IBXP - Index Based XML Parser

Authors: Upendra Kumar Jariya, Rahul Kumar Sharma

http://code.google.com/p/ibxp/

Abstract

This work aims at creating an efficient DOM parser for XML documents. The main attraction for IBXP is significant reduction of processing and storage overheads by using the power of indexing mechanism and lighter data structures.

Existing DOM parsers break the input XML string (character sequence) into many small substrings (tokens) and then create a tree of objects (the DOM) in the memory. Apart from this, heavy data-structures like Maps and Lists are used to store attributes and child nodes respectively. This process consumes both, time and the memory.

In IBXP, the XML data is stored in the form of a single string and a data structure containing integer offsets is created to represent the DOM. This way, only offsets are stored to navigate through all the elements, children, attributes and other nodes; this significantly reduces initial processing time and memory consumption both. Further updates to the DOM are maintained in a separate modify-buffer. The algorithm may also use meta-data to keep the search faster.

Thus, IBXP becomes more efficient by using indexes and reducing the object creation and could save significant processing time and memory as compared to existing open source DOM parsers. The algorithm currently targets Java platform and will support the current version of JAXP.

Keywords: IBXP, XML, Parser, Index, JAXP, DOM, Java.

1 Target Audience

Target audience for this paper includes following:

- People who are familiar with XML usage in software development
- Algorithm designers and research scholars
- Architects and Developers working on XML based design.
- People working on parsing techniques
- People working on Data Structure design
- People studying in Graduate/ Post Graduate programs in Computer Science, Information Technology and software related courses

2 Introduction

Apart from the memory allocated to the member variables, the JVM adds internal information to each allocated object to help in the garbage collection process. It also adds other information required by the Java language definition, which is needed in order to implement such features as the ability to synchronize on any object. This extra memory per object is termed as object overhead. Although the object overhead varies for the JVM being used, machine's architecture and the operating system but usually it takes 1 or 2 extra words per object. Modern computers have a word size of 16, 32, or 64 bits. This way 4, 8, or 16 extra bytes are consumed as object overhead. Since this space overhead is a per object value, so the percentage of overhead decreases with larger objects. It can even lead to out of memory errors when you're working with large numbers of small objects.

The primitive types in Java are boolean, byte, char, double, float, int, long, and short. When you create a variable of one of these types, there is no object creation overhead and no garbage collection overhead when you're done using it. Instead, the JVM allocates the variable directly on the stack (if it's a local method variable) or within the memory used for the containing object (if it's a member variable).

Existing XML parsers tokenize the input XML into many small sub-strings and then DOM parsers create an in-memory DOM tree of those String objects. Since this process involves creation of many small sub-strings and other supporting objects, huge amount of time and memory gets wasted. Under the project IBXP, we are trying to reduce the memory and time consumption of XML parsing.

3 Key Aspects of IBXP

- XML Data for all nodes is kept collectively in the form of string
- Numbers are used as pointers to access the Nodes
- **Interfaces to the API would remain same as of the org.w3c.dom** i.e. the JAXP API
- Supports Read and Write both functionalities
- Lightweight data structure simulating the DOM tree
- On demand access to various tokens i.e. Node Name, Node Value etc.

4 Scope

This paper and the algorithm currently targets Java platform and will support the current version of JAXP.

All the functionalities available in JAXP could not be addressed simultaneously; hence only basic algorithm and design is under consideration at present.

5 Design

IBXP is based on a data structure centric design. The main idea is to store the complete XML data in the form of string (character sequence) and having organized indexes to access necessary information as and when needed. Runtime updates to the XML are also stored as string and new indexes are created to add this information in the existing DOM document.
This way, a substring from the XML data is extracted only when specific information (e.g. Node Name, Attribute Valve etc.) is required by the user.

5.1 Data Structure

Following diagram presents an outline view of the data structure being used

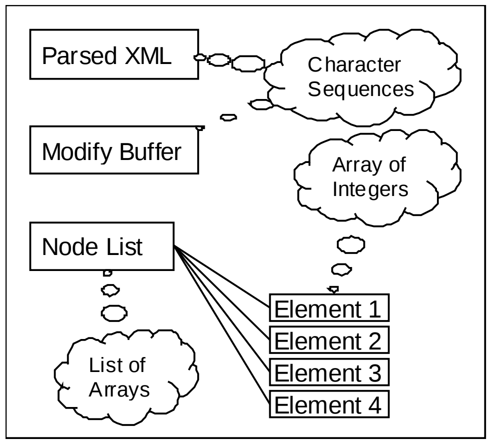


Figure 1 Data Structure

5.1.1 Parsed XML

This is a string (character sequence) which is used to store the complete text of the incoming XML document with a few changes.

5.1.2 Modify Buffer

This buffer (character sequence) is used for runtime modifications in the XML data.
When creating a new XML document from the scratch or after completion of parsing the incoming XML document, all the new nodes or attributes added to the DOM will be stored in this 'modify buffer'.

5.1.3 Element Array

Integer arrays are used to store all the Node specific information including parent node, node type, children, namespace, and starting indexes of names/values of nodes/attributes.
One array stores information about one element i.e. number of elements is equal to the number of 'element arrays'.

5.1.4 Node List

This is a List which contains integer arrays representing individual Nodes. Every 'element array' gets added to this list after its creation.

5.1.5 Indexes

Integers ('int' data-type) are used as indexes. There are mainly two types of indexes; one refers to a location in the 'parsed xml' or 'modify buffer' and another type refers to a location in the 'node list'.

All the 'indexes' are saved in the 'element array' and the 'element array' gets stored in the 'node list'.

5.2 Detailed Design

5.2.1 Few general rules

- If the value of any index is '-1', it means 'no value specified'. This rule has few exceptions e.g. 'node name' is a must.
- '-2' represents start of attributes in the element array
- '-3' represents start of children in the element array

5.2.2 Element Structure

Following diagram depicts a single XML node in the form of the 'element array'

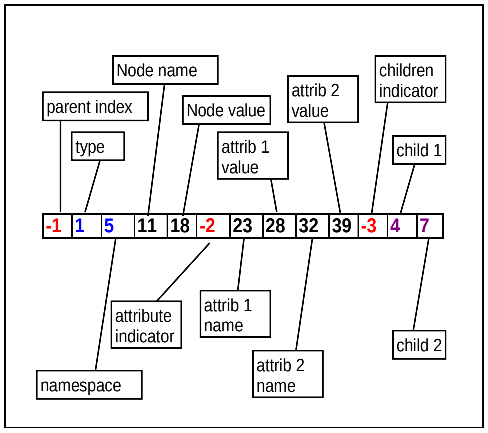


Figure 2 Element Structure

5.2.2.1 Parent Index

This field stores the index of the parent node in the 'node list'. Here '-1' represents that the current node is the 'root node'.

5.2.2.2 Node Type

Node type specifies the type of the current node as defined in org.w3c.dom.Node interface e.g. CDATA Node, Element Node and Comment etc.

5.2.2.3 Namespace Index

Value of this field indicates the index of the namespace of this node.
This functionality is not being supported in the current design; a Map shall be added to the data structure later to keep the mapping of 'namespace index' verses 'namespace'.
Here, '-1' represents 'no namespace specified'.

5.2.2.4 Node Name Index

This field specifies the index of the name of this node in the parsed XML string or the modify buffer.

5.2.2.5 Node Value Index

This field specifies the index of the value of this node in the parsed XML string or the modify buffer.
Here, '-1' represents 'no value specified'.

5.2.2.6 Attribute Name Index

This field specifies the index of the name of this attribute in the parsed XML string or the modify buffer.

5.2.2.7 Attribute Value Index

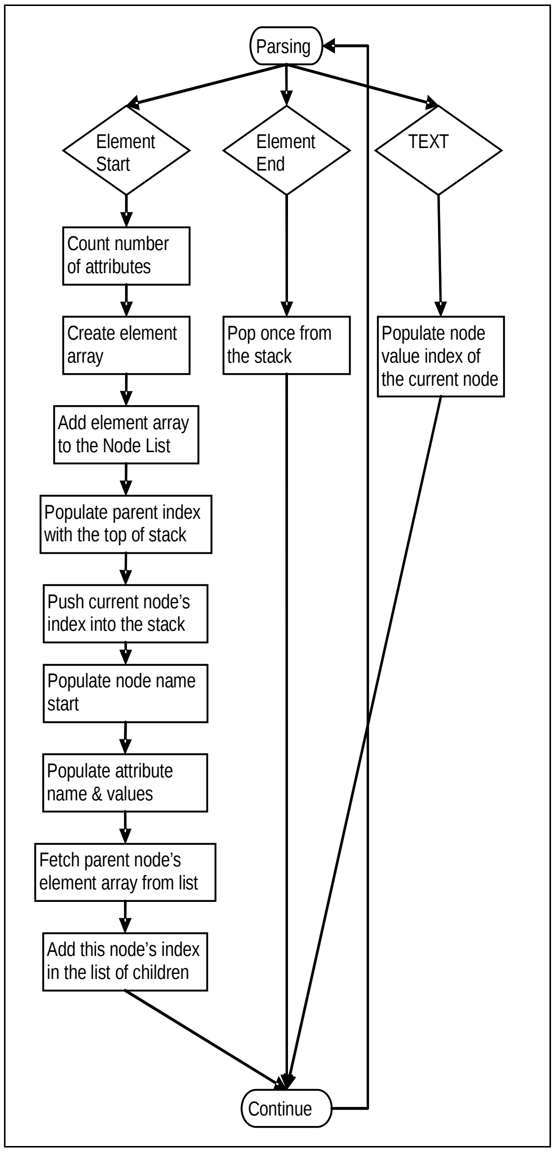
This field specifies the index of the value of this attribute in the parsed XML string or the modify buffer.
Here, '-1' represents 'no value specified'.

5.2.2.8 Child Index

This field stores index of the child element of this element in the node list.

5.3 Algorithm

Following flow-chart gives an outline of the algorithm used for populating the data structure



Parsing
Element Start
Element End
TEXT
Count number of attributes
Create element array
Add element array to the Node List
Populate parent index with the top of stack
Push current node's index into the stack
Populate node name start
Populate attribute name & values
Fetch parent node's element array from list
Add this node's index in the list of children
Pop once from the stack
Populate node value index of the current node
Continue

Figure 3 Algorithm

Here, the node at top of the stack is the current node.
This algorithm is still a work in progress and amenable to change during the course of further development.

6 Example

Go through the following example to get more understanding about the data structure and working of the program

Here are the contents of the input XML file

```
<?xml version="1.0" encoding="utf-8" ?>
<order>
  <book ISBN="0942407296">
    <title>Baking Extravagant Pastries with
Kumquats</title>
    <author>
      <lastName>Contino</lastName>
      <firstName>Chuck</firstName>
    </author>
    <pageCount>238</pageCount>
  </book>
  <book ISBN="0865436401">
    <title>Emu Care and Breeding</title>
    <editor>
      <lastName>Case</lastName>
      <firstName>Justin</firstName>
    </editor>
    <pageCount>115</pageCount>
  </book>
</order>
```

Figure 4 Example XML Data

After parsing, the Parsed XML string becomes

```
<order<0 <book<1 ISBN "0942407296" <title<2
"Baking Extravagant Pastries with Kumquats"
<author<3 <lastName<4 "Contino" <firstName<5
"Chuck" <pageCount<6 "238" <book<7 ISBN
"0865436401" <title<8 "Emu Care and
Breeding" <editor<9 <lastName<10 "Case"
<firstName<11 "Justin" <pageCount<12 "115"
```

Figure 5 Modified XML Data

And the element arrays contain following indexes

Parent	Node Type	Namespace	Node Name	Node Value	Indicator	Indexes or Indicators					
-1	1	0	1	-1	-3	1	7				
0	1	0	10	-1	-2	17	23	-3	2	3	6
1	1	0	36	45							
1	1	0	89	-1	-3	4	5				
3	1	0	99	111							
3	1	0	121	134							
1	1	0	142	155							
0	1	0	161	-1	-2	168	174	-3	8	9	12
7	1	0	187	196							
7	1	0	220	-1	-3	10	11				
9	1	0	230	243							
9	1	0	250	264							
7	1	0	273	287							

7 Benefits

7.1 Memory Efficiency

As the complete XML data is stored as shared string instead of creating different string object for every token, huge amount of memory is saved. Also since we are creating integer indexes instead of creating java objects for every node or attribute, good amount of memory cut-off is achieved here as well.

7.2 Performance Gain

Since new objects are not created corresponding to every token of the XML string, the time consumed due to 'object overheads' is considerably saved. And this way, the parsing is achieved in much lesser time than using regular DOM parsers.

7.3 Supporting JAXP

Since we are supporting JAXP interfaces and providing a much more efficient implementation, no one using any of standard JAXP compliant implementation would need to change his code to use our APIs as we would support JAXP interfaces.

8 Limitations

Present implementation uses SAX parser for initial parsing; events generated by the SAX parser are further used to create the IBXP data structure. We are working to overcome this limitation.

9 Challenges

- Coping with the increasing length of strings and node lists
- Memory clean-ups
- Efficient management of memory and managing time complexity

10 Future Prospects

- To support attribute values to be Objects and JAXP's setUserData functionality, an 'object list' shall be added to the data structure
- Complete support of JAXP API shall be provided
- Other related features like XSLT, XML Comparison and XPATH API etc. shall also be implemented.

11 Implementation

At present we've used SAX parser just to boost-up the development process. But the project scope includes development of a complete parsing solution right from the scratch.
Minimum functions from the JAXP are being implemented for 'Proof of Concept'.

12 Comparison

Present DOM parser consume ample amount of time and memory because they explode the incoming XML string into many small sub-strings called as tokens and create the DOM tree which is a simple tree structure. Apart from this, few of the parsers contain an Array List to save the children references and a Hash Map to store all the attributes.
We tried to skip the creation of most of the unnecessary objects to minimize object overheads and also to gain memory efficiency and performance.

12.1 Test Results

All the tests were performed on the machine having following configuration: Intel Pentium 4 CPU, 2.9 GHz machine, 2 GB of RAM, Windows XP, and JDK 1.4. Xerces implementation has been used for SAX and DOM parsers.

12.1.1 Performance Test

With our current implementation (using SAX), as of yet, our observations show that if the XML data size is up-to 100 KB, then the parsing takes much lesser time then that of the DOM. But as the size of incoming XML data increases, the time reaches to the same as of the DOM. When the XML data size grows up-to and beyond 1 MB, the time consumption exceeds than that of the DOM.

Following chart presents the performance test results from our first test with the initial implementation

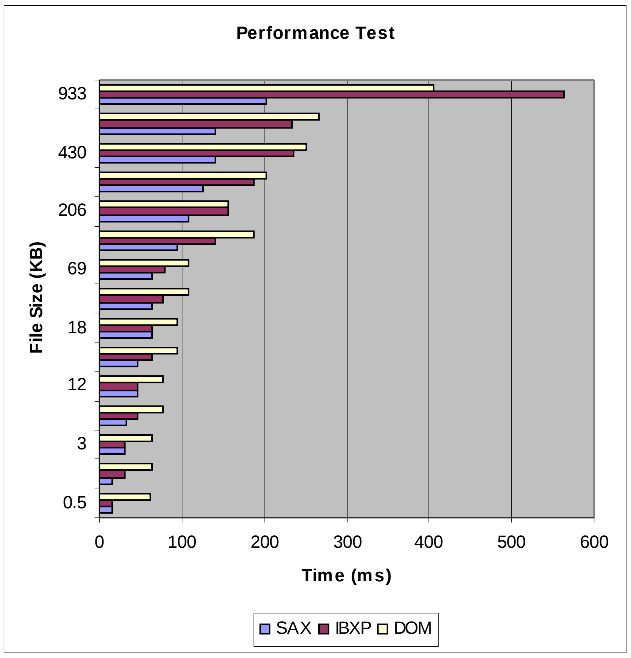


Figure 6 Performance Test Results

12.1.2 Memory consumption

We have calculated the memory consumptions for both IBXP and DOM. For IBXP, we measured the 'deep size' of 'data structure' object which is explained above in this document. For DOM parser, we measured the 'deep size' of the instance of 'org.apache.xerces.dom.DeferredDocumentImpl' class.

We used trial version of JProfiler performance analysis tool for performing these memory comparisons. Following diagrams present the memory comparison between DOM and IBXP

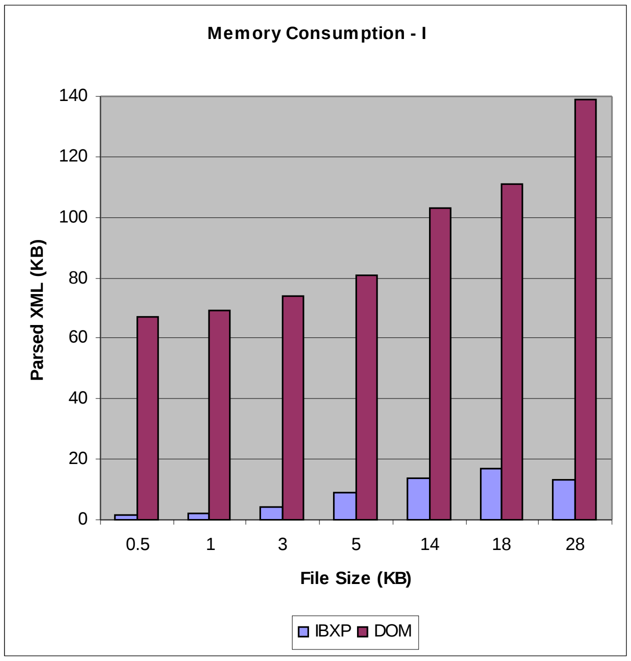


Figure 7 Memory Comparison - I

For another set of tests, following is the chart of memory comparison

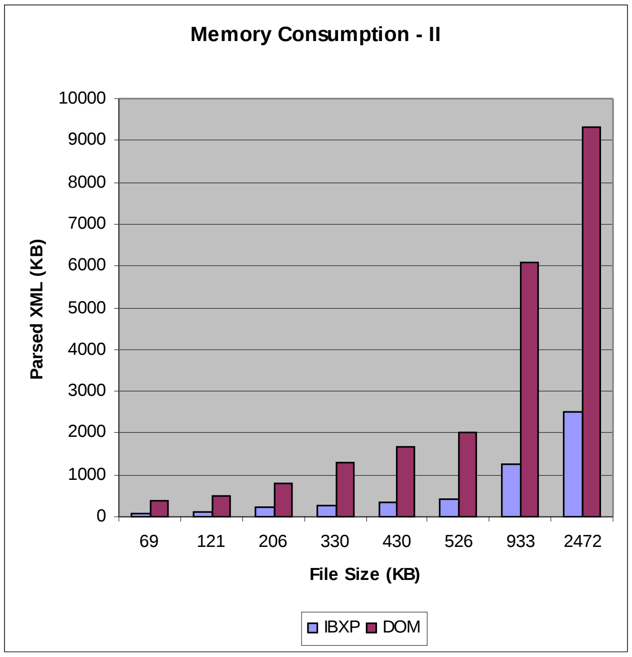


Figure 8 Memory Comparisons - II

As it is clearly visible from the charts above, IBXP consumes much smaller amount of memory as compared to regular DOM parsers.

13 Notes

Please note that,

- In this document, the terms indexes and pointers are used interchangeably.
- 'Element' in this document, includes all type of nodes except the attributes.
- Minimum number of algorithms required are created and specified as of yet. More algorithms will have to be designed as the work progresses.
- The data structure described above may also get a few changes as the work gets progress.

References

[1] XML Specification (*W3C Recommendation for Extensible Markup Language 1.0*)
[2] Java Virtual Machine Specification (Tim Lindholm, Frank Yellin)
[3] Java Language Specification (James Gosling, Bill Joy, Guy Steele, Gilad Bracha)
[4] http://www.javaworld.com/javaworld/jw-11-1999/jw-11-performance.html
[5] http://c2.com/cgi/wiki?JavaObjectOverheadIsRidiculous

Acknowledgements

We are sincerely thankful to Mr. Sayed Mohammad Bahauddin and Mr. Yogesh Mahajan for their suggestions, continuous support and good wishes during the course of this project.

Contact Information

Upendra Kumar Jariya
B.E., Information Technology, 2004
S.G.S.I.T.S. Indore
upendra.jariya@gmail.com,
upendra_jariya@yahoo.com

Rahul Kumar Sharma
B.E., Information Technology, 2004
S.G.S.I.T.S. Indore
rahulsharma21@gmail.com