDOM is specifically built for java while DOM can be used with various programming languages. JDOM is also an open source API [6, pp. 243-287].

Figure 2.8 shows the structure of the classes in the JDOM API. The Document class is the main class in JDOM. The document object carries the tree structure representation of the XML file. Element and Attribute classes are used for elements and attributes while Text and CDATA are for character data in the Element objects within the XML file [6, pp. 243-287].

One important thing about JDOM is, since it is developed specifically for java, java collections can be used while programming, unlike in DOM. For example, if getAttributes() method (which
returns the list of attributes for a particular element) is used we can directly use the java List for the result. So there is no need to look up new syntax or methods [6, pp. 243-287].

JDOM API is made up of concrete classes. The classes are called concrete because all of them can directly be used with the new keyword. This makes programming simpler [6, pp. 243-287].

![UML model of core JDOM classes](image)

Figure 2.8: UML model of core JDOM classes [6, pp. 243-287]
One thing that JDOM does not have is its own parser. It uses the default JAXP SAX parser that comes with java. But we can also use any other parser available [6, pp. 243-287].

**JDOM for reading an XML file**

First we have to create a SAXBuilder object using org.jdom.input.SAXBuilder interface. Then the build method of SAXBuilder is used to create a Document object of the XML file. If there is a problem reading the file, IOException will be thrown. Once a document is created, navigation through the document is possible using various methods of the classes such as Document class, Element class etc. [6, pp. 243-287].

**JDOM for creating an XML file**

JDOM can be used to create an XML file. To create an XML file, first a Document Object and Element object are created. The Element Object is created for the root element. Only the element’s name is required to create an Element object. Using setElementRoot(), the root element can be attached to the Document object. New elements can be added to an element using the addContent() method. Thus a whole tree can be created in the Document object. XMLOutputter class is used to output this Document object to a file [6, pp. 243-287].
2.3.5  DOM4J [6, pp. 288-309][16]

In the name DOM4J, J belongs to java which says this library was specifically built for java. It is distributed under Berkeley Software Distribution (BSD) style license. It is an open source library [16].

DOM4J is constructed around a set of core interfaces [6, pp. 288-309]. Figure 2.9 shows these interfaces.

![DOM4J Core Interfaces](image)

Figure 2.9: DOM4J core interfaces [6, pp. 288-309]

These interfaces together describe an XML document [6, pp. 288-309]. From Figure 2.9 it can be seen that Node interface is extended by all the other interfaces.
Figure 2.10 shows the methods contained in the Branch, Document and Element interfaces. These methods are useful in traversing through the Document object and its children [6, pp. 288-309].
DOM4J for reading an XML file

Like JDOM, DOM4J is not an XML parser. Hence it has to use some other parser to get the Document Object. Usually through org.dom4j.io.SAX, SAX parser is used. By passing the XML file to the read() method in SAXReader, a document object is created. There are several different ways to traverse a document and extract the information including Iterator, Lists and Index-Based Access [6, pp. 288-309].

DOM4J for creating an XML file

DOM4J can also be used to create XML documents. To create an XML file, first Document object is to be created. The Document object can be created with the help of the DocumentFactory class. Using the Element class, elements can be created and added to the Document object with add() method. Once the whole XML tree is created, the XML document is created from the Document object using the XMLWriter class [6, pp. 288-309].

2.3.6 XPath [6, pp. 154-195][17]

XQuery and XSLT Working Groups have together developed XPath. XPath’s main purpose is to address parts of an XML document. In addition to this, manipulation of strings, boolean variables and numbers can be done with the help of XPath. Also XPath helps in pattern matching of the nodes. It uses syntax other than XML syntax to find out the URIs and XML attribute values. The name XPath is derived from the fact that it uses path notation for navigating the tree
structure of the XML document. In XPath, XML is modeled as a tree of nodes, such as element nodes, text nodes, attribute nodes etc. The String value of each of these types of nodes can be computed by XPath [17].

A rich XPath API has been added to JAXP [6, pp. 154-195]. Figure 2.11 shows the JAXP XPath interfaces.

![Figure 2.11: JAXP XPath Interfaces [6, pp. 154-195]](image)

javax.xml.xpath.XPath is the core XPath interface. XPath is not a parser. It uses other parsers to parse the XML documents into a Document object. And then it uses this Document object. There are various evaluate methods in this interface which help in calculating the XPath expression from the document object which has been parsed by other parsers [6, pp. 154-195].
To use the XPath interface, first we need to create an instance of XPathFactory using the newInstance() method. If the DOM parser is being used to create the Document object, no arguments need be passed to the newInstance() method. If JDOM or any other parser objects are being used, the URI belonging to that parser should be passed in the newInstance method. The correct implementation of XPathFactory is found by this newInstance() method. After this, an instance of XPath can be created using the newXPath() method. This XPath instance then can be used to implement an evaluate method which helps in extracting the desired information from the XML file [6, pp. 41-69]. Thus we can read data from the XML document using XPath.

Apart from the APIs discussed, there are some other APIs available in the market which can be used for these purposes. But since the ones discussed above are more popular, they have been selected for this thesis.

2.4 What work related to comparison of the APIs has already been done in past?

Some work has been done related to comparison between the APIs in the past. Some papers that have been published work related to this thesis are given below with a short summary.

In [1], Haw and Rao focus on a comparative study of parsers. An API is used through which an XML document can be read by the parser. The authors have compared the performances of streaming based parsers and tree based parsers. They have also compared the parsing times of
DOM and SAX based parsers like Xerces Java, .NET and xParse for different datasets. They have also implemented a non-validating SAX parser named xParse on a java platform.

<table>
<thead>
<tr>
<th>APIs</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>-Easy navigation</td>
<td>-XML document must be parsed at one time</td>
</tr>
<tr>
<td></td>
<td>-Entire tree loaded into memory</td>
<td>-It is expensive to load entire tree into memory</td>
</tr>
<tr>
<td></td>
<td>-Random access to XML document</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-Rich set of APIs</td>
<td></td>
</tr>
<tr>
<td>SAX</td>
<td>-Entire document not loaded into memory which resulting in low memory</td>
<td>-No built in document navigation support</td>
</tr>
<tr>
<td></td>
<td>consumption</td>
<td>-No random access to XML document</td>
</tr>
<tr>
<td></td>
<td>-Allows registration of multiple ContentHandlers</td>
<td>-No support for modifying XML in place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-No support for namespace scoping</td>
</tr>
<tr>
<td>StAX</td>
<td>-Contains two parsing models, for ease or performance</td>
<td>-No built-in document navigation support</td>
</tr>
<tr>
<td></td>
<td>-Application controls parsing, easily supporting multiple inputs</td>
<td>-No random access to XML Document</td>
</tr>
<tr>
<td></td>
<td>-Powerful filtering capabilities provide efficient data retrieval</td>
<td>-No support for modifying XML in place and is still in an immature state</td>
</tr>
<tr>
<td>Electric XML</td>
<td>-Light Weighted</td>
<td>-No support for validation</td>
</tr>
<tr>
<td></td>
<td>-Fast in performance</td>
<td>-Still in immature state</td>
</tr>
</tbody>
</table>

Table 2.2: Comparison of XML parser APIs [1]
Table 2.2 shows the comparison of the different parser APIs. DOM, SAX, StAX and Electric XML are the APIs compared based their features. Among these DOM, SAX and StAX, along with few other APIs, are considered here.

It is stated in [1] that the best parser in terms of performance is Xerces Java. When the file size increases, xParse proves more efficient.

In [19], Karre and Sebastian have compared five different parsers. The main aspects on which this comparison is done are speed, accuracy and storage space. A group of metrics and scenarios is developed to examine these parsers. A test suite developed by OASIS is used as the benchmark. The parsers they used are Apache, IBM, Sun, SXP and Aelfred. The results of the study show that Apache, IBM and Xerces perform almost the same when the accuracy aspect is considered. If speed is to be considered, Aelfred is the fastest parser.

In [20], Lam, Ding and Liu provide a survey on four parsing models, SAX, DOM, StAX and VDT. The comparison is based on operational and performance characteristics. In this paper, the whole process of parsing including character conversion, lexical analysis and syntactic analysis is taken into consideration. Table 2.3 shows XML processing performance characteristics.

The analysis done by the authors shows SAX and StAX perform really well when there is less memory available and there is no need to go back to the already parsed part of the document. In short DOM works best for database applications, SAX and StAX for streaming applications and VTD for hardware acceleration.
Apart from the papers above there are few other papers which have compared XML parsers. The main difference between the previous work and our work is that we have compared XML parsing APIs and not XML parsers.

Table 2.3: XML processing performance characteristics [20]
3. METHODOLOGY

Today many organizations use XML files to store their data. The data from these files is frequently needed for certain operations. In some cases all the data from the files is needed and in some cases only parts of the data from an XML file is needed. There are many APIs available in markets which help in reading the XML files. For XML beginners it becomes a difficult task in choosing an API. Here we will compare different APIs based on some factors which will help in choosing an appropriate API for an application.

To decide the criteria for comparing the APIs is in itself a task. There are various questions that need be answered to determine these criteria:

- Which programming platform should be used?
- Which APIs should be considered?
- What experiments should be performed?
- What other factors should be considered, other than the experiments, to compare the APIs?
- What work related to comparison of the APIs has already been done in the past?

This chapter primarily answers all the above questions. It also specifies the experimental set up and describes the XML files that were used for experimentation.
3.1 Which programming platform should be used?

The format of XML is such that it can be used across any platform. It can be said that XML has a cross platform data format. Also Java is a standard cross platform programming language. For these reasons, there are various Java APIs available which are developed for XML access and manipulation [21].

Java technology gives an extensive productivity improvement for software developers compared to programming languages such as C or C++ [21]. Software developers who use Java do generate sophisticated programs that are more reusable and more maintainable than programs written in scripting languages. Together, XML and Java help in building sophisticated, interoperable applications at a lower cost and in less time.

There are many Java APIs available free of cost that are suitable for this purpose. The Java JDK contains built in XML parsers. Hence we have chosen java as a programming platform for this thesis.

3.2 Which APIs should be considered?

Choosing the interfaces is one of the important questions for the thesis. Java is the development platform for the experiments, as explained in the section 3.1. Hence while choosing an API, the basic qualification is that it should be java compliant.
The parser APIs can be basically divided into two types:

- Event based API
- Tree based API

The details of the above types have been discussed in section 2.3. To get a good range of results, it will be useful to select at least one API of each type listed above.

- The most important and common event based parser API for XML is the **SAX** API. It is a basic streaming type of API. It can be used with Java. Also it is a push type of API. SAX can be used for reading XML files but not writing or modifying it. Hence SAX will be included because it is the most common type of event-based API.

- The most important and common tree based parser API for XML is the **DOM** API. DOM is a basic type of tree based API. Without studying and understanding DOM, it is difficult to understand other tree based APIs. Also DOM can be used with Java. DOM is also a push parsing type of API. DOM can both read and write the XML files. DOM will be included because it is the most common type of tree based API.

- **StAX**, is a pull parsing API unlike SAX or DOM. But it is a streaming API like SAX. It can be used for reading and writing an XML file. This is the only pull parsing type API considered here.
• Since java and XML are both cross platform, there are a few APIs that are specifically built for java. These APIs use Java interfaces too. This makes programming simpler for a java programmer who is new to XML file parsing. **DOM4J** is an API which was specifically developed for java. It is a tree based API like DOM, and also derived from the idea of the DOM parser. Thus it will be included here.

• Like DOM4J, JDOM is also developed for the java platform specifically. It is also a tree based API.

• There are query languages for databases, similarly, there is a query based XPath API. We can get results by just firing a query. This is important when we need small amount of data from an XML file. XPath uses the tree structure to get the results. Since it is different from the above types, knowing the performance of this API will be helpful.

Table 3.1 shows APIs and the corresponding type.

<table>
<thead>
<tr>
<th>API</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAX, StAX</td>
<td>Event-based API</td>
</tr>
<tr>
<td>DOM, JDOM, DOM4J</td>
<td>Tree based APIs</td>
</tr>
<tr>
<td>XPath</td>
<td>Query Language for XML</td>
</tr>
</tbody>
</table>

Table 3.1: APIs and types
3.3 Which experiments should be performed?

To perform the experiments, the important aspects about XML files should be taken into consideration. These include:

- Find the XML file sizes to be considered.
- Find the important elements in the XML file that are extracted often.
- Find the XML files used as input to the programs.
- Use the Cummins Inc. Project XML files as input.

3.3.1 Find the size of XML documents to be considered

When parsing any XML document, the size of file can matter a lot.

For the event type APIs are SAX, StAX, the size of XML document usually doesn’t matter, since the parsing is done by streaming the XML data to the parser.

On the other hand, for tree based APIs, the tree representation of the XML document is created in the memory. Hence the bigger the XML file, the bigger the tree document that is formed. This can become a problem when the files are really very large.

For these reasons, the size of an XML file plays an important role in choosing any kind of API. A file of size 5 KB can be considered a small file, while a file of size 3 MB can contain a huge
amount of data. Therefore for the experiments we will be using 3 files with different sizes of
**5KB, 1MB and 3MB.** This will give a good range for comparing the processing time results.

### 3.3.2 Find the important elements in an XML file that are extracted often

XML files are basically intended for web pages, reference manuals, tutorials, books, poems, short stories, etc. XML was created as a document format. It is also used in various computer applications for data storage, order processing, database exchange and backup. This implies that XML files can be narrative-like documents or record-like documents [5].

In both the formats, the documents store data. The XML documents are nothing but tree structures with the root element of the document being the tree root. Each and every document can be shown as a tree. These trees are general rooted trees without any restriction on the order of nodes or the number of children a node can have [5].

The main information in these trees is the text that is stored in the document. This emphasizes the importance of the element values and the attribute values, as they form the largest part of text that is available in the document [5].

Hence in these experiments, we will concentrate on extracting the element values and attribute values from the XML file.
3.3.3 Find the XML files used as an input to the programs

As discussed in section 3.3.1, the files with sizes 5KB, 1MB and 3MB will be used for the experiments. To create a file of size 5KB is an easy task, but to create a 1MB or 3MB file, a lot of data is required. Therefore we decided to use the files available on the Sun’s official website, which contains a good collection of large XML files.

Sun’s website, http://java.sun.com/developer/earlyAccess/xml/examples/index.html has many sample XML files. A file named nt.xml is 1MB and contains the complete New Testament. Another file ot.xml is a 3MB file and has all the data form the Old Testament. For the experiments we will be using both these files as input to our programs.

The most important tags in both, nt.xml and ot.xml, are:

- `<bktlong>` contains the complete title of the book
- `<bktshort>` contains the short title of the book
- `<chtitle>` contains the chapter title
- `<v>` contains the contents of the chapter.

These tags make up most of the data in the files. Data from all of these tags is extracted in the experiments.
3.3.4 Use the Cummins Inc. Project XML files as input

In the above sections of this chapter one have discussed the file size of the XML files that should be considered. But in many cases not only the file size matters. The number of XML files also matters. In many industrial applications, the number of files that need to be parsed for a data extraction is large.

In the summer of 2011, I worked as a software engineering intern at Cummins Inc., Columbus, IN. During my work there I had to deal with XML files extensively. One of the projects I was working on had the main purpose of extracting information from a number of XML files. Thereafter I had to generate a report of all the extracted data which was a text file. These XML files that were used contained data related to the electronic control module in an engine.

During the early stages of this project I did not have much idea about different types of APIs. With a very small amount of research, because of the shortage of time, I decided to use the SAX API.

The results of this thesis may point at using some API rather than SAX. To run the experiments, in one of the cases, I have used the 59 XML files from the Cummins Inc. Project. Because of proprietary information, the file contents are not included here. But comparing processing time for extracting the same information, using all the 6 APIs is considered as one experimental case.
3.4 What other factors should be considered other than experiments to compare the APIs?

Whenever it comes to using an API, the difficulty of the level of programming for that API becomes an important factor if there are time constraints. Hence how difficult it is to learn and use a particular API in programming sense becomes an important factor in choosing an API. Thus, we also compared the use of APIs based on their programming ease.

3.5 Experimental setup

For running the tests, two systems were used. System specifications are given below.

- **System 1**
  
  Operating System: Windows 7 Enterprise
  
  Processor Name: Intel(R) Core(TM)i5 CPU M 520@2.40GHz
  
  Processor Speed: 2.40GHz
  
  Memory: 3 GB
  
  System Type: 32-bit Operating System
• **System 2**

  Operating System: Mac OS X 10.7.3

  Processor Name: Intel Core i5

  Processor Speed: 2.30 GHz

  Memory: 4 GB

  System Type: 64-bit Operating System

Software used:

• **Integrated Development Environment:**

  Eclipse IDE for Java Developers

  Version: Indigo Service Release 1

• **Programming Language: Java**

  Java Product: Java™ Platform SE 6 U31

  Java Version: 6.0.310.5
4. RESULTS

This chapter contains the results of the experimental cases discussed in chapter 3. The list of the cases is given below.

A. Comparing performance. We chose the following cases to study the range of commonly occurring sizes.

1. Reading an XML file of size 5 KB.
2. Reading an XML file of size 1MB.
3. Reading an XML file of size 3MB
4. Reading 59 XML files (Cummins Inc. Project).

B. Comparing the performance of creating a small XML file.

C. Comparing programming ease.

Each of the experiments in A and B was executed 10 times. All of them were executed on 2 different systems. Details of the system configurations are given in section 3.6.

Table 4.1 shows the list of APIs for the corresponding experimental cases.
Table 4.1: List of experimental cases and the corresponding APIs used.

<table>
<thead>
<tr>
<th>API</th>
<th>Cases A1, A2, A3</th>
<th>Case A4</th>
<th>Case B</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAX [19]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>DOM [21]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>StAX [22]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DOM4J [23]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JDOM [24]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>XPath [25]</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.1 above shows XPath was not used for cases A1, A2 and A3 because as already stated XPath is basically a query language. Hence it is mostly used when some parts of data are required from the XML document. But in cases A1, A2 and A3, most of the data will be retrieved. Hence XPath is not used. For case B, SAX is not used because it cannot create XML documents.

### 4.1 Experiments in case A and B performed on Windows 7 system

Section 4.1 consists of result tables for each API. Each table shows the minimum, maximum, average performance time (msec) and standard deviation for all the experiments performed. This section contains results for the Windows 7 operating system.
4.1.1 Case A1

Reading an XML file of size 5 kb.

System: Windows 7

Table 4.2 shows the list of APIs and the corresponding reading times:

<table>
<thead>
<tr>
<th>API</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAX</td>
<td>61</td>
<td>63.1</td>
<td>67</td>
<td>2.18</td>
</tr>
<tr>
<td>DOM</td>
<td>72</td>
<td>74.2</td>
<td>80</td>
<td>2.57</td>
</tr>
<tr>
<td>StAX</td>
<td>43</td>
<td>44.6</td>
<td>47</td>
<td>1.26</td>
</tr>
<tr>
<td>DOM4J</td>
<td>148</td>
<td>150.1</td>
<td>154</td>
<td>2.77</td>
</tr>
<tr>
<td>JDOM</td>
<td>129</td>
<td>133.2</td>
<td>136</td>
<td>2.78</td>
</tr>
</tbody>
</table>

Table 4.2: Case A1 for Windows 7 system
Figure 4.1 shows the different APIs on the X-axis and the processing time for reading a file of size 5KB on the Y-axis.

For reading a small XML file of size 5KB, the results above show that StAX API performs fastest with the average processing time of 44.6 msec. It is then followed by SAX parser and then by DOM parser. It can be seen that the tree based parsers JDOM and DOM4J take maximum time compared to all other APIs to read small XML files, above 100 msec, which is more than double the amount of time taken by StAX API. Note that standard deviations are small in all cases.
4.1.2 Case A2

Reading an XML file of size 1MB.

System: Windows 7 system

Table 4.3 shows the list of parser APIs and the corresponding reading times.

<table>
<thead>
<tr>
<th>API</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAX</td>
<td>861</td>
<td>877.8</td>
<td>894</td>
<td>9.69</td>
</tr>
<tr>
<td>DOM</td>
<td>1137</td>
<td>1161.2</td>
<td>1178</td>
<td>11.59</td>
</tr>
<tr>
<td>StAX</td>
<td>862</td>
<td>879.2</td>
<td>894</td>
<td>11.57</td>
</tr>
<tr>
<td>DOM4J</td>
<td>1150</td>
<td>1173</td>
<td>1193</td>
<td>14.40</td>
</tr>
<tr>
<td>JDOM</td>
<td>1153</td>
<td>1169.6</td>
<td>1184</td>
<td>10.71</td>
</tr>
</tbody>
</table>

Table 4.3: Case A2 for Windows 7 system
Figure 4.2 shows the different APIs on the X-axis and processing time for reading a file of size 1MB on the Y-axis.

In this case file size has increased here as compared to case A1, and 2 APIs, SAX and StAX have best performance time. The tree based APIs seem to have almost the same processing times. Comparing this case to case A1, the difference between the lowest and the highest performance time has reduced. Also time taken by DOM was considerably less than DOM4J and JDOM in case A1, but here it is almost the same.

As file size has increased, standard deviation also increased to maximum of 12 msec.
4.1.3 Case A3

Reading an XML file of size 3MB

System: Windows 7 system

Table 4.4 shows the list of parser APIs and the corresponding reading times.

<table>
<thead>
<tr>
<th>API</th>
<th>Execution Time (msec)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
<td>Average</td>
<td>Maximum</td>
<td>Standard Deviation</td>
</tr>
<tr>
<td>SAX</td>
<td>3889</td>
<td>3957.5</td>
<td>4087</td>
<td>69.37</td>
</tr>
<tr>
<td>DOM</td>
<td>4112</td>
<td>4252.5</td>
<td>4311</td>
<td>60.49</td>
</tr>
<tr>
<td>StAX</td>
<td>3811</td>
<td>3872.9</td>
<td>3951</td>
<td>45.56</td>
</tr>
<tr>
<td>DOM4J</td>
<td>4321</td>
<td>4382.7</td>
<td>4440</td>
<td>43.21</td>
</tr>
<tr>
<td>JDOM</td>
<td>4405</td>
<td>4472.2</td>
<td>4529</td>
<td>51.75</td>
</tr>
</tbody>
</table>

Table 4.4: Case A3 for Widows 7 system
Figure 4.3 shows the different APIs on the X-axis and the processing time for reading a file of size 3MB on the Y-axis.

StAX API still performs best followed by SAX API. The difference between the highest and the lowest processing time has still reduced. For 3MB file, streaming-based parsers have almost the same average time, and the same is true for tree-based parsers.

For this case, standard deviation has increased much more and ranges from 43 to 70 msec.
4.1.4 Graph plotted for average execution time against file size for all the APIs

Figure 4.4 shows the chart plotted for average execution time against file size for all the APIs combining cases A1, A2 and A3 for Windows 7 system.

For all the file sizes, StAX API always takes the minimum amount of execution time. It is then followed by SAX API. All tree based APIs take more time than streaming APIs.

Figure 4.4: Chart plotted for average execution time against file size for all the APIs combining cases A1, A2 and A3 for Windows 7 system
4.1.5 Case A4:

Reading 59 XML files (Cummins Inc. Project)

System: Widows 7 system

Table 4.5 shows the list of parser APIs and the corresponding reading times.

<table>
<thead>
<tr>
<th>API</th>
<th>Execution Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
</tr>
<tr>
<td>SAX</td>
<td>291</td>
</tr>
<tr>
<td>DOM</td>
<td>391</td>
</tr>
<tr>
<td>StAX</td>
<td>210</td>
</tr>
<tr>
<td>DOM4J</td>
<td>451</td>
</tr>
<tr>
<td>JDOM</td>
<td>441</td>
</tr>
<tr>
<td>XPath</td>
<td>2012</td>
</tr>
</tbody>
</table>

Table 4.5: Case A4 for Widows 7 system
Figure 4.5 shows the different APIs on the X-axis and the processing time for reading a 59 XML files on the Y-axis.

XPath API requires maximum time. StAX API programs execute in minimum time. Average time taken by XPath is almost 10 times the time taken by StAX. StAX is then followed by SAX.
4.1.6 Case B

Creating an XML file of size 1KB.

System: Widows 7 system

Table 4.6 shows the list of parser APIs and the corresponding creation times.

<table>
<thead>
<tr>
<th>API</th>
<th>Execution Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
</tr>
<tr>
<td>DOM</td>
<td>53</td>
</tr>
<tr>
<td>StAX</td>
<td>19</td>
</tr>
<tr>
<td>DOM4J</td>
<td>50</td>
</tr>
<tr>
<td>JDOM</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 4.6: Case B for Widows 7 system
Figure 4.6 shows average time for creating an XML file for different APIs.

StAX API takes minimum time of 19.6 msec. DOM takes maximum time of 54.4 msec. Standard deviation lies between 0.5 and 2.5 msec.

4.2 Experiments in case A and B performed on Mac OS system:

Section 4.2 consists of result tables for each API. The tables show the minimum, maximum and average performance time (msec) and standard deviation for all the experiments performed. This section contains results for Mac OS system.
4.2.1 Case A1

Reading an XML file of size 5KB.

System: Mac OS system

Table 4.7 shows the list of APIs and the corresponding reading times.

<table>
<thead>
<tr>
<th>API</th>
<th>Execution Time (msec)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
<td>Average</td>
<td>Maximum</td>
<td>Standard Deviation</td>
</tr>
<tr>
<td>SAX</td>
<td>36</td>
<td>38.1</td>
<td>41</td>
<td>1.44</td>
</tr>
<tr>
<td>DOM</td>
<td>45</td>
<td>46.2</td>
<td>48</td>
<td>0.91</td>
</tr>
<tr>
<td>StAX</td>
<td>27</td>
<td>29.2</td>
<td>31</td>
<td>1.47</td>
</tr>
<tr>
<td>DOM4J</td>
<td>81</td>
<td>84.5</td>
<td>87</td>
<td>1.95</td>
</tr>
<tr>
<td>JDOM</td>
<td>73</td>
<td>76.3</td>
<td>78</td>
<td>1.33</td>
</tr>
</tbody>
</table>

Table 4.7: Case A1 for Mac OS system
Figure 4.7 shows the different APIs on the X-axis and the processing time for reading a file of size 5KB on the Y-axis.

For reading a small XML file of size 5KB, the results above show that StAX API performs best with the average processing time of 29.2 msec. It is then followed by SAX parser and then by DOM parser. It can be seen that tree based parsers JDOM and DOM4J take maximum time to read small XML files, above 70 msec.

The standard deviation is very small and varies between 0.5 and 2 msec for all APIs.
4.2.2 Case A2

Reading an XML file of size 1MB.

System: Mac OS system

Table 4.8 shows the list of parser APIs and the corresponding reading times.

<table>
<thead>
<tr>
<th>API</th>
<th>Execution Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
</tr>
<tr>
<td>SAX</td>
<td>524</td>
</tr>
<tr>
<td>DOM</td>
<td>734</td>
</tr>
<tr>
<td>StAX</td>
<td>425</td>
</tr>
<tr>
<td>DOM4J</td>
<td>743</td>
</tr>
<tr>
<td>JDOM</td>
<td>784</td>
</tr>
</tbody>
</table>

Table 4.8: Case A2 for Mac OS system
Figure 4.8: Case A2 for Mac OS system

Figure 4.8 shows the different APIs on the X-axis and the processing time for reading a file of size 1MB on Y-axis.

As in case A2 again StAX performs best with lowest execution time of 433.3 msec. SAX is the second best performer with the time of 544 msec. As the file size increases, DOM, JDOM and DOM4J have almost the same execution times.

As the file size increases, the standard deviation also increases to a maximum of 12 msec for JDOM and a minimum of 4.49 msec for StAX. This also shows StAX is more consistent than other APIs.
4.2.3 Case A3

Reading an XML file of size 3MB

System: Mac OS system

Table 4.9 shows the list of parser APIs and the corresponding reading times.

<table>
<thead>
<tr>
<th>API</th>
<th>Execution Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
</tr>
<tr>
<td>SAX</td>
<td>1630</td>
</tr>
<tr>
<td>DOM</td>
<td>1869</td>
</tr>
<tr>
<td>StAX</td>
<td>1586</td>
</tr>
<tr>
<td>DOM4J</td>
<td>2035</td>
</tr>
<tr>
<td>JDOM</td>
<td>1928</td>
</tr>
</tbody>
</table>

Table 4.9: Case A3 for Mac OS system
Figure 4.9: Case A3 for Mac OS system

Figure 4.9 shows the different APIs on the X-axis and the processing time for reading a file of size 3MB on Y-axis.

For file size 3MB, StAX still performs best with execution time of 1681.8 msec. As in previous cases, StAX is followed by SAX, then DOM and at the end JDOM and DOM4J. But there is very little difference in the execution times of DOM, DOM4J and JDOM. So it can be said that tree based parsers perform almost the same with increasing file sizes.

For file size 3MB, standard deviation increased a lot to a maximum of 138 msec as compared to 2 msec for a file of size 5KB. From the results it can be seen that the tree based APIs have maximum deviation.
4.2.4 Graph plotted for average execution time against file size for all the APIs

Figure 4.10: Chart plotted for average execution time against file size for all the APIs combining cases A1, A2 and A3 for Mac OS system

Figure 4.10 shows average execution time for all the file sizes in cases A1, A2 and A3 for all APIs.

It can be clearly seen that StAX performs best in all the cases. SAX performs best after StAX. For smaller files, DOM performs better than JDOM and DOM4J. But as the file size increases, DOM, JDOM and DOM4J take almost equal execution time.
4.2.5 Case A4

Reading 59 XML files (Cummins Inc. Project)

System: Mac OS system

Table 4.10 shows the list of parser APIs and the corresponding reading times.

<table>
<thead>
<tr>
<th>API</th>
<th>Execution Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
</tr>
<tr>
<td>SAX</td>
<td>264</td>
</tr>
<tr>
<td>DOM</td>
<td>417</td>
</tr>
<tr>
<td>StAX</td>
<td>218</td>
</tr>
<tr>
<td>DOM4J</td>
<td>408</td>
</tr>
<tr>
<td>JDOM</td>
<td>451</td>
</tr>
<tr>
<td>XPath</td>
<td>1586</td>
</tr>
</tbody>
</table>

Table 4.10: Case A4 for Mac OS system
Chart 4.11 shows the different APIs on the X-axis and the processing time for reading 59 XML files on Y-axis.

StAX API execution time of 223.6 msec is lowest for reading the 59 files. SAX is a little bit more than StAX. DOM, DOM4J and JDOM require almost the same execution times. XPath on the other hand requires maximum time of 1595.2 msec which is almost 7 times the time required for StAX API.

Standard deviation ranges from 3 msec to 7 msec. The lowest standard deviation is for StAX API.
4.2.5 Case B

Creating an XML file of size 1KB.

System: System 2

Table 4.11 shows the list of parser APIs and the corresponding file creation times:

<table>
<thead>
<tr>
<th>API</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>56</td>
<td>59</td>
<td>62</td>
<td>1.88</td>
</tr>
<tr>
<td>StAX</td>
<td>13</td>
<td>14.4</td>
<td>16</td>
<td>1.07</td>
</tr>
<tr>
<td>DOM4J</td>
<td>36</td>
<td>37.9</td>
<td>40</td>
<td>1.28</td>
</tr>
<tr>
<td>JDOM</td>
<td>19</td>
<td>20.3</td>
<td>21</td>
<td>1.05</td>
</tr>
</tbody>
</table>

Table 4.11: Case B for Mac OS system
Figure 4.12: Case B for Mac OS system

Figure 4.12 shows the different APIs on the X-axis and the processing time for creating an XML file on Y-axis.

StAX API takes minimum time of 14.4 msec to write an XML file of size around 250 KB. DOM API takes maximum time of 59 msec to write the same file. This time is almost 4 times the time taken by StAX.

Standard deviation is very small in the range of 1 to 2 msec.
4.3 Comparing the performance of two systems

In this section, we will be comparing the performance of two the systems which have been used to run the experiments. Comparison is done between the processing times of the sum of average execution time of all APIs for a case in Windows 7 with the equivalent measure for Mac OS.

Table 4.11 shows the list of cases and the corresponding sums of average execution times for both the systems:

<table>
<thead>
<tr>
<th>Cases</th>
<th>Sum of average execution time for system 1 (msec) (Windows 7)</th>
<th>Sum of average execution time for system 2 (msec) (Mac OS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case A1</td>
<td>465.2</td>
<td>274.3</td>
</tr>
<tr>
<td>Case A2</td>
<td>5260.8</td>
<td>3290</td>
</tr>
<tr>
<td>Case A3</td>
<td>20937.8</td>
<td>9695.3</td>
</tr>
<tr>
<td>Case A4</td>
<td>3860.9</td>
<td>3386.5</td>
</tr>
<tr>
<td>Case B</td>
<td>158.2</td>
<td>131.6</td>
</tr>
</tbody>
</table>

Table 4.12 Comparison of system 1 vs. system 2.
From the values in Table 4.11 it can be seen that the sum of times in Mac OS is always less than that in Windows 7. This may be because of one big difference between the systems, which is that Windows 7 is 32-bit while Mac OS is 64-bit.

4.4 Comparison based on programming ease and understanding the API

Whenever it comes to using an API, the difficulty of programming for that API becomes an important factor if there are time constraints for the project. Hence we are interested in exploring, how difficult it was to learn and use a particular API in the programming sense.

Figure 4.13 shows a sample XML file. Code for extraction of tags Name, Degree and Department for the sample XML file for all APIs was written. A comparison of coding styles style for all the APIs is done based on this code.

```xml
<Information>
<Name>Neha Gujarathi</Name>
<Department>Electrical and Computer Engineering</Department>
<Degree>M.S. Computer Engineering</Degree>
</Information>
```

Figure 4.13: Sample XML document

Figure 4.14 contains the code written for the SAX API. To write code for the SAX API, first a good understanding of how the API works is necessary. The programming style is a bit complicated for beginners. Understanding of how the handler works and which methods to use
when is important here. The number of lines of code is more for SAX than for other APIs. But once we understand the working of the API it becomes easy to write the code.

```java
public class SAX {
    public static void parseUsingSAX(String fileName) throws ParserConfigurationException, SAXException, IOException {
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser saxParser = factory.newSAXParser();
        DefaultHandler handler = new DefaultHandler()
        {
            boolean flagName, flagDegree, flagDepartment;
            public void startElement(String uri, String localName, String qName, Attributes attributes) throws SAXException {
                if (qName.equalsIgnoreCase("Name"))
                    flagName = true;
                if (qName.equalsIgnoreCase("Degree"))
                    flagDegree = true;
                if (qName.equalsIgnoreCase("Department"))
                    flagDepartment = true;
            }
            public void characters(char ch[], int start, int length) throws SAXException {
                if (flagName) {
                    String author = new String(ch, start, length);
                    System.out.println("Name: "+author);
                    flagName = false;
                }
                if (flagDegree) {
                    String title = new String(ch, start, length);
                    System.out.println("Degree: "+title);
                    flagDegree = false;
                }
                if (flagDepartment) {
                    String genre = new String(ch, start, length);
                    System.out.println("Department: "+genre);
                    flagDepartment = false;
                }
            }
        );
        saxParser.parse(fileName, handler);
    }
```

Figure 4.14: SAX API example code
Figure 4.15 shows the DOM example code. If any tag name is known and its value is needed from the XML document, it can be easily retrieved using `getElementsByTagName()` method. In such cases, where a small amount of data is needed, DOM is very easy to use. But it gets complex when most of the data from the file is needed in the same sequence as it can be read from the XML file. As files become bigger, the depth of the document tree increases and hence the “for loops” while programming become more complex. DOM will be useful for record like documents rather than narrative documents and where data in sequence is not needed. It will also be helpful when most of the data from the file is not needed.
public class StAX {
    public static void parseUsingStAX(String fileName) throws
    FileNotFoundException, XMLStreamException, FactoryConfigurationException {
        FileInputStream fileInputStream = new FileInputStream(fileName);
        boolean flagName = false, flagDegree = false, flagDepartment = false;
        XMLStreamReader xmlStreamReader = XMLInputFactory.newInstance().createXMLStreamReader(fileInputStream);
        while (xmlStreamReader.hasNext()) {
            int eventCode = xmlStreamReader.next();
            switch (eventCode) {
                case 1:
                    if (xmlStreamReader.getLocalName().equals("Name"))
                        flagName = true;
                    if (xmlStreamReader.getLocalName().equals("Degree"))
                        flagDegree = true;
                    if (xmlStreamReader.getLocalName().equals("Department"))
                        flagDepartment = true;
                    break;
                case 4:
                    if (flagName) {
                        System.out.println("Name:
" + xmlStreamReader.getText());
                        flagName = false;
                    }
                    if (flagDegree) {
                        System.out.println("Degree:
" + xmlStreamReader.getText());
                        flagDegree = false;
                    } if (flagDepartment) {
                        System.out.println("Department:
" + xmlStreamReader.getText());
                        flagDepartment = false;
                    } break;
            }
        }
    }
}

Figure 4.16: StAX API example code

Figure 4.16 shows the StAX API code. Like SAX it is a streaming based parser. Hence coding is almost similar to SAX except that in StAX, XMLStreamReader will return an integer value for
events like start element, characters etc. Using these integer values we can perform whatever action we need depending on the case. The number of lines of code is larger, as it is in SAX.

```java
public class DOM4J {
    public static void parseUsingDOM4J(String fileName) throws DocumentException{
        File xml = new File(fileName);
        SAXReader reader = new SAXReader();
        Document doc = reader.read(xml);
        Element root = doc.getRootElement();
        for(Iterator i = root.elementIterator(); i.hasNext();)
        {
            Element el=(Element)i.next();
            if(el.getName()=="Name")
                System.out.println("Name: "+el.getText());
            if(el.getName()=="Degree")
                System.out.println("Degree: "+el.getText());
            if(el.getName()=="Department")
                System.out.println("Department: "+el.getText());
        }
    }
}
```

Figure 4.17: DOM4J API example code

Figure 4.17 shows DOM4J API code. While using DOM4J, a Document object which contains a tree representation of the XML document is created as in DOM. Therefore traversing the tree will help in accessing data. Since DOM4J was built specifically for Java, Java utilities can be used in DOM4J. The number of lines of code is smaller. Also coding is simple. But it can get difficult and complex if the XML document is a narrative document and the depth of the tree increases and all of the data is to be extracted in sequence.
A code sample for the JDOM API is shown in Figure 4.18. JDOM is also a tree based API. Hence retrieving the information can be done by traversing the tree structure in the Document object. Some functions can help in directly going to a part of the tree just by giving the element name. Hence dealing with small files or trees with small depth becomes very easy compared to SAX or StAX. But as in DOM or DOM4J, if the document is a narrative document, and all of the data is to be retrieved in sequence, coding becomes difficult. The number of lines of code is smaller than in SAX or StAX.

```java
public class JDOM {
    public static void parseUsingJDOM(String fileName) throws JDOMException, IOException {
        SAXBuilder builder = new SAXBuilder();
        Document doc = builder.build(fileName);
        Element root = doc.getRootElement();
        List list = root.getChildren();
        for (int i = 0; i < list.size(); i++) {
            Element element = (Element) list.get(i);
            if (element.getName().equals("Name"))
                System.out.println("Name: "+element.getText());
            if (element.getName().equals("Degree"))
                System.out.println("Degree: "+element.getText());
            if (element.getName().equals("Department"))
                System.out.println("Department: "+element.getText());
        }
    }
}
```

Figure 4.18: JDOM API example code
XPath API example code is shown in Figure 4.19. XPath code is a set of queries to fire. Hence it is very easy to write once we understand the format and syntax of queries. If the amount of data to be retrieved from the XML document is small, XPath queries can be helpful. But the downside with using XPath is that it takes the maximum amount of time to get data.

Table 4.13 contains the list of APIs and their corresponding suitability for different kinds of files in terms of coding.
4.5 Summary of experimental results

To summarize the results in this chapter, StAX API performs best in all the cases (A1, A2, A3, A4 and B) for both systems (Windows 7 and Mac OS) with respect to execution time. The second best performance is given by SAX API. One important observation is that, as the size of the file goes on increasing, the relative difference between execution time for all APIs goes on decreasing. Also the standard deviation increases as the file size increases. Overall results
indicate that the event based parsers perform better than the tree based ones. However, when it comes to programming complexity, SAX is most complex, followed by StAX, for beginners. Coding for tree based parsers is almost the same as traversing the tree structure. But for a programmer with experience, good understanding of the events can make it very easy to program for SAX or StAX APIs. In addition, when most of the data from large XML files is needed, event based APIs should be preferred over tree based when programming ease is considered.

4.6 Comparing the work in this thesis to related previous work

As mentioned in Chapter 2, some work has been done in the past related to the XML parsers ([1][19][20]). But the work in [1] and [19] compares the performance of the XML parsers and not the APIs. An API must be used through which an XML document can be read by a parser. In [1], comparisons of SAX, DOM and StAX APIs are given. Functionality is compared but performance is not compared. In [20] the performance of SAX, DOM and StAX is compared, but the performance benchmarks are not the same as in this thesis. The work in [20] also does not include JDOM, DOM4J and XPath and it does not deal with creating the XML files. This work only compares the functional characteristics rather than performance characteristics, i.e., they compare XML parsers and not XML APIs.
5. CONCLUSIONS AND FUTURE WORK

In this thesis we have compared the performance of various APIs that are used for XML file processing. We came up with a step by step process in the form of questions which helped in determining a set of experiments to compare performance. All the experiments were performed on two commonly occurring systems, Windows 7 and Mac OS.

From the results in chapter 4, it can be concluded that StAX API has always performed best in terms of performance time. SAX has the second best execution time. DOM API has taken less time as compared to DOM4J and JDOM for files of size 5KB. But as the size of the XML file increases, every tree based API (DOM, DOM4J and JDOM) takes more or less the same amount of time. To conclude, we can say that streaming type APIs (SAX and StAX) have worked faster than tree type APIs, in both reading and writing scenarios. The query language for XML, XPath, is easy to code but is the slowest of all the APIs considered. Hence it should only be used when the amount of data to be extracted is minimal and execution time is not an issue.

Considering the programming ease, tree based APIs are easier to code as they are similar to the code of traversing trees. But when most of the data from the XML file is needed and files are large, streaming APIs should be preferred since the programming in this case is straightforward.

For future work, there are additional types of APIs such as VTD [22] which works in a complete by different manner. Comparing the performance and programming ease of these other APIs can be a good topic for future work.
Also in this thesis, we compared the performance for creating small XML documents. In future this can be expanded to creating XML files of bigger sizes and comparing the performance for creating them.
REFERENCES


