Compressed Representation of XML Documents with Rapid Navigation

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by

MOHAMMAD KAMEL KHARABSHEH
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF LEICESTER

JANUARY 2014

Declaration of Authorship

I hereby declare that content of this thesis is my own work and that it is the result of work done during the period of registration. To the best of my knowledge, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

Abstract

XML(Extensible Markup Language) is a language used in data representation and storage, and transmission and manipulation of data. Excessive memory consumption is an important challenge when representing XML documents in main memory. Document Object Model (DOM) APIs are used in a processing level that provides access to all parts of XML documents through the navigation operations. Although DOM serves as a a general purpose tool that can be used in different applications, it has high memory cost particularly if using naïve. The space usage of DOM has been reduced significantly while keeping fast processing speeds, by use of succinct data structures in SiXDOM [1]. However, SiXDOM does not explore in depth XML data compression principles to improve in-memory space usage. Such XML data compression techniques have been proven to be very effective in on-disk compression of XML document. In this thesis we propose a new approach to represent XML documents in-memory using XML data compression ideas to further reduce space usage while rapidly supporting operations of the kind supported by DOM.

Our approach is based upon a compression method [2] which represents an XML document as a directed acyclic graph (DAG) by sharing common subtrees. However, this approach does not permit the representation of attributes and textual data, and furthermore, a naive implementation of this idea gives very poor space usage relative to other space-efficient DOM implementations [1]. In order to realise the potential of this compression method as an in-memory representation, a number of optimisations are made by application of succinct data structures and variable-length encoding. Furthermore, a framework for supporting attribute and textual data nodes is introduced. Finally, we propose a novel approach to representing the textual data using Minimal Perfect Hashing(MPH).

We have implemented our ideas in a software library called DAGDOM and performed extensive experimental evaluation on a number of standard XML files. DAGDOM yields a good result and we are able to obtain significant space reductions over existing space-efficient DOM implementations (typically 2 to 5 times space reduction), with very modest degradations in CPU time for navigational operations.

Acknowledgements

First and foremost I would like to thank God. You have given me the power to believe in myself and pursue my dreams. I could never have done this without the faith I have in you, the Almighty.

I take immense pleasure in expressing my sincere and deep sense of gratitude to my supervisor, Professor Rajeev Raman, for his guidance, understanding, patience, encouragement, and, most importantly, his friendship. Without his supervision and constant help this thesis would not have been possible. Apart from the subject of my research, I learnt a lot from him, which will be useful in different stages of my life. I solemnly submit my honest and humble thanks to him for bringing my dreams into reality. I would also like to thank Professor Rick Thomas, PhD tutor, Dr Fer-Jan de Vries and Professor Thomas Erlebach, who morally boosted me and provided me with their guidance, support and assistance. I would also like to thank two special colleagues: Dr ONeil Delpratt for his assistance and kindness throughout the early stage of my research, and Stelios Joannou for his support and help during my research.

Also, I wish to extend my gratitude to thank my colleagues and staff at the Computer Science Department, University of Leicester, especially those members of my doctoral committee who have willingly and ably helped me out throughout my doctoral research.

I also thank my friends (too many to list here but you know who you are!) for providing support and friendship that I needed. Their motivation has given me the confidence to reach this stage in my work. I will never forget all the chats and beautiful moments I shared with some of my friends. They were fundamental in supporting me during stressful and difficult moments. I would like to thank Dr Sameeh Al-Sarayreh, Dr Anan Younes, Dr Maen Aljezawi, Dr Mustafa Al-khawaldeh, Dr Hamza Aldabbas, Mohammed Alabdullatif, Mohammad Alshira′h, Abdullah Audat, Moneer Nusir, Abdularhaman Alshabeb and Ayman Albarasneh, who personally supported and encouraged me during my PhD journey. My special gratitude goes to my friend Dr Ehsan Khalefa for her moral support and motivation, which drives me to give my best, and for being with me through the thick and thin of life. I find myself lucky to have friends like these in my life.

I would like to acknowledge the people who mean the world to me: my mom, my sister Reem, my brothers Qutaibah, Sa′ad and Obaidah, and Obaidahs wife Reham and his kids Kamel and Mohammad for their faith in me and allowing me to be as ambitious as I wanted. Their support and unending encouragement have been unconditional all these years. They have cherished with me every great moment and supported me whenever I needed it. I consider myself the luckiest man in the world to have such a supportive family standing behind me with their love and support.

Finally, this thesis is detected to the memory of my Father. He was a wonderful father and showed a genuine interest in and concern for my life, my work and my wellbeing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Extensible Markup Language (XML) is a language standardised by the World Wide Web Consortium (W3C). XML is a markup language and is a multipurpose data format designed to transport and store data. In addition, it is designed to be self-descriptive and you must define your own tags [10]. However, XML was created to overcome the HTML limitations, the Hypertext Markup Language that is the basis for all Web pages and that is designed to display data not to carry data [11]. HTML was designed for human web users. Therefore, the user knows the information, but the machine does not know what the information is. Now, developers can use XML with self-describing data to create documents and how people are using those documents to improve the web. More significantly, it is easy for a machine to process the information as well. XML is well-suited to the representation of complex, hierarchically structured data [11].

XML has a variety of uses in many aspects of web development, e-business, portable applications and IT applications. For example, in web publishing XML makes creating e-commerce applications perceptive by allowing the creation of interactive pages [12]. Many web pages are represented in XML, and it is easier and more efficient to retrieve useful results when using web searching and XML, making information access easier for applications and devices; there are XML-based standards for access and data exchange, such as e-business applications, Web-Services Description Language (WSDL) [13], Simple Object Access Protocol (SOAP) [14] and Universal Description, Discovery, Integration (UDDI) [15]. XML is also used to express metadata in reusable format, and to store scientific data such as the VO-Table XML format [16], and the MEDLINE XML format [17]. XML is used in databases, with a number of query languages that have been developed (e.g. XQuery) just like the SQL standard for traditional database systems; to provide access and retrieval of the data.

1.1 XML Processing

When working in XML there are several standard ways for processing XML documents. We take some of the standard APIs (Application Programming Interfaces) and languages to describe the processing of XML documents:

- *Simple API for XML (SAX)* [18]. This is an event-driven functionality, which accesses the XML document from the beginning to the end sequentially, and sends a stream of event to the programmer by call-back methods that the programmer has written.

- *Document Object Model (DOM) APIs* [3]. The DOM is very popular, and is used to parse and process XML documents. It is a standard for exposing document elements for programming functionality.

- *eXtensible Stylesheet Language Transformations (XSLT)* [19]. By using XSLT we can process multiple XML documents and transform XML documents to other XML documents or another form such as PDF, HTML, etc. XSLT applies user-defined transformation to an XML document.

- *XQuery* [20]. XML is a language for querying XML documents. XQuery uses a number of query languages to retrieve the data and is supported by all major databases [3], [7], [10]; also *XQuery* can express queries across all parts of XML documents and is built on *XPath* expressions.
- *XPath* [21] is a language for locating and processing nodes in an XML document. Because an XML document is a hierarchical structure and XPath operates on the tree structure of the XML documents, it becomes possible to use path expressions for finding information by navigating in XML documents.

To summarise, the SAX presents a linear view of the document, SAX does not provide any support for performing processing that requires a tree-like view of the XML document and SAX is read-only, i.e. we cannot insert or delete nodes. Unlike SAX, DOM requires a pre-processing phase to construct a representation of the document and will be able to access all parts of XML document representation through navigation operations.

SAX uses little memory in representation and is extremely fast and works for huge documents but the API is less intuitive as it is event-based. DOM is memory-based; the XML document is parsed to create a DOM tree structure, which makes it traversable and editable. Since the XML structure is resident in memory, the larger the XML structure is the more memory it will consume. In comparison to SAX, DOM can be much slower due to it is resource usage.

Although DOM is a relatively low-level interface and, as we mention above, has to load the entire document before data can be read, it is the basis for a large number of real-world applications (e.g. the Zorba [22] XQuery processors document store only supports XDM (XQuery and XPath Data Model) operations on the document: XDM and DOM are quite similar. DOM serves as a general-purpose tool that can be used as a stand-alone or with other standard applications and they are considered high-level processors, such as XPath, XSLT and XQuery [20]. In addition, DOM provides a robust API to easily modify and extract data from an XML document.

1.2 XML Bloat

As we mentioned above, XML has some advantages over things like CSV files, such as: it can represent hierarchical data simply and many applications use XML. However, XML is not especially good at handling very large amounts of data, can quickly become difficult to read if a lot of information is included in one file, and certain types of data (images, other binary data) are not represented well in XML.

XML files are always much larger than the flat file, because when representing any flat file XML adds tags to it to indicate the meaning of the text or to separate the document into sections. For example, if we take the file called *orders.xml* from the Relational Database Benchmark [9](see Appendix A for the full description of the standard files). The orders file contains the following field (O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS, O_TOTALPRICE, O_ORDERDATE, O_ORDER-PRIORITY, O_CLERK, O_SHIP-PRIORITY, and O_COMMENT) and if we take the first line from the `orders` flat file as follows: *1,370, O, 172799.49, 1996-01-02, 5-LOW, Clerk#000000951, 0, blithely final dolphins solve– blithely blithe packages nag blith.*

Then tags are added to this flat line to convert it to XML:

```
<T><O_ORDERKEY>1</O_ORDERKEY><O_CUSTKEY>370
</O_CUSTKEY><O_ORDERSTATUS>O</O_ORDERSTATUS>
<O_TOTALPRICE>172799.49</O_TOTALPRICE><O_ORDERDATE>1996-
01-02</O_ORDERDATE><O_ORDER-PRIORITY>5-LOW</O_ORDER-
PRIORITY><O_CLERK>Clerk#000000951</O_CLERK><O_SHIP
PRIORITY>0</O_SHIP PRIORITY><O_COMMENT> blithely final
dolphins solve– blithely blithe packages nag blith.</O_COMMENT> </T>
```

We can notice the size of the flat orders file is 1.6MB and 5.1MB as XML format, so the size of this file has tripled. This problem with XML documents is XML bloat. So this problem will increase the transmission times and the space usage, particularly in machines that have limited space, such as mobile devices. And the costs of storage/backup in PCs and servers will be increased.

1.3 XML Compression

One way to solve XML bloat and to address the space consumption of XML documents is through data compression. Compressors can be normal text compressors (Gzip, Bzip) or XML-specific compressors (Xmill [5] or Xbzip [6]). We achieved very good compression ratio from the XML-specific compressors. Unfortunately, the above-mentioned XML compressors do not support processing operations such as navigation without potentially decompressing the whole document. Some query-friendly XML compressors have recently been developed that support operations on the compressed file while requiring (at most) partial decompression, such as [5, 6]. Although some of these such as XBzipIndex [6] support DOM-like navigation, they are orders of magnitude slower than standard DOM implementations. We need to investigate in-memory XML document representations that support fast DOM-like operations.

1.4 In-Memory Representation

Many applications use XML not just as an archival format, but also process XML documents through standard interfaces/languages such as DOM, XSLT or XQuery, all of which store the XML document in a document tree held in the main memory.

The main memory (RAM) usage of standard implementations of these interfaces that support DOM-like operations such as Xerces can be up to 15 times larger than the (already 'bloated') XML file. The standard implementations of DOM use a naïve approach of representing the document tree. For example, in Xerces, an element node may contain pointers to: first child, last child, parent, next and previous siblings, element name, attributes, owner document, etc. These pointers are essential for navigation but come a very high space cost.

We are now going to consider the problems caused by excessive memory usage. We take a look at the memory usage of two implementations of DOM, Xerces [23] (in C++, a reference implementation of DOM) and a space-efficient DOM implementations called SiXDOM [7], in Table 1.1 which shows the space usage percentage of XML representations compared to the original size of the XML files. As an example, `proteins.xml` file of memory with size 600 MB after representation to process the size will occupy 11GB when using Xerces, and 163MB when using (SiXDOM) [7]. Therefore, if the memory needed to represent an XML document is greater than the physical RAM on the machine, then the document may not load, or a substantial part may be stored on Virtual Memory (VM). In other cases VM is either limited (e.g. in Java VM is at most 2GB), or not available at all (e.g. the Android operating

Table 1.1: Space usage of XML Representations in DOM(Xerces) and SiXDOM (in *seconds*)

File Name	Size(MB)	SiXDOM	DOM(Xerces)
Orders	5.1	98%	1051%
Protiens	600.0	23%	1525%
Factor1	116.5	52%	404%
Factor2	233.7	23%	1608%

Table 1.2: Running time of XML Representations in DOM(Xerces) and SiXDOM

File Name	Size(MB)	SiXDOM	DOM(Xerces)
Orders	5.1	0.02	0.01
Proteins	600.0	5.18	360.65
Factor1	116.5	0.43	0.19
Factor2	233.7	0.87	0.37

system). We can notice some XML files such as `factor96.xml` (with size 2.9 GB) in our tests were not able to be processed by our test machine when using Xerces; this is because they exceed the maximum virtual space.

In Table 1.2 we draw comparisons of running time (wall_clock time in a *second*) when visiting all nodes in XML documents (traversal speed) between SiXDOM and Xerces. Table 1.2 shows XML processing much slower and badly affects the traverse speed when we need a huge amount of memory to represent the XML documents such as `proteins.xml`

1.4.1 Our Approach

In summary, the problem of representation of XML documents in main memory still exists and the above-mentioned XML compressors do not support processing operations such as navigation without potentially decompressing the whole document.

In [7] the DOM is implemented using succinct data structures. As an example, SiXDOM avoids using pointers between nodes in the XML document tree by storing a balanced parenthesis string of `2n` bits to encode the tree structure, and using a succinct index to perform rapid navigation in the tree. As we noted in Section 1.4, SiXDOM [7] shows better savings of the memory usage, but SiXDOM did not use any kind of compression in representations. For example, a highly regular tree with `n` nodes and a randomly generated tree with `n` nodes would both take `2n` bits to represent. XML trees, however, are generally quite regular and therefore should be compressible.

Therefore, the objective of this research is to achieve DOM functionality with speed performance while using tree compression methods to exploit the regular structure of typical XML documents, and in addition to improve the space usage.

The objectives of this thesis are as follows:

(a) To develop a space-efficient in-memory representation of XML documents with memory usage an order of magnitude less than existing DOM implementations.

(b) Fast support for DOM operations at a speed that is comparable to standard DOM implementations.

1.5 Contributions and Organisation of Thesis

In this thesis we propose a new approach called DAGDOM to support navigation operations with space efficiency regarding the in-memory representation, as follows:

(a) We introduce a new approach to represent XML documents in-memory in a highly-compressed format, while supporting operations of the kind supported by DOM. More details are as follows:

- We created a compressed in-memory representation in order to support fast navigation without the need for decompression, by using a standard DOM API. At the first stage we achieved this by making navigation operations in the compressed tree as a directed acyclic graph (DAG) by sharing common subtrees, called a compressed skeleton, with the same efficiency as navigating on the virtual tree (original tree without compression). But, at this stage, we ignored the other components of XML document such as attribute and text data.

- We improved the space usage of the compressed skeleton representation by application of other space-efficient Xerces DOM implementations (SiXDOM) [1].

- We developed the compression method which we used to represent the tree skeleton in-memory, and further compression is performed to yield a shrunken skeleton in order to obtain significant space reductions, in addition to handling all components of the XML document such as attribute and text data.

- We made further improvements in DAGDOM by creating a new representation for any sequence of integers called *Variable Length Encoding.* This representation contains five approaches of encoding called: *Naïve approach, Threshold, Byte Based, Bit Based and Fixed Bit Based.*

(b) We advance the knowledge of representing the textual data in XML documents by investigating the efficient representation and access of the individual textual data. Where existing solutions focus on the textual data compression, we show the importance of compressing the pointers to the individual textual data themselves, which would generally be expensive. We present new strategies based upon grouping together the textual data that shares the same parent element and give them the same label, then used the minimal perfect hash function in order to maintain the mapping strategy between skeleton document order and textual element order numbering very space-efficiently.

(c) We provide experimental evaluation of DAGDOM against other DOM implementation and a space-efficient DOM implementations. There is a significant space reduction using DAGDOM over existing space-efficient DOM implementations (typically 2 to 5 times space reduction), with very modest degradations in CPU time for navigational operations.

The rest of the thesis is organised as follows: Chapter 2 gives background details of XML, the DOM architecture and standards, and Succinct Data Structures. Chapter 3 gives the overview of XML compression and details of XML compressors with support for DOM-like in-memory representation, as well as an overview of SiXDOM implementation details. In Chapter 4, we define the properties of the compression methods and the first implementation of DAGDOM. Chapter 5 presents an in-memory representation of the XML document using DAGDOM, and how we developed the compression method, and presents a new variable length encoding. Chapter 6 giving a study of representing the text and attribute node representation of XML documents efficiently, also presents a strategy to efficiently store and access textual data contained in XML documents. Finally, in Chapter 7 we give the closing remarks of the thesis achievements and contributions, and outline future development of DAGDOM.

Chapter 2

Preliminaries

In this chapter, we introduce basic background knowledge on XML. Then we give details of the DOM specification. We also discuss succinct data structures and finally we give an introduction to libbzip2.

In Section 2.1, we will give an overview about XML, the syntax rules of XML, components of an XML document and we will talk about XML validation. In addition we will give an example that shows the XML document as XML tree. Section 2.2 will focus on the DOM architecture and standards. In this section we will talk about DOM node types. Next, in Section 2.3 we will discuss parsing and traversing of XML documents by using Node API and `TreeWalker`. In Section 2.4, we will discuss succinct data structures and we will give examples to show the differences between naïve and succinct representation. Finally, in Section 2.5 we will give an introduction to the *Burrows-Wheeler* transform in addition to an overview of Bzip2 and how the block size affects compression performance.

2.1 XML

As we mentioned in Chapter 1, XML is a markup language was designed to transport and store data

2.1.1 Markup and Text

Generally, the flat file is just a text; when we add the tags to structure and describe these data then the data are marked up and called an XML document. The markup is to put the text itself inside the symbols `<` and `>`. For example, if we look at a file containing the following fields (STUDENT_ ID, STUDENT_NAME, DOB), and containing the following data under the fields: 099019620, Tom Smith, 19-01-1990, markups construct the tags beginning with `<` and ending with `>`, which are called start-tags, in addition to ending the tags by end-tags beginning with `</` and ending with `>`, then the XML document will be as follows:

```
<STUDENT_ID>099019620 </STUDENT_ID><STUDENT_NAME> Tom Smith
</STUDENT_NAME><DOB>19-01-190</DOB>
```

The tags help to distinguish a piece of text from any other piece of text, and often give information about, or provide meaning, to the text they contain. From the above example, in element `<STUDENT_NAME>` Tom Smith `</STUDENT_NAME>` we know that the content is probably a string for the student name. Elements cannot stand alone unless they are the root element or there is only one element, and may contain other elements providing they are properly nested, and they must be contained within a hierarchy of elements that begins with the root element.

2.1.2 Components of an XML Document

Each XML document contains several components, see Example 2.1. Optional components include *document prolog* at the start of the XML documents which contains the XML declaration and DTD [24]. e.g. `<?xml version="1.0">`, and processing instructions which appear on the prolog or the body of the XML document. *Document instance* component: this is the main part of XML documents after the prolog which contains the following sub-components:

- `Elements`: these are labels inside the markup symbols `<` and `>`. Each XML document considers the first element on the document after the prolog as a root element like `<biblio>` in Example 2.1. The elements must follow the XML naming rule. The XML element names contain letters, digits and other characters, but must start with a letter, colon or underscore and cannot contain spaces.

- `Attributes`: these represent certain properties of the elements in the XML document. The attributes appear after the element name and before closing the opening tag and they consist of attribute name (which is an XML name) followed by equal sign, then the attribute must have a value between double (or single) quotes. In Example 2.1 see the `id` (attribute name) with value '1' (attribute value) after the element `book`.

- `Entity References`: used when we need to use special characters, and cannot enter them on the keyboard. The `Entity References` are preceded by the ampersand (&) symbol followed by a semicolon (;), the XML specification defines five entities in XML:
  - `&quot;` (double quotation mark)
  - `&amp;` (ampersand)

  - `&apos;` (apostrophe)
  - `&lt;` (less than sign)
  - `&gt;` (greater than sign)

- `Comments`: these are used to make explanatory notes on the XML document appear in the form `<!-- -->` like the HTML comments.

- `CDATA section`: this markup is used when we need to insert any kind of text, and is interpreted as characters not as markup or entity references. For example, sub-program code like: `<![CDATA[ while(int i=1; i<=5) Product= Product*i; i++ ]]>`.

- `Processing Instruction`: these allow the document to insert instructions for applications and are enclosed between `<?` and `>`, the XML name, called the target, followed by a list of value pairs called the data. The common use of the processing instruction is for the style-sheet, for example:

  `<?xml-stylesheet href= "headlines.css" type="text/css" ?>` (style-sheet declaration connected to the document).

Example 2.1: Part of a Large Bibliographic Database [6]

```
<?xml version="1.0">
  <biblio>
    <book id = "1">
    <author>J. Austin</author>
    <title>Emma</title>
    </book>
    <book id = "2">
    <author>C. Bronte</author>
    <title>Jane Eyre</title>
    </book>
  </biblio>
```

2.1.3 Well-Formed and Valid XML Documents

XML documents are classified into two levels:

- Well-formed XML documents, which obey to the syntax of rules that defined by XML specification [10]. The document contains XML tags and these tags, as we mention above, contain a piece of text. All the tags are nested properly and start from a single element root and the data values should be within start and end tags.
- Valid XML documents: this level must conform to the well-formed level and check the rules that it is defined in a Document Type Definition (DTD). The DTD rules are associated with a file that describes the format of an XML document's markup [25].

2.1.4 XML Tree

In an XML tree we can see the hierarchical structure of the element content in the document instance. The XML tree labelled the element names as tree nodes and if the elements have data values these will be stored at the leaves in the tree. In Section 2.2, Figure 2.1 we will see as an example the order of element nodes reading from top to bottom with the same order of the elements in the document after we parse the XML document (such as Example 2.1 in Section 2.1.2) as a tree structure.

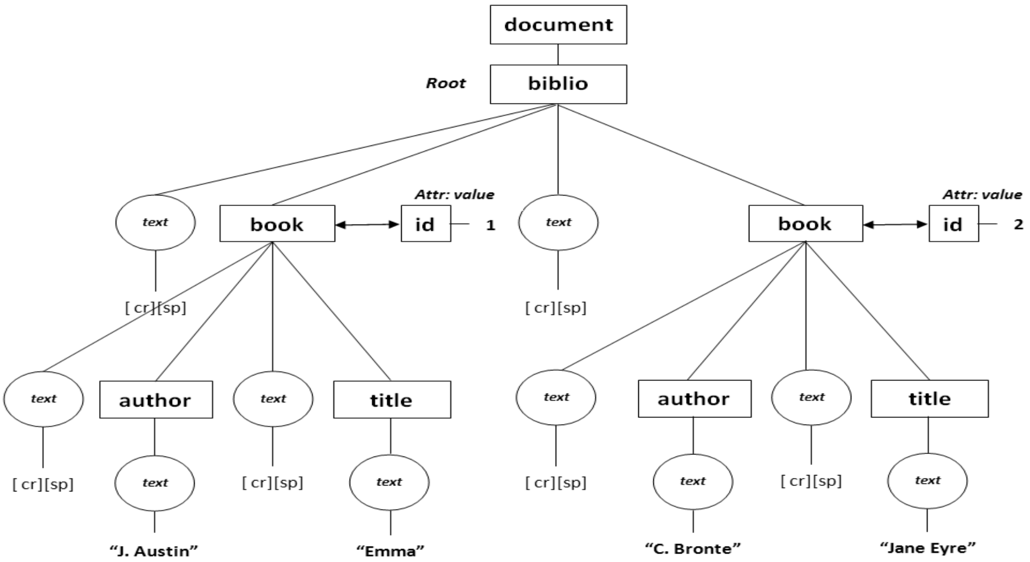


Figure 2.1: Representation of the XML document as a DOM tree [3]

2.2 DOM Architecture and Standards

The DOM is used to access and manipulate XML and HTML documents by a set of application programming interfaces (APIs) that defines the logical structure [4], [26]. The DOM APIs are divided into groups called modules and categorised into levels. According to the feature they support each module will be given a name, and each level has its own operation for the APIs.

The DOM parses the entire XML document as a tree structure in-memory using a standard parser like a SAX parser and loads the entire document. Through the navigation operations, the DOM provides access to all parts of the representation of the XML document and provides flexibility of repeated navigation, retrieval and/or update of the document. In addition, it serves as a general-purpose tool that can be used stand-alone or with other standard applications, such as XPath, XSLT and XQuery [21], [19], [20] [27], [28]. An example of the DOM is given in Figure 2.1, which represents the XML document in Example 2.1 as a DOM document tree.

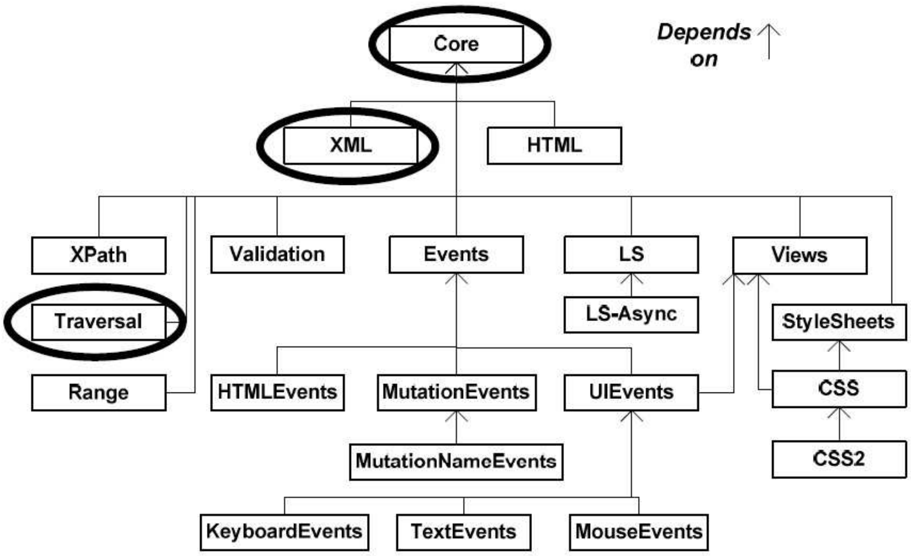


FIGURE 2.2: DOM modules defined in the DOM specification [4]

The DOM is divided into modules such as Core Module, XML Module, and Traversal Module as we can see in Figure 2.2. These modules contain the DOM APIs [3]. Further details will be given in the next sections.

2.2.1 DOM Node Types

The APIs in the Core Module contains the Node (Basic API), and there are twelve values, based upon the type of node as follows:

- `Element` node: when we represent the element in the XML document, then we can access these elements through the `Node` and `Element` interface.
- `Attribute` node: this type of node is not part of the DOM tree and represents the attributes of an element node in the XML document; we can access this

type of node by the variable in the `Node` interface which is a `NameNodeMap` interface.

- `Text` node: this node is the textual value of an element in the XML document which appears as a leaf node only; we can access it through `Node`, `Text` and `CharacterData` interface
- `CDATASection` node: we can access this type of node through `Node` and `CDATASection` interfaces.
- `EntityReference` node: we can access this type of node through the `Node` and `EntityReference` interface.
- `Entity` node: we can access this type of node by interface `Node` and `Entity` interface.
- `ProcessingInstruction` node: this leaf in the DOM tree represents a processing instruction in the XML document.
- `Comment` node: this node appears as a leaf in the DOM tree and represents a comment in the XML document.
- `Document` node: in the XML document there is a single root to the document which is represented at the root of the DOM that supports the creation of node objects and we can access the entire DOM tree.
- `DocumentType` node: we can see this kind of node as a child of the document node and it represents the DTD in the XML document. The DOM tree has only one instance of the `DocumentType` node.
- `DocumentFragment` node: this kind of node is used in dynamic implementations of DOM and represents a sub-tree inserted into the DOM tree.

- `Notation` node: notations defined in the DTD and having no parent nodes, represent a notation in the XML document.

2.3 Parsing and Traversing XML Documents.

2.3.1 Parsing the XML Document

In order to read and to manipulate XML documents, we can use an XML parser in a processing phase (such as `Xerces` [23], [29]) to read the XML document and convert it to the XML DOM object in memory. After that we can use the traversal module for navigation as is shown in the next section.

2.3.2 Traversing an XML Document Using Node API

As we mentioned before, the `Node` interface contains a number of variables, such as `nodeName`, `nodevalue` and `attributes`, and also contains the navigation operations based upon the DOM tree as follows: `firstChild`, `nextSibling`, `previousSibling`, `parent` and `lastChild`.

In the traversal module there is *Document order*, which is the order in which nodes are visited in the document during parsing. The node ordering that is the opposite of document order is called *Reverse document order*.

We can traverse the document based on the navigation operations of the `Node` interface. For example, in *document order* traversal we start from the root node to navigate the DOM tree. Through all first child nodes and not at a leaf node or if we have visited the current node's sub-tree before, we then navigate to the right sibling nodes. We repeat this process until we have reached the last leaf node on the right-most sub-tree. In *Reverse document order* from the right-most sub-tree we can use the `previousSibling`, `lastChild` and `parent` to complete traversal.

2.3.3 Traversing an XML Document Using TreeWalker

Both orders of traversal are applied to the TreeWalker or NodeIterator interfaces as follows:

- The `NodeIterator` helps us to view the XML documents logically in a flat manner. We can move forward, as in an array of nodes by the operation `nextNode()`, and this represents document-order. By the operation `previousNode()` we can move backward, and this represents reverse document order traversal.
- The `TreeWalker` has similar operations to the `Node API` operation used in tree navigation in addition to the three operations: `nextNode()`, `PreviousNode()`and `getCurrentNode()`. All the `TreeWalker` operations are used to maintain the tree (or sub-tree) structure of the document. When we call any navigation operation that returns a node to the user, then the iterator of the current node which is held within `TreeWalker` will be updated, providing that the node returned is not null.

  The main difference between these two interfaces is that the TreeWalker presents a tree-oriented view of the nodes in a subtree, rather than the iterator's list-oriented view. In other words, an iterator allows you to move forward or back, but a TreeWalker allows you to also move to the parent of a node, to one of its children, or to a sibling [30]

2.4 Succinct Data Structures

This section is based on Chapter 4 of the thesis by Delpratt [7]).

Succinct data structures use space that approaches the information-theoretic lower bound on the space that is required to represent the data, and support operations upon the representation in constant time. When we need to represent certain data objects (for each kind of object) by succinct data structures we begin by giving their succinct lower bound [8]. In order to support a number of operations upon the data object, we then discuss the corresponding data structures that use a small amount of space in addition to the succinct bound. We now give examples of succinct lower bounds.

2.4.1 Bit-Vector Data Structure

The set of objects in this core is the set of all bit-strings of length n. It is assumed that the algorithm knows that it has to store a bit-string, and also knows the length of the bit-string. Since there are 2^n such bit-strings, by taking the logarithm base two of this number, we get:

Proposition 2.1:([7], Chapter 4) *The succinct lower bound for representing a bit-string of length n bits is n bits.*

Operations

Efficient query operations such as `RANK` and `SELECT` to be used to encode a sequence of variable-length items (more details in Chapter 5, Section 5.3)

`RANK` and `SELECT` are two operations that are used in a bit-string representation, such as a Bit_Vector data structure [31], [32], represent the object as a bit-string and use these operations to be support. For a bit-string `x` of length `n`, the definitions of these operations are:

- `SELECT`$_1$`(x,i)`: Given an index `i`, returns the position of the `ith` **1** bit in `x`
- `RANK`$_1$`(x,i)`: Returns the number of **1s** to the left of `i`, and including, position `i` in `x`.

For example, if a bit-string `x`= **1001101001** of length `n`=10, then: `SELECT`$_1$`(x, 5)=10`, `RANK`$_1$`(x, 5)=3`. `SELECT`$_0$ and `RANK`$_0$ are defined analogously for the **0** bits in the bit-string.

Naive Representation

To support `RANK`$_1$ by a naive representation would be to explicitly store the count of **1s** at each position in the bit-string in an array of length n, with space usage $nlgn$ bits. As mentioned above, `RANK`$_0$ would be automatically supported. To support `SELECT`$_1$ we could explicitly store the position of each **1** in the bit-string in an array of length n_1, where n_1 is the count of **1s**; therefore the space usage is $n_1 lgn$ bits. Supporting `SELECT`$_0$ is analogous, but applied to the **0s**. Supporting `RANK` and `SELECT` together requires $2nlgn$ bits(upper bound).

Succinct Representation

The following is known about a bit-string of length n:

Theorem 2.1:([33], Chapter 37) *There are bit-vector data structures* that use *n+O(n)* bits to support `SELECT` and `RANK` operations in *O(1)* time.

2.4.2 Bit-String Access

In many cases we will have to store a sequence of integers and encoded them by Variable Length Encoding (VLE). A Bit-String access is a data structure that stores a bit-string of length n in an integer array of size $\lceil n/32 \rceil$, and supports the following operations:

- `subBitString(i,j):` extracts the substring from positions `i` to `j` from the bit-string. We assume that the extracted `subBitString` fits into a single word, i.e. `j-i+1`<=`32`.
- `getAlignedword(i):` this returns the substring from position `i` to `i+31` from the bit-string.

The code is carefully optimised since these operations are used frequently. For example, considering 32-bit integers, we need to compute $\lfloor n/32 \rfloor$ and `i mod` 32, to determine the integer containing the ith bit, and the offset of the ith bit within the integer. The former is computed using shifts, and the latter by AND with a pre-computed mask. The main reason for separating the functions `subBitString` and getAlignedWord is that the former requires a branch statement to separate the cases where the `substring` is all in one word and where the `substring` is split across two words, and the latter does not. Since branch mis-predictions are quite expensive, the latter should be faster. In addition, the former also needs to perform division and modulo operations on two indices, while the latter does this only on one index, and has very simple code [7]. We used in VLE the `subBitString` operation, more details in 5,Section 5.3.

2.4.3 Succinct Prefix Sum

The set of objects here is a sequence *x=(x_1, .., x_n)* of positive integers that add up to m. It assumed that the algorithm knows n and m. It can be shown that there are $l = \binom{m-1}{n-1}$ such objects in the set. For example, for `n=2` and `m=4` we have $\frac{3!}{1!2!}= 3$ sequences: (1,3), (3,1) and (2,2). Taking the logarithm base two of l, and using the inequality $\binom{m}{n} \leq (\frac{me}{n})^n$ [34] we obtain:

Proposition 2.2.([7],Chapter4) *The information-theoretic lower bound for representing a sequence of* n *positive integers that add up to* m *is* $\lceil lgl \rceil \leq nlg(\frac{m}{n}) + nlge$ *bits.*

Operations

The object to be represented is a sequence of positive integers *x=(x_1, .., x_n)* where $\sum_{i=1}^{n} x_i = m$. The operation to support is as follows:

- $SUM(x, j)$: Returns $\sum_{i=1}^{j} x_i$

For example, if *x= 2, 1, 5, 6* then $SUM(x, 3) = 8$

Naive Representation

To support the SUM operation by a naive representation would be to explicitly store each prefix SUM value, requiring $n \lceil lgm \rceil$ bits.

Succinct Representation

Theorem 2.2. [35] *A sequence x with $|x| = n$ and $\sum_{i=1}^{n} x_i = m$ can be represented in $nlg(m/n) + O(n)$ bits so that $SUM(x, i)$ can be computed in $O(1)$ time.*

The performance bounds were achieved by the following data structure. Let y_i= $SUM(x, i)$ for *i=1,..,n*. Let u be an integer, $1 \leq u < lgm$:

- We use a bit-string R of length *n(lgm-u)* bits, which stores the *lower-order* *lgm-u* bits of each y_i value concatenated together.
- We use a bit-string P of length n+2^n bits. The multi-set of values formed by the *top-order* u bits is represented by coding the multiplicity of each of the values *0,..,*2^u*-1* in unary using 0s, with the 1s as separators. The unary values are concatenated together (P has n 0s and 2^u 1s)

We select $u = \lfloor lgn \rfloor$, so $|P| = O(n)$. We augment this bit-string with additional bits to support $SELECT_0$ (using an implementation from Section 4.2.1). $SUM(x, i)$ is computed as follows: we first retrieve the *lower-order* bits represented in R by the substring starting at pointer $z = (j - 1) \times (lgm - u) + 1$ and ending at pointer $y = j(lgm - u)$. The *top-order* bits are retrieved by computing $SELECT_0$-j on P. The lower and upper order values are concatenated to give y_i , which is returned in *O(1)* time.

The Prefix-Sum class contains a constructor that takes the array of string lengths in addition to the number of strings. For example: if an array of strings s_1, s_2,..s_n is stored in one single character array, called `A`. Then if we store the length of strings into an array of integers as ls_1, ls_2,ls_n the Prefix-Sum class will take this array of length with n strings. In order to access `i-th` string Prefix-Sum and by the *SUM* methods will compute $j = SUM(i)$ then we will access the `i-th` string by return a pointer to `A[j]`.

2.5 Introduction to Libbzip2

There are different compression methods, and we have used some of them in this thesis in order to store textual data in a compressed way and with less decompression time, such as a compression library of Bzip2 [36] data format called libBzip2 with different block sizes. Before we describe the library of Bzip2, we will discuss the compression algorithm called Burrows-Wheeler transform (BWT)(block-sorting compression) [37] which is used by Bzip2 to compresses files. Let S=s_1,s_2,..,s_k be a string of characters. *BWT(S)* is computed as follows:

1. Create $k \times k$ matrix A:
    - i-th row of A is the string S rotated by *i-1*positions.
2. Sort rows of A in lexicographic order and call the sorted matrix A'. Suppose the original string S is now row number i of A'. Output this number i.
3. Output, row-by-row, the symbols in the last column ofA'.

For example:

Input: `good,_jolly_good`

```
g o o d , _ j o l l y _ g o o d <- original
o o d , _ j o l l y _ g o o d g
o d , _ j o l l y _ g o o d g o <- shift by 2
d , _ j o l l y _ g o o d g o o
, _ j o l l y _ g o o d g o o d
_ j o l l y _ g o o d g o o d ,
j o l l y _ g o o d g o o d , _
```

```
o l l y _ g o o d g o o d , _ j
l l y _ g o o d g o o d , _ j o
l y _ g o o d g o o d , _ j o l <- shift by 9
y _ g o o d g o o d , _ j o l l
_ g o o d g o o d , _ j o l l y
g o o d g o o d , _ j o l l y _
o o d g o o d , _ j o l l y _ g
o d g o o d , _ j o l l y _ g o
d g o o d , _ j o l l y _ g o o

_ g o o d g o o d , _ j o l l y
_ j o l l y _ g o o d g o o d ,
, _ j o l l y _ g o o d g o o d
d , _ j o l l y _ g o o d g o o
d g o o d , _ j o l l y _ g o o
g o o d , _ j o l l y _ g o o d <- row number 6
g o o d g o o d , _ j o l l y _
j o l l y _ g o o d g o o d , _
l l y _ g o o d g o o d , _ j o
l y _ g o o d g o o d , _ j o l
o d , _ j o l l y _ g o o d g o
o d g o o d , _ j o l l y _ g o
o l l y _ g o o d g o o d , _ j
o o d , _ j o l l y _ g o o d g
o o d g o o d , _ j o l l y _ g
y _ g o o d g o o d , _ j o l l
```

Output: `y,dood__oloojggl`

The purpose of the BWT is to group characters according to their context, or the characters that immediately follow it. Knowledge of the context of a character helps compression: for example, if we know the context of a character **hat..**, then we know that the given character is very likely to be one of **w**, **W**, **t** or **T**, rather than any of the other possible ASCII characters. To isolate the different contexts, BWT is often followed by a move-to-front (MTF) transform. The output of the MTF will have small integers within a given context, but will have larger integers when moving across contexts, and provided we have enough occurrences of any given context, the output will consist mainly of small integers which are highly compressible. The software BZIP2 [36] implements the above procedure, but since sorting all rotations of a given input file is computationally expensive, BZIP2 usually breaks up an input file into blocks of a fixed size and compresses each block individually. The smaller the block size, generally speaking, the faster the compression, but since in small blocks there will be fewer occurrences of any context, the compression ratio will be lower. As a compromise BZIP2 uses block sizes of 64KB as default. In addition to a command-line compressor (bzip2), there is also a compression library libbizip2, which is more useful to us. We now describe some of its features.

BZ2_bzBuffToBuffCompress

```
int BZ2_bzBuffToBuffCompress(char* dest, unsigned int* destLen, char* source,
unsigned int sourceLen, int blockSize100k, int verbosity, int workFactor);
```

By this method we attempt to compress the data in the source into the destination buffer. One of two values are returned as follows:

Table 2.1: libBZip2-block compression ratio(Bytes): Textual data of XML documents is arranged in document order

File	B=512	B=1024	B=2048	B=4096	B=8192
Orders	1.46	1.85	2.33	2.90	3.51
SwissProt	1.58	1.90	2.37	2.94	3.61
dblp	1.45	1.82	2.23	2.66	3.11
Treebank_e	1.57	1.84	2.09	2.27	2.40
Proteins	1.49	1.70	1.99	2.43	2.92

- The destination buffer is big enough for the data from the source and the destination buffer is set to the size of the compressed data. The compressed data is bzip2 format.
- The compressed data does not fit; the destination buffer is unchanged and returns output buffer is full

We measure the compression ratio as follows:

Compression Ratio = Size of original XML file size/Size of compressed XML file size

We run the test on different block sizes(Bytes); 512, 1024, 2048, 4096 and 8192, on our data files (Orders, SwissProt, dblp, treebank and Proteins) and report the compression ratio(Bytes) for all blocks as we can see in Table 2.1

We observe that the compression ratio by using libBZip2 with larger block sizes is generally better than smaller block sizes.

BZ2_bzBuffToBuffDecompress

```
int BZ2_bzBuffToBuffDecompress(char* dest, unsigned int* destLen, char*
source, unsigned int sourceLen, int small, int verbosity);
```

Table 2.2: libBZip2-block decompression time(*second*): Textual data of XML documents is arranged in document order

File	**B=512**	**B=1024**	**B=2048**	**B=4096**	**B=8192**
Orders	0.00004	0.00006	0.00010	0.00016	0.00028
SwissProt	0.00003	0.00006	0.00010	0.00016	0.00028
dblp	0.00004	0.00006	0.00010	0.00017	0.00030
Treebank_e	0.00003	0.00005	0.00010	0.00017	0.00033
Proteins	0.00004	0.00006	0.00011	0.00018	0.00032

By this method we attempt to decompress the data in the source into the destination buffer. One of two return values are as follows:

- The destination buffer is big enough for the data from the source and the destination buffer is set to the size of the uncompressed data. The compressed data is bzip2 format.
- The compressed data does not fit; the destination buffer is unchanged and returns output buffer is full and the source assumed to hold a bzip2 format.

We run the test on different block sizes(Bytes), 512, 1024, 2048, 4096 and 8192, on our data files (Orders, SwissProt, dblp, treebank and Proteins).

The report of the decompression time(*second*) in all blocks can be seen in Table 2.2. We observe that the libBZip2 with smaller block size is generally better than larger block size. Therefore, applications would benefit from the smaller block size because the decompression of the smaller block is faster.

Chapter 3

Previous Work

In this chapter, we discuss some XML compressors, in addition to discussing the SiXDOM implementations. In Section 3.1 we talk about the XML compression. Next, in Section 3.2 we will discuss some XML compressors with DOM-like support. In Section 3.3 we discuss some of the previous work on XML compressors that have in-memory and/or disk-based representations. In Section 3.4, also we discuss some of Query-friendly XML compressors. Finally, in Section 3.5 we describe the SiXDOM architecture and interface in addition to providing a summary in Section 3.6.

3.1 XML Compression

As we noted in Chapter 1, XML bloat increases the cost of storage/backup of XML files in PCs and servers, but it can be solved through data compression, in particular XML-specific compressors.

The XML-specific compressors are designed to represent the XML documents. Each compressor treats XML characteristics in a different way, but the aim is to provide better compression. In this Chapter we focus on in-memory representations and support-tree navigation. There are some compressors that separate the structures of XML documents for the data values, and represent the structure in-memory, leaving the data values on the disk. Good results have been produced from many of these compressors such as, ([38], [27], [39], [40], [41], [42], [43], [44], and [45]). In addition give more details on a specifec XML compresser and related to our work such as [2], [6], [5] and [26],

Usually the representations of the XML compressors are:

- Local homogeneity: represents the structure of the XML document separately to the data values which are grouped into containers such as XMill [5].
- Homomorphic: which preserves the structure of the XML document with data values such as XGrind [44].

We can measure the compression performance of XML compressors based upon standard categories of XML documents such as Data-centric documents, which focus on the regular structure of tags, and Document-centric, which focus on the text of XML documents. We can measure the compression ratio as follows:

Compression ratio=Size of compressed XML data/Size of original XML data.

We now discuss some XML-specific compressors that achieve very good compression ratios.

3.2 XML Compressors with DOM-like Support

DDOM

In [46] an XML compressor called Dictionary-compression based Document Object Model (DDOM) was implemented in Java and considered to be locally homogeneous.

In DDOM the tree structure of the XML document is represented by two arrays; the representations are in a document-order. The first array is to maintain the node type information, which is called TYPE, and the second one is to maintain an index value for each element to its name. In this compressor the Text nodes are maintained in dictionaries associated with their parent node. Compared to Xerces-J and Crimson (DOM implementation) [47], DDOM showed a good compression ratio; it was between 20% and 60% for data-centric XML files and between 70% and 80% for document-centric XML files. But DDOM does not provide a solution for XML bloat because for real life XML documents have a large space usage and is 3 to 4 times more than the size of the file.

BPLEX

BPLEX has a very compact pointer-based representation of the XML tree structure and is locally homogeneous [29]. The data values in this compressor are represented in string buffers for the leaf text nodes and for the attribute values which are associated with the element nodes. In this compressor we can make the navigation with partial decompression and the DOM interface can be supported in addition to storing the string buffers with standard techniques [38] more space-efficiently.

SEDOM

This compressor is a DOM implementation and is considered to be locally homogeneous. SEDOM [26] supports retrieval, update and XPath operations on the document and contains the following components:

- Name index: stores the unique element names.

- Framework or document structure: this represents the document tree as a one-dimensional array in document-order.
- Compressed containers: stores the data values arranged into blocks for fast retrieval and updating.
- Container block index: this represents indexing information for each block in the compressed containers.

The results for SEDOM showed that its main memory usage is less than 6.9% of the main memory usage of the pointer-based DOM implementations for the XML files that are discussed in [26]. By SEDOM we can support a DOM update with large documents, but it is much slower than a single pointer access for the pointer-based DOM such as Xerces.

3.3 XML Compressors

We now discuss one of the specialised XML compressor that achieve an excellent compression ratio [5], but does not support query processing operations such as navigation without potentially decompressing the whole document.

XMiLL

XMill [5] is a popular tool and three principles were applied during the compression:

- Separate structure (element tags and attribute names) from data and tokenise the tags and attributes. Start tags and attribute names are dictionary-encoded, such as `T1, T2,...`, etc. Replace end tags with `/` token. In order to distinguish an attribute name from an element name, the attributes are represented

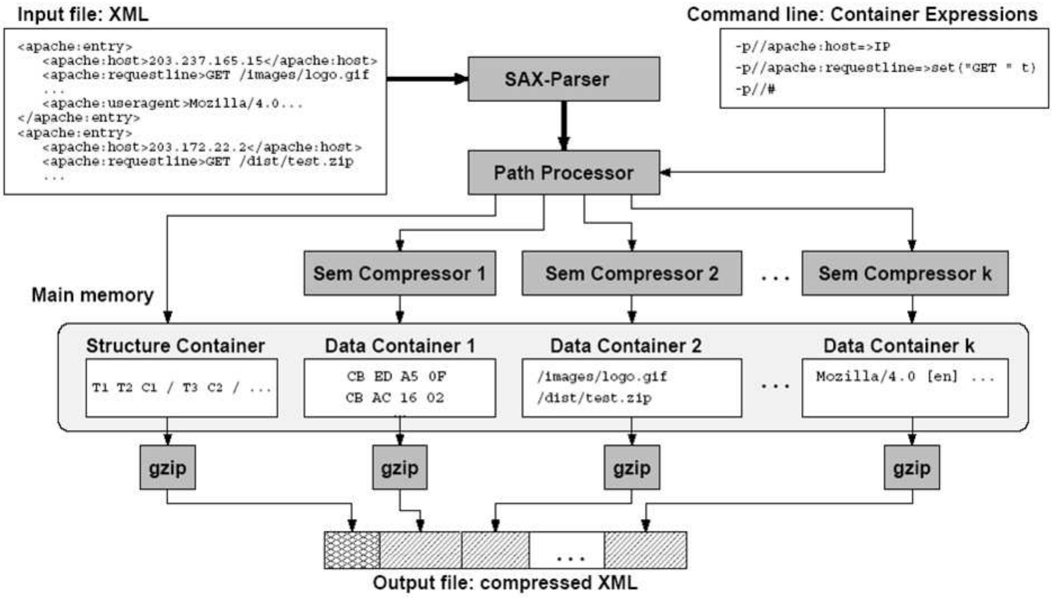


FIGURE 3.1: XMill Architecture [5]

with a prefixed symbol `@`, and the data values are replaced with their container `ID` number, as we can see in the next step.

- Group related data items into a single container; compress each container separately. The mapping strategy in this technique is to assign containers to specific paths by the data values path and by user parameters.
- Apply appropriate semantic compressors to each container. Grouping related data items with similar patterns of data into a single container led to the provision of a better compression. XMill applies a specialised compressor for each container.

Figure 3.1 shows the architecture of XMill. XMill provides a good compression performance, but the compressed data does not supporting querying.

3.4 Query-friendly XML Compressors

Some query-friendly XML compressors have recently been developed that support operations on the compressed file while requiring (at most) partial decompression such as [6]. Although some of these such as XbzipIndex [6] support DOM-like navigation, they are significantly an order of magnitude slower than standard DOM implementations.

XBzipIndex

XBzipIndex uses the XBW transform to maintain the XML document in a highly compressed format and, by uncompressing only a tiny fraction of the data, enables both navigation and searching. XBzipIndex which is focus on a query-friendly [6].

In XBzipIndex, the document is represented as a tree as in Figure 3.2, as follows: start from <`biblio`>, which is the root of the XML document, and replace all the element string like <`biblio`> by <`biblio`; add this symbol `@` at the start of the attribute name, which is stored on the tree as an element node like `@id`; create a special node starting with `=` called a skip for text and another one for the textual value starting with `Ø` character, which is considered a child for the skip node. After that, as we show in Figure 3.3, create three columns as follows: the first column for the string called Slast (if the node Sα is a last child a **1** is stored, otherwise a **0** is stored), the second for string Sα (the label for each node in the tree), and the third for Sπ (shows the upward path of nodes, starting from parent). Then the XBW transform applies a stable sort according to the string Sπ on the left part of Figure 3.3 (`Slast, S`α, Sπ) in lexicographical order to create the representation shown on the right side of the same Figure. In the bottom of Figure 3.3 we show

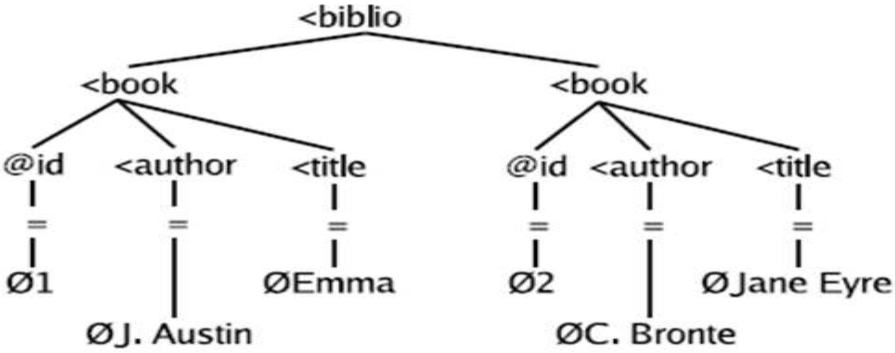
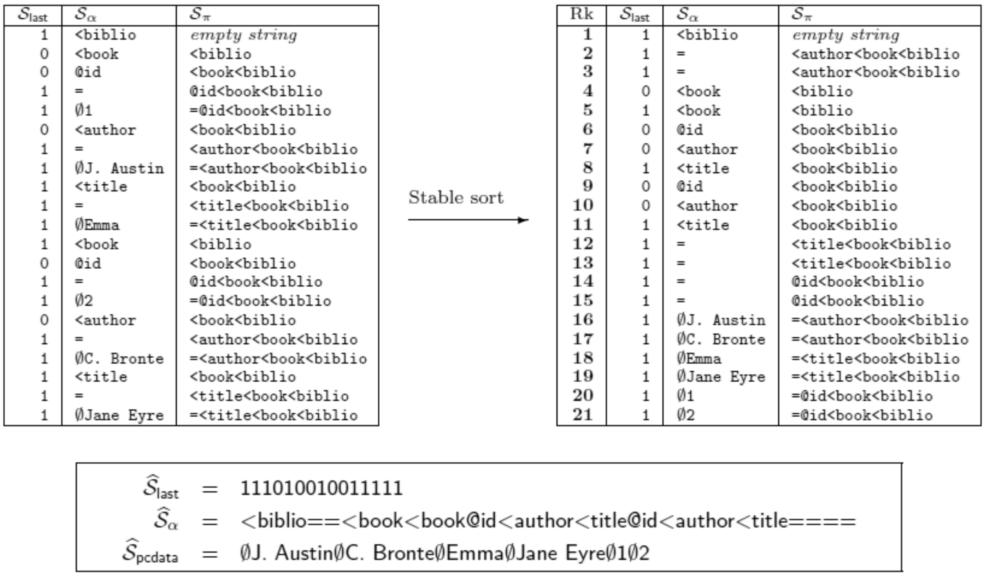


FIGURE 3.2: XML Document as a Tree [6]


Stable sort

$\mathcal{S}_{\text{last}}$	$\mathcal{S}_\alpha$	$\mathcal{S}_\pi$
1	<biblio	*empty string*
0	<book	<biblio
0	@id	<book<biblio
1	=	@id<book<biblio
1	∅1	=@id<book<biblio
0	<author	<book<biblio
1	=	<author<book<biblio
1	∅J. Austin	=<author<book<biblio
1	<title	<book<biblio
1	=	<title<book<biblio
1	∅Emma	=<title<book<biblio
1	<book	<biblio
0	@id	<book<biblio
1	=	@id<book<biblio
1	∅2	=@id<book<biblio
0	<author	<book<biblio
1	=	<author<book<biblio
1	∅C. Bronte	=<author<book<biblio
1	<title	<book<biblio
1	=	<title<book<biblio
1	∅Jane Eyre	=<title<book<biblio

Rk	$\mathcal{S}_{\text{last}}$	$\mathcal{S}_\alpha$	$\mathcal{S}_\pi$
1	1	<biblio	*empty string*
2	1	=	<author<book<biblio
3	1	=	<author<book<biblio
4	0	<book	<biblio
5	1	<book	<biblio
6	0	@id	<book<biblio
7	0	<author	<book<biblio
8	1	<title	<book<biblio
9	0	@id	<book<biblio
10	0	<author	<book<biblio
11	1	<title	<book<biblio
12	1	=	<title<book<biblio
13	1	=	<title<book<biblio
14	1	=	@id<book<biblio
15	1	=	@id<book<biblio
16	1	∅J. Austin	=<author<book<biblio
17	1	∅C. Bronte	=<author<book<biblio
18	1	∅Emma	=<title<book<biblio
19	1	∅Jane Eyre	=<title<book<biblio
20	1	∅1	=@id<book<biblio
21	1	∅2	=@id<book<biblio

$$\begin{aligned}\widehat{\mathcal{S}}_{\text{last}} &= \text{111010010011111}\\ \widehat{\mathcal{S}}_{\alpha} &= \text{<biblio==<book<book@id<author<title@id<author<title====}\\ \widehat{\mathcal{S}}_{\text{pcdata}} &= \text{∅J. Austin∅C. Bronte∅Emma∅Jane Eyre∅1∅2}\end{aligned}$$

FIGURE 3.3: XBW Transform Example [6]

the output of the XBW transform three arrays `S^last, S^`α`,` (labels of the internal nodes) and `S^pcdata` (textual values).

The authors experimentally show that XBzipIndex is better compared to other compressors and faster on some paths and content search operations, but significantly slower than standard DOM implementations when they support DOM-like navigation. In addition, the complex structure of the Xbzip representation means that efficiently supporting additional operations on the document while keeping the space usage may be challenging, as suggested by the conclusions section of [6].

Path Queries on Compressed XML

In [2] compression is provide on the XML tree structure, which represents an XML document as a directed acyclic graph (DAG) by sharing common subtrees. [2] reduces the XML tree to a minimal DAG representation and further reduction was achieved to the size of the minimal DAG representation by using multiplicity counters for consecutive equal subtrees. Further details will be provided in Chapter 4.

Vectorizing and Querying Large XML Repositories

In [48] a locally homogeneous compressor is presented for a query-friendly. As we mention in [2], this compressor compressed the XML tree structures as DAG representation and was extended to supports XQuery with partial decompression as in [2], which support only XPath. In this technique the data values was represented as containers called vectors as in XMill [5]. The experimental evaluation of the XQuery system of [48] (called VX) is better than MonetDB [27] and Galax [49] for a particular query that required constructing portions of the original XML document by almost 2.5 orders of magnitude, but MonetDB was significantly better than VX for the queried the matching of all data values.

3.5 SiXDOM Implementations

This section is based on Chapter 7 of Delpratt's thesis [7]. In this section, we will describe the SiXDOM implementation [7].

3.5.1 SiXDOM Architecture

There are four components in SiXDOM as we can see in Figure 3.4, starting from the DOM `document` node that contains a pointer to each component in SiXDOM components as follows:

- `STree`, which is the succinct tree data structure (DS).
- `Namecode` DS stores the XML names for the nodes in the document.
- `Text` DS handles the textual data in the document.
- `Attribute` DS handles the attribute nodes in the document and their associations to the element nodes.

A compressed representation to the textual data was provided by SiXDOM-CT after replacing the Text DS component with a text data structure; we will discuss this in more detail in Chapter 6.

We now discuss each component in more detail, to see which operations they support in each component and how they support those operations.

STree & Node Object

STree & Node Object support the below operations:

- `parent()`
- `childNodes()`
- `firstChild()`
- `hasChildNodes()`

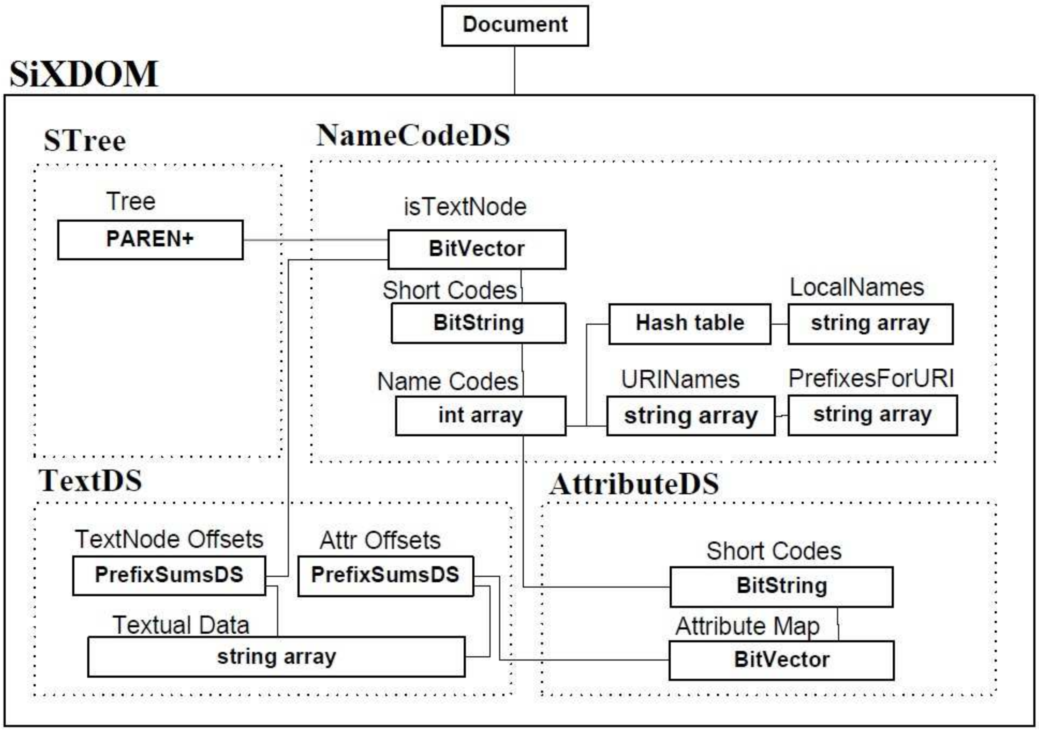


FIGURE 3.4: DOM architecture. SiXDOM is stored in the Document node. SiXDOM components are shown with dotted boxes. Connecting lines show relationships between data structures [7]

- `lastChild()`
- `compareDocumentPosition()`
- `nextSibling()`
- `previousSibling()`

In addition, it supports the following:

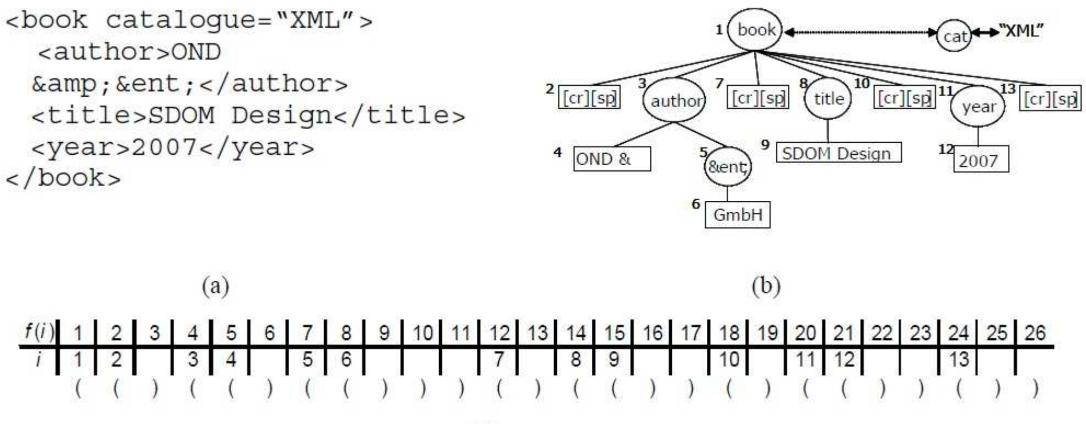
- The `TreeWalker` interface. Here we have the same navigation operations as in the `Node` interface, in addition to the `nextNode`(), `previousNode`() and `currentNode`() operations.

- The `item`() and `length`() operations in the `NodeList` helper interface are available to the DOM.

- The `following`, `preceding`, `descendant` and `ancestor` in XPath.

SiXDOM avoided using pointers to represent the tree node objects and used PAREN+ (more details in [7], Chapter) representation as the tree structure, and used a double number to represent the node: the node number i^{th} in document order (from `1` to `n`) and its position $\varphi(i)$ in the succinct tree bit-string representation (from `1` to n), where n is the number of nodes in the tree. We can recall that $i = RANK_0(\varphi(i))$, if we represent `"("` by **`0`** and `")"` by **`1`**. Each node object contains the integers and a reference to the containing `document` node. We can navigate by first accessing in the node the pointer to the document node, and accessing the `PAREN+` object, then calling the navigation operations of the underlying succinct tree representation, which in turn gives the answer as a double number, which is then wrapped in a node object. SiXDOM creates a node object as we need this to make a navigational operation on the existing node object, unlike a pointer-based DOM implementation. When the XML document is parsed, SiXDOM does not create all the node objects in a document, becuase creates a node object whenever a navigational operation is invoked on an existing `node` object.

In order to represent an internal node, we need 192 bits (it assumed 64 bit machine) to represent the double number and the document node pointer, because we access all the SiXDOM internal components by document node object, unlike a DOM implementation (such as Xerces), which requires several more pointers to represent an internal node in particular. Through the navigational operations we can navigate the tree representation and the C++ object must be explicitly freed. Using the `TreeWalker` class is an alternative way to avoid the creation of node objects.



```
<book catalogue=“XML”>
  <author>OND
 &amp;&ent;</author>
 <title>SDOM Design</title>
 <year>2007</year>
</book>
```

FIGURE 3.5: (a): Simple XML document fragment. (b): Corresponding DOM tree representation. (c) Parentheses representation of the tree structure with double numbering of nodes. [7], [8]

At this stage, SiXDOM focuses only on the structure of the DOM tree and ignores the storage of the node type information. In Figure 3.5 we can see the parentheses sequence of the XML document in (a) `element` nodes are identified in circles and text nodes in boxes in (b) and the parentheses string in (c). The double numbering is encapsulated in a `node` class object to represent the nodes. For example, the $8-th$ node (the element ”title”) is at the 14^{th} position in the bit-string. The entity &ent; represents the text ”GmbH”.

SiXDOM defined two operations on the parentheses representation to improve PAREN+ with the speedup of the primitive operation to go from a node to the next/previous node in document order as follows:

- `NEXTOPEN()`: returns $(i+1,\varphi(i+1))$ if $i<n$ and NULL otherwise, where $x=<i,\varphi(i)>$ (return the position and RANK of the next opening parenthesis given that we are at the opening parenthesis at position x in the bit-string)

- `PREVIOUSOPEN()`: analogous.

NameCode Data Structure

NameCode Data Structure is used to store the name and type of information of each node in the DOM tree and primarily supports the below operations of the DOM `Node` interface:

- `getNodeName()`
- `getNodeType()`
- `getTagName()`
- `hasChildNodes(`
- `getPrefix()`
- `lookupPrefix()`
- `getLocalName()`
- `getPrefix()`

In addition, it supports the following:

- The operations `getElementByTagName`()and `getElementByTagNameNS`() in the `Document` interface.
- The operation `getTagName`()in the `Element` interface.

In order to support the operations above, SiXDOM provided a solution with three parts: `isTextNode` bit-vector, the `Namepool` and the `shortCode` data structure.

Namepool Data Structure

There is a `name-code table` to store each unique element name in an array as 64 bits. We then split the nodes into text nodes and number them from `1..t` in document order, and the non-text tree nodes from 1.. `e` (mostly `element` nodes, but including `comment` nodes, `entityReference` nodes etc.), where `t` and `e` are the number of text nodes and non-text tree nodes, respectively (note `t+e=n`); then we can compress effectively. SiXDOM used the `isTextNode` bit-vector for the splitting, as below.

IsTextNode Bit-Vector

The `isTextNode` is defined as follows: if the ith node in document order is a text node then the ith bit is set to `1`, otherwise it is set to `0`, and we can use the `RANK` operation into the `isTextNode` bit-vector to provide consecutive numbering of text nodes from `1` to `t` and of non-text tree nodes from `1` to `e`.

Short-Code Data Structure

A short-code is a positive integer. An array of size `e` created to fill with short-code for the ith non-text tree node in document order, it was interpreted as follows:

- If the ith short-code is 12 or less, then the ith node is not an element node, and the short-code value gives its node type. The possible node types and their values are: (`CDataSection` (4), `entityRef` (5), `processingInstruction` (7), `comment` (8) or `docType` (10) and `Entity` (6), `Notation` (12)). We notice that: (*Element* (1), `Attributes` (2), `Text` (3), and `Document` (9)).

- If the ith short-code j is 13 or greater, then the node is an `element` node, and $j-13$ is an index into the name-code table, pointing to the entry in this table corresponding to the `i-th` element name.

Textual Data Structure

Textual Data Structure is used to store and retrieve the textual data of individual nodes or groups of nodes within the XML document. Primarily this component supports the DOM operations of the Node interface given below:

- `getNodeValue()`
- `getTextContent()`

In addition, it supports the following:

- The `getElementByID()` method of the `Document` interface.
- The `getValue()` method of the `Attribute` interface.
- The `getData()` method of the `ProcessingInstruction` interface.

In SiXDOM store the textual data of the XML document into a single C++ array. The textual data for the following node types are stored:

- `Text:` data value associated with the text node.
- `Attributes:`attribute node value.
- `Processing Instruction:`data component of the processing instruction.
- `Comment:`content of the comment node.

- `CDATASection:`content of the CDATA Section.

More detail about the textual data structure will be provided in Chapter 6.

Attribute Data Structure

Attribute Data Structure provides mapping of attribute nodes to the element nodes in the DOM tree by store the name information of each attribute node and the associated node value. Mainly this component supports the DOM operations of the `NamedNodeMap` and the `Attribute` interfaces given below:

- `item()`
- `getName()`
- `length()`
- `getOwnerElement()`
- `getNamedItem()`
- `isID()`
- `getNamedItemNS()`

In addition, it supports the following:

- The `getElementById()`and `getElementByTagNameNS()` methods of the `Document` interface.
- The `getAttributes()`and `hasAttributes()` methods of the `Node` interface.

- The `getAttribute()`, `getAttributeNS()`, `getAttributeNode()`, `getAttributeNodeNS()` and `hasAttribute()` methods of the `Element` interface.
- The `getTarget()` method of the `ProcessingInstruction` interface.
- The `getName()` method of the `DocType` interface.
- The `getNotationName()` method of the `Entity` interface.

Separately from the tree representation, SiXDOM represents `attribute` nodes, and provides a mapping strategy to map `elements` to their `attributes`, and `attribute` names to their values.

As in `NameCode Data Structure`, use the `isTextNode` bit-vector number's non-text tree nodes from 1 to e. We then created a sequence of non-negative integers $X = (x_1, .., x_e)$ of length e as follows. If the i^{th} non-text tree node is an `element` node, then x_i is the count of attributes it has. If the i^{th} non-text tree node is any of `processingInstruction`, `CDataSection`, `docType`, `document` or `comment` nodes, give it a dummy attribute with $x_i = 1$. Represent X to satisfy the below goals:

- All attributes should be numbered from 1 to a (where a is the total number of attributes, including dummy attributes) and the attributes associated with a given non-text tree node should be numbered consecutively.
- Given a non-text tree node, it should be possible to determine quickly the range of integers that number its (dummy) attributes, if any.

3.5.2 SiXDOM Interface

Class Structure

SiXDOM is an application that was designed to support DOM and is compatible with XSLT/XQuery processors. There is an intermediate interface that calls the succinct data structures directly, which in turn is called by the DOM operations. Similar to the interface that is used in Saxon [50], the intermediate interface has the `NodeInfo` and `DocumentInfo` interfaces and we can access the `TinyTree` data structure directly (`TinyTree` is the internal tree structure in Saxon). SiXDOM application also in C++ supported a ported version of the `NodeInfo` and `DocumentInfo` interfaces to be a plug-in replacement for `TinyTree` The `TinyNodeImpl` (which implements the `NodeInfo`) was replaced with SiXDOM's Node class. In order to represent the node object directly, the `Node` class consists of two integers for the node and a pointer to the `Document` node, with an additional layer implementing the `NodeInfo`. We can access the SiXDOM data structure by the Document node. Some of the DOM operations directly match those in the `NodeInfo`; for example in the `NodeInfo` we can retrieve the node type information of a node by `getNodeKind()`, which has the same function as the DOM operation `getNodeType()`. In SiXDOM's `NameCode` data structure we can directly retrieve the name-code of a node by the `getNameCode()` operation, which uses this operation by the DOM operation `getNodeName()`, where we find in the hash table the matching node name to the name-code.

SiXDOM provides direct support for the DOM node navigation operations, In addition it supports the `iterateAxis` operations of `NodeInfo`, except the `namespace` axes.

3.6 Summary

In SiXDOM, the DOM is implemented using succinct data structures. As an example, SiXDOM avoids using pointers between nodes in the XML document tree by storing a balanced parenthesis string of 2n bits to encode the tree structure, and using a succinct index to perform rapid navigation in the tree. A standard representation of the document tree would use at least three pointers (parent, first child and next sibling), so the savings can be considerable. Speeds comparable to standard DOM implementations, together with memory usage (typically a fraction of the XML file size), are achieved by SiXDOM [1] by using succinct data structures [33].

However, SiXDOM did not use any kind of compression in representations. For example, a highly regular tree with `n` nodes and a randomly generated tree with `n` nodes would both take `2n` bits to be represented. XML trees, however, are generally quite regular and therefore should be compressible.

Chapter 4

In-Memory Representation Based upon MacMill

In this chapter we examine the XML compression method MacMill [2] in detail. MacMill is a compression method that represents the XML document by extracting the attributes and textual data, leaving a skeleton. The skeleton is then compressed into a directed acyclic graph (DAG) by sharing common sub-trees. In this chapter we outline our approach to representing XML documents in-memory in a highly compressed format, while supporting operations of the kind supported by DOM. Our approach is based upon the MacMill compression method.

This chapter is organised as follows. In Section 4.1 we explain MacMill. Then we explain how to support DOM-like navigation using MacMill in Section 4.2. Next, in Section 4.3 we give some details of the algorithms and implementation. In Section 4.4 we present experimental results and analyse them. Finally, we give a summary in Section 4.5.

4.1 MacMill

MacMill describes the structure of XML documents by storing the tree whose nodes are labelled with the names of the elements only.

4.1.1 Overview

Given an XML document, if we remove attributes and textual data, we are left with just the structure and the names of the elements. We call this document the *Virtual Skeleton (VS)*[1]. Figure 4.1(a) shows an example of an XML document and Figure 4.1(b) shows the VS. Buneman *et al.* describe how to compress the VS by representing each distinct sub-tree only once in order to create a minimal DAG.

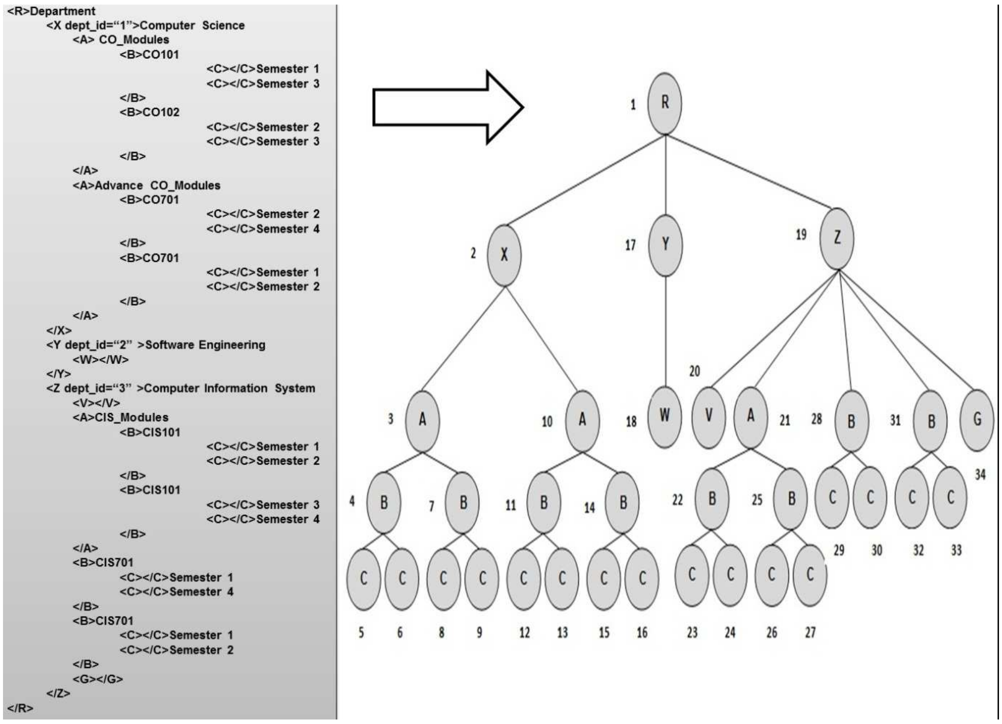
On the left of Figure 4.2 we show the minimal DAG for the above VS of Figure 4.1(b) obtained by sharing common sub-trees. For example, nodes X and Z in the VS have identical sub-trees, so we merged these sub-trees into one sub-tree. Another observation is that on the left of Figure 4.2 there are multiple out-edges from the same node such as nodes X, A or Z, etc., and on the right of Figure 4.2 the consecutive multiple out-edges to the same node are collapsed into one edge, and the number of collapsed edges are added as additional information to the node the we called the *Compressed Skeleton(CS).*

4.1.2 MacMill Output Format

Buneman *et al.* created the software package MacMill that implements the above compression method. In this section we will give more details about the output of

---

[1] Virtual skeleton is used in a similar context to skeleton: document-tree structures whose nodes are labelled with elements only



```
<R>Department
        <X dept_id="1">Computer  Science
             <A> CO_Modules
                      <B>CO101
                                       <C></C>Semester 1
                                       <C></C>Semester 3
                      </B>
                      <B>CO102
                                       <C></C>Semester 2
                                       <C></C>Semester 3
                      </B>
             </A>
             <A>Advance  CO_Modules
                      <B>CO701
                                       <C></C>Semester 2
                                       <C></C>Semester 4
                      </B>
                      <B>CO701
                                       <C></C>Semester 1
                                       <C></C>Semester 2
                      </B>
             </A>
        </X>
        <Y dept_id="2" >Software Engineering
             <W></W>
        </Y>
        <Z dept_id="3" >Computer Information  System
             <V></V>
             <A>CIS_Modules
                      <B>CIS101
                                       <C></C>Semester 1
                                       <C></C>Semester 2
                      </B>
                      <B>CIS101
                                       <C></C>Semester 3
                                       <C></C>Semester 4
                      </B>
             </A>
             <B>CIS701
                      <C></C>Semester 1
                      <C></C>Semester 4
             </B>
             <B>CIS701
                      <C></C>Semester 1
                      <C></C>Semester 2
             </B>
             <G></G>
        </Z>
</R>
```

FIGURE 4.1: (a) XML Document Shown on the Left. (b) Virtual Skeleton on the Right

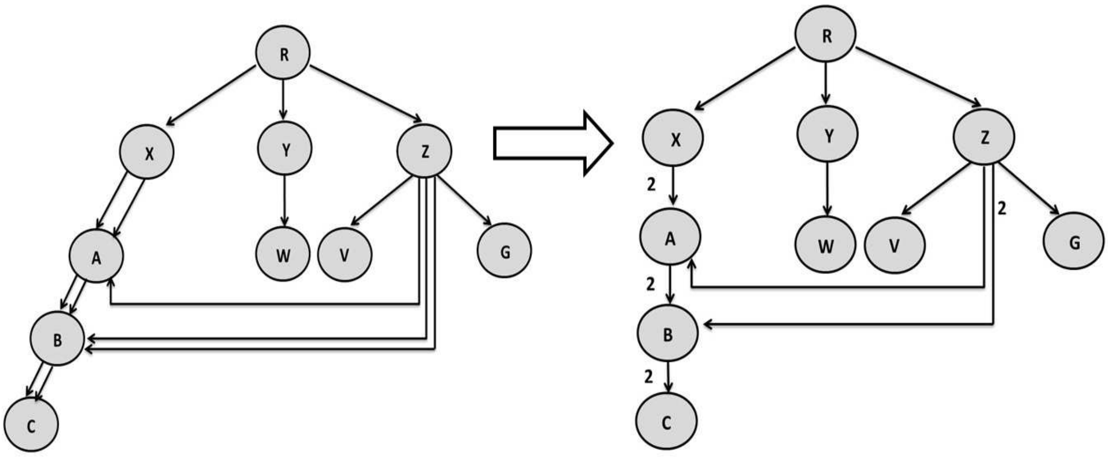
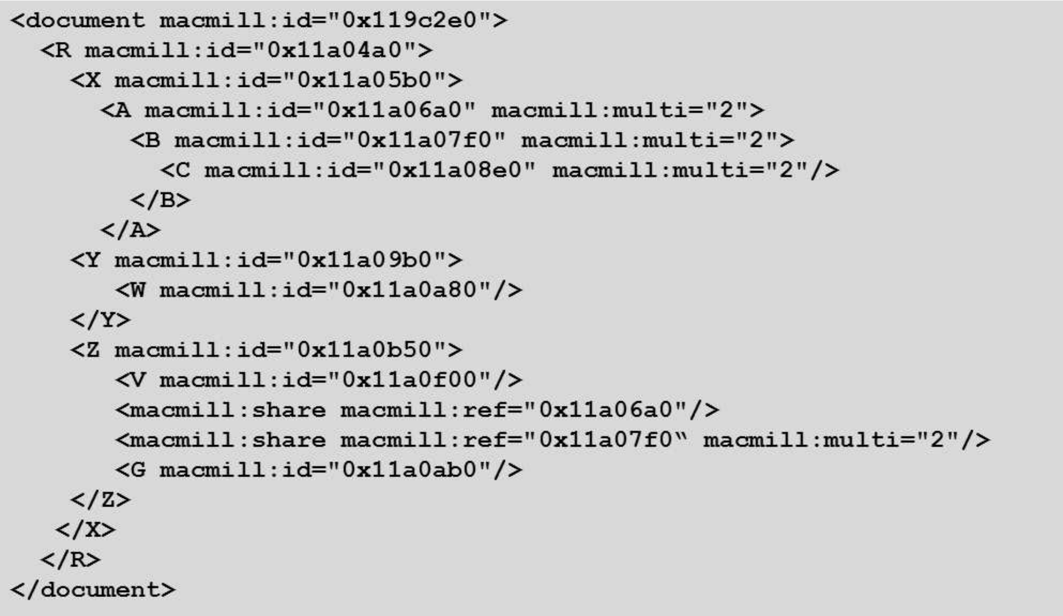


FIGURE 4.2: Minimal DAG of Virtual Skeleton



```
<document macmill:id="0x119c2e0">
  <R macmill:id="0x11a04a0">
    <X macmill:id="0x11a05b0">
      <A macmill:id="0x11a06a0" macmill:multi="2">
        <B macmill:id="0x11a07f0" macmill:multi="2">
          <C macmill:id="0x11a08e0" macmill:multi="2"/>
        </B>
      </A>
    <Y macmill:id="0x11a09b0">
       <W macmill:id="0x11a0a80"/>
    </Y>
    <Z macmill:id="0x11a0b50">
       <V macmill:id="0x11a0f00"/>
       <macmill:share macmill:ref="0x11a06a0"/>
       <macmill:share macmill:ref="0x11a07f0" macmill:multi="2"/>
       <G macmill:id="0x11a0ab0"/>
    </Z>
   </X>
  </R>
</document>
```

Figure 4.3: XML file output by MacMill

MacMill. In Figure 4.3 we show the output of MacMill for the XML document in Figure 4.1(a).

MacMill produces the CS in the form of an XML file. The output of MacMill contains an additional tag which is called `document` and considers the root of the CS. Every element in the output of the MacMill file is one of two kinds: a *normal* or a *shared* node.

- Normal node (with the same element name as in the VS) contains attribute named `macmill:id`; the value for this attribute for all nodes is a unique string.
- Shared node that contains attributes named `macmill:ref`; the value for these attributes for all nodes is string refer to an `ID` (i.e. normal node) [2]

Every node in the VS is represented by a node in the CS called *representative*. All the representative nodes are normal nodes. The share nodes refer to the representative node by the first attribute value of these nodes. All nodes can have a second attribute called `macmill:multi`, which contains a string value that shows how many out-edges there are to the same node, as we have shown in the right of Figure 4.2.

The output of MacMill is shown as a DOM tree in Figure 4.4. We notice that the differences between the right of Figure 4.2 and Figure 4.4 as follows: there is a shared sub-tree in the right of Figure 4.2 which is represented as a node in Figure 4.4 called `ms` (`macmill:share`), and refer to normal node by share it attribute value. The other difference is to add another attribute for any node has multi-out edges.

For example, in Figure 4.2 we can see the sub-trees with root A repeated two times as a child of node X and one time as a child of node Z, and the sub-tree with root B repeated two times as a child of node Z. In the CS in Figure 4.4, the first two sub-trees with parent A and children of X are represented in the left part as three nodes A[multi= 2], B[multi= 2], and C[multi= 2] with one edge from X; then the third sub-tree with root A which is a child of Z is represented as a single node called ms with one out-edge from Z and points to the A sub-tree in the left part with a dashed line. The two sub-trees with parent B which are child of Z are represented as a node called ms [multi =2] with one out-edge from Z and are pointing to the B sub-tree in the left part with a dashed line.

The advantage of using XML as the output format is that it includes more information than the others; it often provides information about or gives meaning to the text it contains. In addition, we can read the XML file by using any standard DOM parser, including Xerces and SiXML, which are available for all popular programming languages.

We now evaluate the performance of MacMilll as a pure compressor. Table 4.1 shows the performance of MacMill in comparison to the VS. We notice from Table 4.1 that MacMill provides an excellent compression ratio (reducing the skeleton size after making the compression) on highly compressible data files, in particular if the file

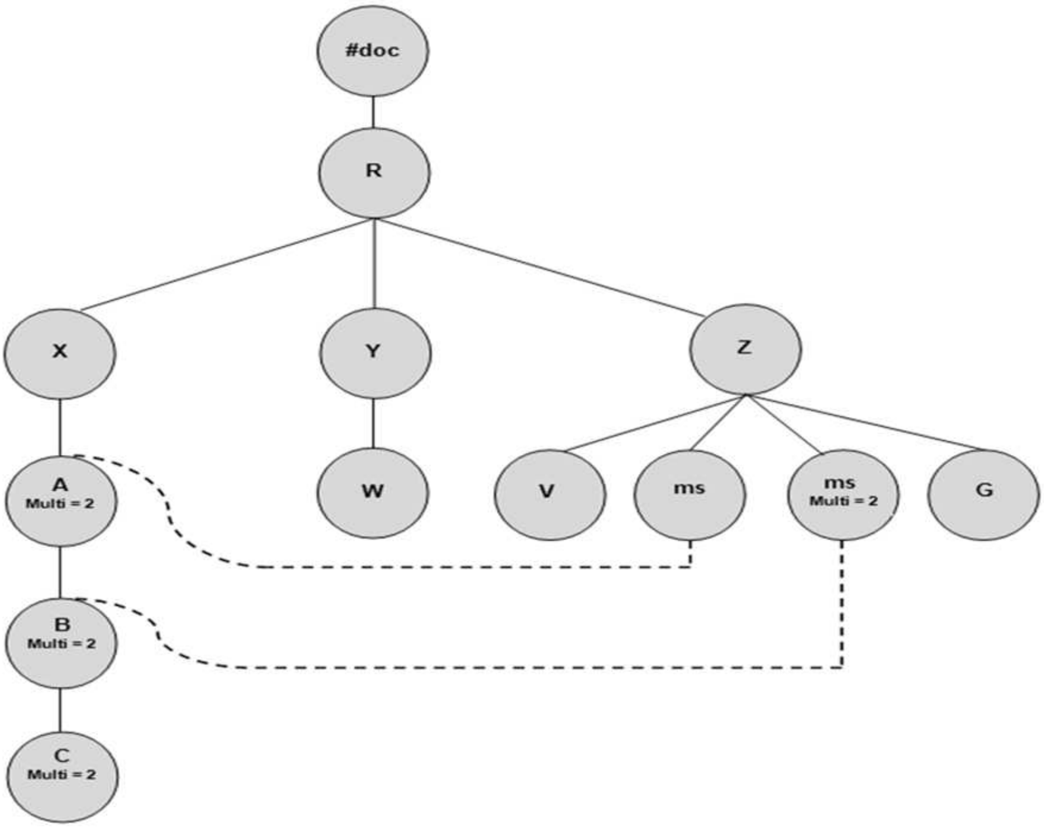
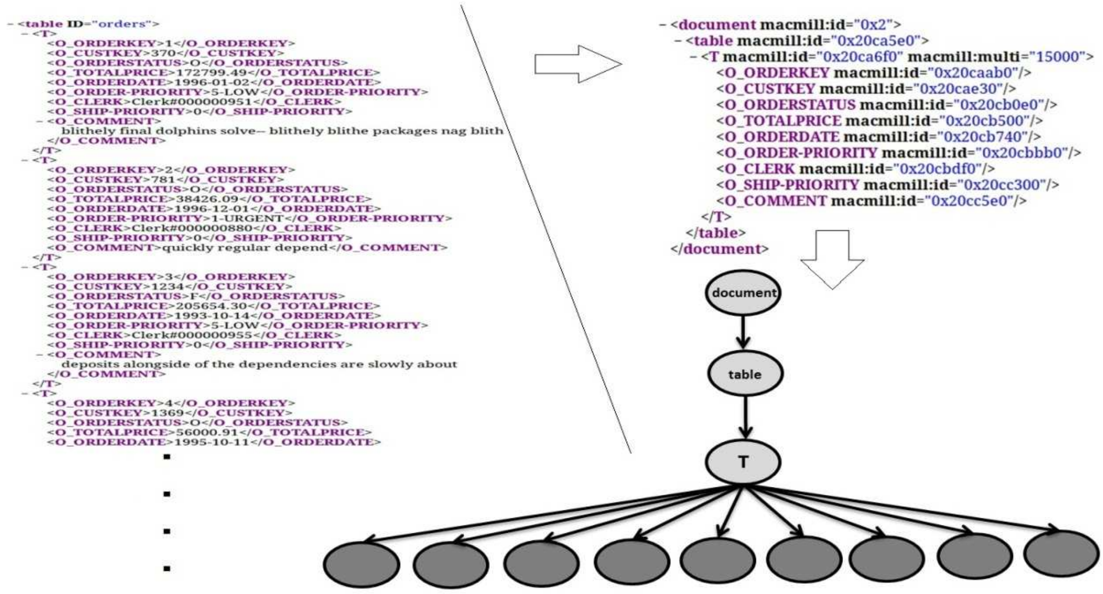


Figure 4.4: Compressed Skeleton

has many common sub-trees such as `orders.xml`. The VS size for `order.xml` is 3.7 MB; using MacMill reduced the size to 600 Bytes and compressed very well as we show in Figure 4.5.

However, if the file has few common sub-trees this approach does not give a good result, e.g., consider the file `treebank_e.xml`; we can clearly see the VS file is much smaller than the CS. The output of MacMill is very verbose because a significant amount of markup is added in the compressed file output, which explains why the skeleton is much larger than the original file if not many sub-trees are shared. Another observation from Table 4.1 is the number of nodes in VS to the number of



```
- <table ID="orders">
  - <T>
      <O_ORDERKEY>1</O_ORDERKEY>
      <O_CUSTKEY>370</O_CUSTKEY>
      <O_ORDERSTATUS>O</O_ORDERSTATUS>
      <O_TOTALPRICE>172799.49</O_TOTALPRICE>
      <O_ORDERDATE>1996-01-02</O_ORDERDATE>
      <O_ORDER-PRIORITY>5-LOW</O_ORDER-PRIORITY>
      <O_CLERK>Clerk#000000951</O_CLERK>
      <O_SHIP-PRIORITY>0</O_SHIP-PRIORITY>
    - <O_COMMENT>
        blithely final dolphins solve-- blithely blithe packages nag blith
      </O_COMMENT>
    </T>
  - <T>
      <O_ORDERKEY>2</O_ORDERKEY>
      <O_CUSTKEY>781</O_CUSTKEY>
      <O_ORDERSTATUS>O</O_ORDERSTATUS>
      <O_TOTALPRICE>38426.09</O_TOTALPRICE>
      <O_ORDERDATE>1996-12-01</O_ORDERDATE>
      <O_ORDER-PRIORITY>1-URGENT</O_ORDER-PRIORITY>
      <O_CLERK>Clerk#000000880</O_CLERK>
      <O_SHIP-PRIORITY>0</O_SHIP-PRIORITY>
      <O_COMMENT>quickly regular depend</O_COMMENT>
    </T>
  - <T>
      <O_ORDERKEY>3</O_ORDERKEY>
      <O_CUSTKEY>1234</O_CUSTKEY>
      <O_ORDERSTATUS>F</O_ORDERSTATUS>
      <O_TOTALPRICE>205654.30</O_TOTALPRICE>
      <O_ORDERDATE>1993-10-14</O_ORDERDATE>
      <O_ORDER-PRIORITY>5-LOW</O_ORDER-PRIORITY>
      <O_CLERK>Clerk#000000955</O_CLERK>
      <O_SHIP-PRIORITY>0</O_SHIP-PRIORITY>
    - <O_COMMENT>
        deposits alongside of the dependencies are slowly about
      </O_COMMENT>
    </T>
  - <T>
      <O_ORDERKEY>4</O_ORDERKEY>
      <O_CUSTKEY>1369</O_CUSTKEY>
      <O_ORDERSTATUS>O</O_ORDERSTATUS>
      <O_TOTALPRICE>56000.91</O_TOTALPRICE>
      <O_ORDERDATE>1995-10-11</O_ORDERDATE>
```

```
- <document macmill:id="0x2">
  - <table macmill:id="0x20ca5e0">
    - <T macmill:id="0x20ca6f0" macmill:multi="15000">
        <O_ORDERKEY macmill:id="0x20caab0"/>
        <O_CUSTKEY macmill:id="0x20cae30"/>
        <O_ORDERSTATUS macmill:id="0x20cb0e0"/>
        <O_TOTALPRICE macmill:id="0x20cb500"/>
        <O_ORDERDATE macmill:id="0x20cb740"/>
        <O_ORDER-PRIORITY macmill:id="0x20cbbb0"/>
        <O_CLERK macmill:id="0x20cbdf0"/>
        <O_SHIP-PRIORITY macmill:id="0x20cc300"/>
        <O_COMMENT macmill:id="0x20cc5e0"/>
      </T>
    </table>
  </document>
```

FIGURE 4.5: Part of Order.xml and the Output of MacMill on the Right

TABLE 4.1: The Performance of MacMill Compression

XML File	**Original (MB)**	**Virtual Skeleton(MB)**	**Compressed Skeleton(MB)**	**Number of Nodes in VS**	**Number of Nodes in CS**
Order	5.1	3.7	0.0006	150002	11
SwissProt	109.5	43.2	41.0	2977032	777871
Treebank_e	82.1	24.7	80.5	2437667	1295124
Dblp	127.7	53.7	13.7	3332131	279499
Proteins	600	348.2	43.0	21305819	864461
Factor38.4	4608	1126.4	179.6	64152027	3505541

nodes in CS; we can see that, if MacMill failed to reduce the number of nodes, then we will suffer from these nodes because they have a lot of information.

4.2 DOM with MacMill

We now explain how to support DOM-like navigation on the CS such as for basic navigation operations: (`getFirstChild()`, `getLastChild()`, `getNextSibling()`, `getPreviousChild()` and `getParent()`). We need additional requirements in order to support DOM-like navigation before using MacMill directly. In this section we will consider the main obstacles below:

- As noted in Section 4.1.2, each node in the VS is represented by a normal node in the CS. However, a single normal node in the CS may represent many nodes in the VS. Therefore we need to distinguish between the different nodes.
- Nodes may have multiple parents. When trying to find the `parent()` in the VS, we need to know which of the possible representatives in the CS are the true parent.
- There are two kinds of pointers: one is the DAG `idref` to `id` pointer, and the other is pointers as in C++; because if we meet a share node we need to take the attribute value (`ref`) and start a search in the XML file for a normal node that has the same attribute value (`id`).

Our Solution

The first consideration is what kind of DOM-like access we can support using the CS. It seems that the most convenient approach is to use navigation similar to the `TreeWalker` DOM interface.

As we support a `TreeWalker` interface, our approach will be as follows:

- For each node in the CS, we will store its sub-tree size. Therefore, start from the root and for every node in the CS, it computes the size of the sub-tree under the node in the VS.

For example if we look at Figure 4.1(b) in Section 4.1.1 and want to find the next sibling of node V on the VS but look to navigate on the compressed tree, as is shown in Figure 4.4, this operation will be difficult because on the CS (unlike the VS) we find the previous sibling is a node called ms, which points to the node A without keeping the node number in the virtual skeleton (DFS numbering) as a factor in the navigation.

- The `TreeWalker` will contain a stack for going up in the tree to find the parent.

For example, from Figure 4.1(b) if we are looking to find the parent of node A21, it will be very easy to find that it is node Z, but from Figure 4.4 it will be difficult to find the parent because maybe the node in the CS has more than one parent, such as the parent of node A is node X or node Z.

- Hash tables. We build a hash table so that for any node we can retrieve the corresponding sub-tree size (this was not present in [2]). We need to build a hash table for the `macmill:id` in order to find the representative for the share node.

In theory, we can support node `ids` on the VS. That is, given a node `id` in the VS (e.g. a position in document order) we can generate a `TreeWalker` for that node reasonably efficiently. Inside TreeWalker there is a node object which contains not only node `id`, but also a number from `1` to `2n` (where is n is the number of nodes in VS). More details in Chapter 5, section 5.1.

4.3 Algorithms and Implementation

We now give some details of the algorithms for navigation in the tree. We assume that the CS is stored in a DOM representation that supports node `ids` (e.g. each node in the CS can be referred to by an `id`). As noted above, we have two hash tables: one, called `stsize`, maps each `id` of a node in the CS to its sub-tree size. The second, called `idref`, maps the value of each `macmill:ref` attribute in the CS to the `id` of its representative node in the CS.

The `TreeWalker` as noted above will contain a stack. If the treewalker is at some node v in the VS, then the top of the stack in the `TreeWalker` will point to the representative of v in the CS. In addition, it will contain:

- `CurrentPosition`: the node number in VS document order.
- `CopyNumber`: the order number of a node between all the nodes with the same parent.
- `OfNumber`: the order of out-edges for a node.

From Figure 4.6, we can see A3 is the first copy of A (`CurrentPosition=3 and CopyNumber=1`), A10 is the second copy of A (`CurrentPosition=10` and `CopyNumber=2`) and A has a two copies of B (`OfNumber=2`)

The item below in the stack points to the representative in the CS of the parent of v, together with all the above auxiliary information, and so on (the bottom of the stack contains the root of the CS).

We now show how to support the operations: `getFirstChild()`, `getLastChild()`, `getNextSibling()` `getPreviousSibling()` and `getParent()`.

v is the node current position of the TW in the VS

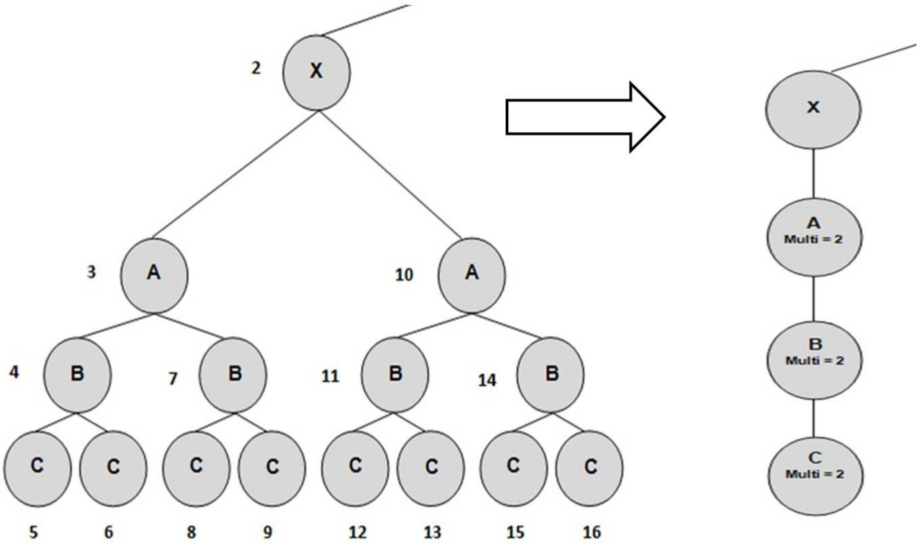


Figure 4.6: Part of VS and CS

getFirstchild(a)

```
Let the top of the stack point to node a
Let b be the first child of a in the CS
IF a has no first child in the CS THEN // Using DOM FirstChild()
    v has no First Child
    Return NULL
IF b is a share node THEN
    b = idref[b]
    Create new stack node, make it point to b
    Set the CurrentPosition to CurrentPosition+1
    Set the CopyNumber to 1
```

```
        IF no Mullti Attribute THEN
            Set the OfNumber to 1
            ELSE
            Set the OfNumber to the value of Multi
    Set push on top of stack
ELSE // if b is a Normal node NOT share
    Create new stack node, make it point to b
    Set the CurrentPosition to CurrentPosition+1
    Set the CopyNumber to 1
        IF no Mullti Attribute THEN
            Set the OfNumber to 1
        ELSE
            Set the OfNumber to the value of Multi
    Set push on top of stack
Return b
```

getLastChild(a)

```
Let the top of the stack point to node a
Let b be the last child of a in the CS
```

```
IF a has no last child in the CS THEN // Using DOM getLastChild()
    v has no Last Child
Return NULL
IF b is a share node THEN
    b = idref[b]
    Create new stack node, make it point to b
    Set the CurrentPosition to CurrentPosition + (Subtree-size of b - Subtree-size of a )
        IF no Multi Attribute THEN
            Set the OfNumber to 1
            Set the CopyNumber to 1
        ELSE
            Set the OfNumber to the value of Multi
            Set the CopyNumber to the value of Multi
    Set push on top of stack
ELSE // if b is a Normal node NOT share
    Create new stack node, make it point to b
    Set the CurrentPosition to CurrentPosition + (Stsize [ b] - Stsize[a])
        IF no Multi Attribute THEN
```

```
                Set the OfNumber to 1
                Set the CopyNumber to 1
            ELSE
                Set the OfNumber to the value of Multi
                Set the CopyNumber to the value of Multi
Return b
```

getNextSibling(a)

```
Let the top of the stack point to node a
Let b be the next sibling of a in the CS
IF a has no next sibling in the CS THEN // Using DOM NextSibling()
    v has no next sibling
    Return NULL
IF CopyNumber of a less than OfNumber THEN
    Set CopyNumber of b to 1
    IF b is a share node THEN
        share node then b = idref[b]
        Set CurrentPosition = CurrentPosition +Stsize[b]
    ELSE
        Set CurrentPosition = CurrentPosition +Stsize[b]
        Return b
ELSE // b is NOT Normal
    b is a share node
    b = idref[b]
    Let c be the next sibling of b in the CS
    IF b has no next sibling in the CS THEN
        v has no next sibling
        Return NULL
    Move the stack node, make it point to c
    Set CurrentPosition = CurrentPosition +Stsize[b]
    Set CopyNumber to 1
    IF no Mullti Attribute THEN
        Set the OfNumber to 1
    ELSE
        Set the OfNumber to the value of Multi
    Return b
```

getPreviousSibling(a)

```
Let the top of the stack point to node a  Let b be the previous sibling  IF a has
no next sibling in the CS THEN // Using DOM getPreviousSibling()
```

```
    v has no previous sibling      Return NULL
IF CopyNumber of a -1 less than 1 THEN
    v has no previous sibling
    Return NULL
    IF b is a share node THEN
        b = idref[b]
        Set the CurrentPosition to CurrentPosition - Stsize [ a]
            IF no Multi Attribute THEN
            Set the OfNumber to 1
            Set the CopyNumber to 1
        ELSE
            Set the OfNumber to the value of Multi
            Set the CopyNumber to the value of Multi
    ELSE // b is Normal NOT share
         Set the CurrentPosition to CurrentPosition - Stsize [ a]
        Set the CopyNumber to the value of CopyNumber -1
        Set the OfNumber to the value of OfNumber
ELSE // b is share NOT normal
    b is share node
    b = idref[b]
    Let c be the previous sibling of b in the CS
    IF b has no previous sibling in the CS
        v has no previous sibling
        Return NULL
    Move the stack node, make it point to c
    Set the CurrentPosition to CurrentPosition - Stsize [a]
    Set the CopyNumber to the value of CopyNumber -1
    Set the OfNumber to the value of OfNumber
Return b
```

getParent(b)

```
    Let the top of the stack point to node a
    Let the below of the stack point to node b
    IF b has no ancestor in the CS THEN
        Return NULL
    ELSE
        Set a the ancestor of b
        Return a
```

In Figure 4.7 we illustrate multiple uses of navigation operations on the compressed skeleton of Figure 4.4 when using DAGDOM, if we are looking to find the last child for the node Z, with current position number= 19 on the VS. This operation, after applying the `getLastChild()` of DAGDOM, will find the last child is G with current position= 34. This is a simple operation because node Z pointed directly to node G in the CS (the same as the VS). However, if we are looking to find the previous sibling for node G, this operation will be difficult because in the CS we find the previous sibling is a node called ms with multi = 2, and a pointer to the node B, without considering the current position as a factor in navigation. Nonetheless, after applying `getPreviousSibling()` as in DAGDOM we will find the B with current position = 31.

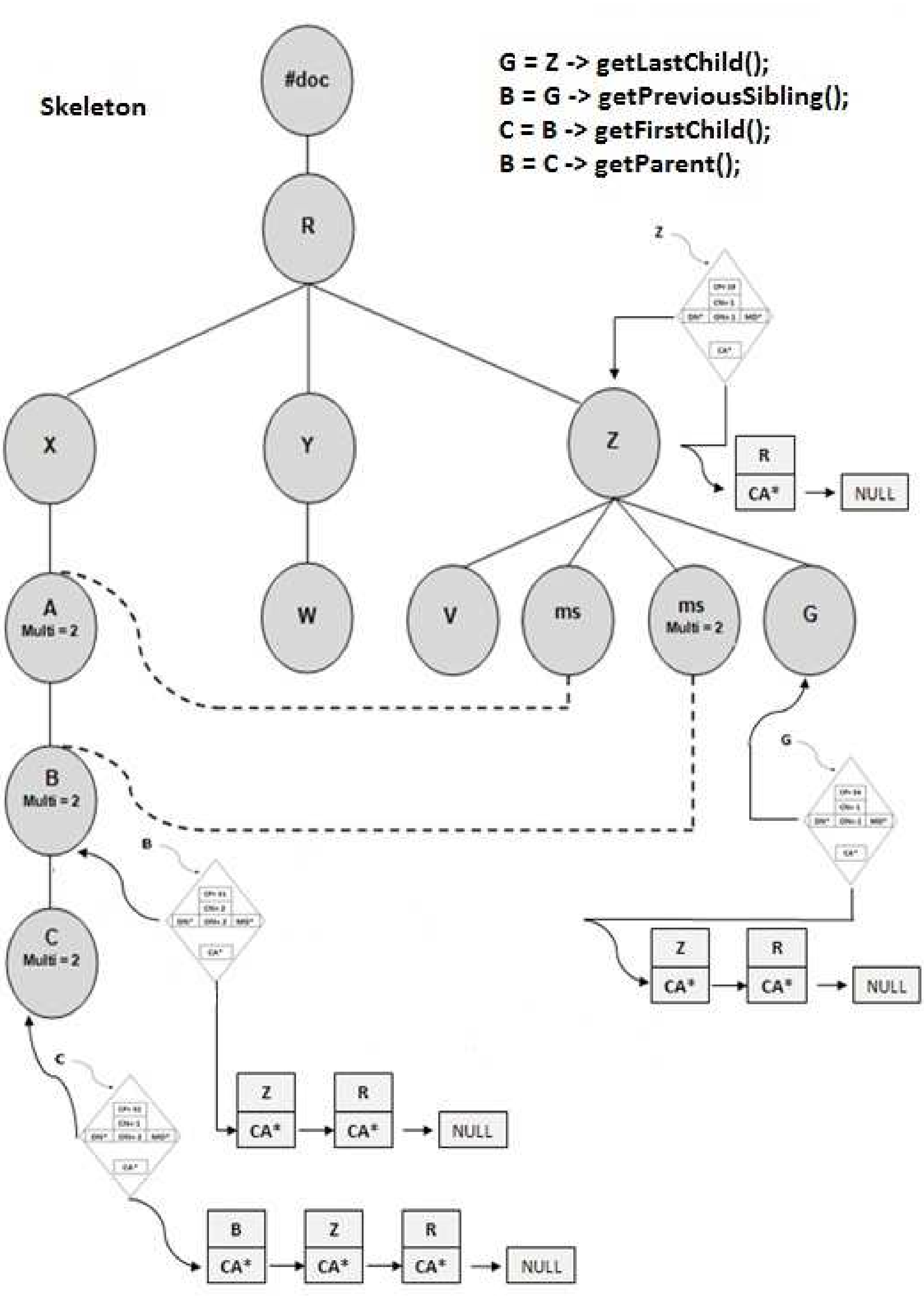
Implementation

In this section we will describe some implementation details. We have a C++ `DAGDOM` class. The data members of this class are:

- `p_skel` of type `DOMNode` // pointer to skeleton.
- `stsize`, `reftable` are hash tables.

In the DAGDOM class, there are the following functions:

- `DAGDOM()`: this is the constructor that takes the output of the MacMill file then parses this file by using `Xerces` and set `p_skel` to the `root` of the parsed document(CS).
- `DAGDOM_initialize()`: takes the root of the CS and, for every node in the CS, it computes the size of the sub-tree under that node in the VS. Furthermore,



```
G = Z -> getLastChild();
B = G -> getPreviousSibling();
C = B -> getFirstChild();
B = C -> getParent();
```

Figure 4.7: Sequence of Multiple Navigation Operations

it sets up a hash table so that for any node, its sub-tree size can be retrieved, and for any CS pointer, the target of the pointer can be found.

- `getRoot()`: returns a pointer to the root of the VS, which is represented by DAGDOM object after initialising the data members of `DAGDOM_Node` class, as in the pseudocode below.

The pre-processing time is not measured in the thesis. There was no detailed optimisation of this.

getRoot()

```
Let v be the root of VS

Create new stack node, make it point to p_skel

Set the CurrentPosition = 1

Set the CopyNumber = 1

Set the OfNumber = 1

Set the ancestor of stack node to NULL

Return v
```

```
  Let x be the root of CS
```

DAGDOM_ initialize()

```
  IF Share node THEN

      Set subtreesize = stsize[ref] // ref is the attribute value of share node
which refers to the attribute value of normal node
```

```
    Return Multi* subtreesize
  ELSE
    Set reftable[id]=x // id is the attribute value of normal node
    Set subtreesize=1
    FOR x has a firstchild
      x has a nextsibling
      subtreesize = subtreesize+ recursive call DAGDOM_initialize()
    ENDFOR
```

4.4 Experimental Evaluation

In this section we draw comparisons of the space usage between Xerces and DAGDOM. We tested DAGDOM and Xerces on five XML files taken from the XML corpus [9]. Our choice of files gives us a range of typical XML documents.

Our tests obtained document-order traversals `(FirstChiled, NextSibling, and Parent)`. In [1] obtained document-order traversals and reverse document-order traversals `(LastChild, PreviousSibling, and Parent)` and the result approximately shows no differences between both tests. Therefore, in our tests we considered document-order traversals `(FirstChiled, NextSibling, and Parent)`.

Before we started our test, we put the Xerces and DAGDOM into the same environment to be fair in our test. So this was done by the following:

- In DOM (Xerces), we take the XML file such as `orders.xml` file and create a new file called `VSorders.xml` considering the element node only and ignoring the other components of the XML document (attribute, text node, etc).
- In DAGDOM we take the XML file such as `orders.xml` file and by using MacMill software we create a compressed file called `Morders.xml` considering the element nodes only and ignoring the other components of the XML document (attribute, text node, etc).

We note that the virtual skeleton of XML file such as, VSorders.xml or the compressed skeleton such as Morders.xml represented in main memory using Xerces.

4.4.1 Basic Setup

The test machine was an Intel(R) Pentium(R) with 8GB RAM, 2.80GHzCPU and a 3MB cache size, running Ubuntu 10.04.4 LTS. The compiler was g++.

4.4.2 Main Memory Usage

In Table 4.2 we draw comparisons of the space usage (MB) between Xerces, DAGDOM (Using Xerces with MacMill), and SiXDOM. If we look at an XML file such as `orders.xml` we can see that DAGDOM shows much better memory usage than Xerces, because in Xerces there is no ability to use the compressed file in the representation. But, the VS of `orders.xml` with size 3.7MB used more memory usage in-memory representation.

On the other hand, for example in `Treebank_e.xml`, we can notice that the MacMill software did not give a good compression ratio. Therefore, DAGDOM uses up to double the space usage of Xerces.

Table 4.2: Main Memory Usage of XML representations in Xerces, DAGDOM and SiXDOM(MB). The File Sizes in MB. * The space usage is too low

File Name	**Files Size**	**Xerces**	**DAGDOM**	**SiXDOM**
Orders	5.1	28.62	*	4.5
SwissProt	109.5	553.38	540	33.8
Treebank_e	82.1	452.8	949.9	31.1
dblp	127.7	614.86	185.1	39.2
proteins	600	3920.93	563.9	86
factor38.4	4608	11775.89	2245.7	221

Table 4.3: Compression Ratio (CR) in CS to VS and Main Memory Usage (MMU) in DGADOM to Xerces

XML File	**VS_Size (MB)**	**CS_Size (MB)**	**CR(CS:VS)**	**MMU**
Order	3.7	0.0006	6466.2:1	28620:1
SwissProt	43.2	41	1.1	1
Treebank_e	24.7	80.5	0.3	0.5
dblp	53.7	13.7	3.9	3.3
proteins	348.2	43	8.1	7
factor38.4	1126.4	179.6	6.3	5.2

The experimental results shows that SiXDOM has the ability to reduce memory usage more efficiently than DAGDOM, even though we used the same VS files as we used in Xerces without any compression.

Therefore, in the next chapter we will make further optimisations of the information stored in the CS to improve the memory usage of DAGDOM.

In Table 4.3, another observation is that if we look at the results of all our data that, in our approach using the same underlying DOM implementation, we can see the compression ratio very useful (which means that if we compress the XML file on disk, then main memory usage of the XML representation was reduced). As a result, the space usage is improved when we have more compression ratio.

Table 4.4: Running Time of XML Representations in Xerces, DAGDOM and SiXDOM (*seconds*)

File Name	**File Size (MB)**	**Xerces**	**DAGDOM**	**SiXDOM**
Orders	5.1	0.01	0.19	0.01
SwissProt	109.5	0.11	6.70	0.21
Treebank_e	82.1	0.16	6.03	0.28
dblp	127.7	0.12	7.35	0.20
proteins	600.0	0.83	44.85	1.97
factor38.4	4608.0	360.00	143.52	5.70

4.4.3 Running Time

In Table 4.4 we draw comparisons of running time (wall-clock time in *seconds*) between Xerces, DAGDOM (Using Xerces with MacMill), and SiXDOM.

Table 4.4 shows the traversal time for the basick navigation operations (`FirstChild, NextSibling, and Parent`) in a *second.* The result shows 20 times slowdown in CPU time for navigational operations in comparison with SiXDOM. Compared to Xerces implementation we notice in some data files like `proteins.xml` are 50 times slower, but on the large files such as `factor38.4.xml`, Xerces is two times slower.

4.5 Summary

Our approach yields a good result in comparison to Xerces but it is still outperformed by SiXDOM. If we look at Table 4.3 in Section 4.4.2, we can see the compression ratio between CS and VS. If we look at the results in all data we will see that, in our approach using the same underlying Xerces DOM implementation, the space usage is improved. But we lose to SiXDOM space usage and CPU time for navigational operations, so we are investigating further optimisations of the information stored in the CS to improve the memory usage of our approach. Further optimisation to the DAGDOM naive implementation by careful application of succinct data structures will be detaild in Chapter 5.

Chapter 5

In-Memory Representation of the XML Document Using DAGDOM

As we saw in Chapter 4 a naive implementation of our approach (DAGDOM) gives very poor space usage and CPU time for navigational operations relative to other space-efficient DOM implementations (e.g. SiXDOM) [1]. DAGDOM uses the Xerces DOM implementation to represent the CS in-memory, but DAGDOM does not work very well because the compression method [2] may result in a bad compression ratio on some XML files and, as a result, in cases such as `treebank_e.xml` the memory usage could be double even that of Xerces, and 30 times more than SiXDOM.

An obvious optimisation is to replace the use of Xerces to represent the CS in DAGDOM with SiXDOM. In Table 5.1 we compare the space usage just of loading the output of MacMill into SiXDOM (without creating the hash tables: `idref` and `stsize`) versus the space usage of VS in SiXDOM. In Table 5.1 we can see the main memory usage representation for the VS files by using SiXDOM in column 3 and the main memory usage of the output of MacMill by using SiXDOM as well in

Table 5.1: In-Memory Representation of MacMill and Virtual Skeleton by SiXDOM. File Sizes in MB

XML file	**VS File Size**	**SiXDOM-VS (MB)**	**MacMill File Size**	**SiXDOM-MacMill (MB)**
Orders	3.7	4.5	0.0006	0.6
SwissProt	43.2	33.8	41	54.3
Treebank_e	24.7	31.1	80.5	148.1
Dblp	53.7	39.2	13.7	19.9
Proteins	348.2	86.0	43	61.4

column 5. The result shows the in-memory representation of some VS files is better than MacMill files.

At this stage, the main memory usage when we use SiXDOM to represent the skeleton is better than previous results of our approach, but this is not very promising. We notice that SiXDOM does not compress the attributes, and that each element in the output of the MacMill file has at least one attribute. Table 5.1 shows that simply substituting Xerces by SiXDOM in DAGDOM still gives very poor space usage.

In this chapter we further develop our approach in order to obtain significant space reductions over existing space-efficient Xerces DOM implementations as follows:

- **Auxiliary Attribute Removal Phase**: as we noted in Chapter 4, the output of MacMill is very verbose, and one particular problem is the use of auxiliary attributes. In Section 5.1, we consider efficient storage of the information contained in these auxiliary attributes which makes use of some special properties of SiXDOM. The resulting approach, called **DAGDOM A**, is then experimentally evaluated.

- **Share Node Removal Phase**: to further improve the space usage, we have made some of statistics test to the output of MacMill data files; we had some observations (such as, share nodes with sub-tree size equal to **1**) which have led to modifying the output of MacMill, in addition to representing the auxiliary information in a compressible way. We show that by post-processing the output of MacMill appropriately, we can get significant space improvement. This is discussed in Section 5.2. The resulting approach, called **DAGDOM B**, is then experimentally evaluated.
- **Variable Length Encoding Phase**: We have made further optimisations to representations of auxiliary information by using variable length encoding in order to obtain more space reduction. This is discussed in Section 5.3. The resulting approach, called **DAGDOM C**, is then experimentally evaluated.

Finally, we analyse the experimental results of the above phases and show the improvements of the space usage and running time in Section 5.4. We note that DAGDOM A-C are incremental enhancements.

5.1 Auxiliary Attribute Removal Phase

We now look at the reasons why loading MacMill files into SiXDOM yields poor results. As noted in Chapter 4, the output of Macmill is verbose. In particular, each node in the MacMill output has either one or two attributes as follows:

- `macmill:id` attribute: each normal node in the CS has an attribute called `macmill:id`. The value of this attribute appears to be a memory address in hexadecimal, on a 32-bit machine; the attribute value is a string with up to 10 characters (`0x` followed by the address with 8 hexadecimal digits).

- `macmill:ref` attribute: each share node in the CS has an attribute called `macmill:ref`. The value of this attribute appears to be a memory address in hexadecimal and refers to a particular normal node in the CS.
- `macmill:multi attribute:` most of the element nodes in the CS's contain a second attribute value which is called `macmill:multi`.

We note that SiXDOM does not store attribute values in compressed form. This can be a significant overhead. For example, `treebank_e.xml` has a total of 1.25 million `macmill:id` or `macmill:ref` attributes (see Table 5.3). Even just storing these attributes will require up to 25MB of space. Thus, the auxiliary attribute values added by MacMill can be very space-consuming. We now discuss how to remove these attributes.

MacMill:id Attributes

We need this attribute in order to support navigation on the CS. If we have a share node, we look at value on `macmill:ref` as the first step. We then search for an equal value on `macmill:id` (from normal nodes) as a second step to retrieve the element node.

However, SiXDOM effectively numbers nodes in document order from `1..n` where `n` is the number of nodes in the tree, and does not create node objects upon loading the document. A `Node` object containing DOM represents the nodes by a double number: document order number and its position in the succinct tree bit-string representation (from `1` to `2n`) [51], [1]. Figure 5.1 shows an example for SiXDOM node tree representation. Thus, we can remove `macmill:id` attributes.

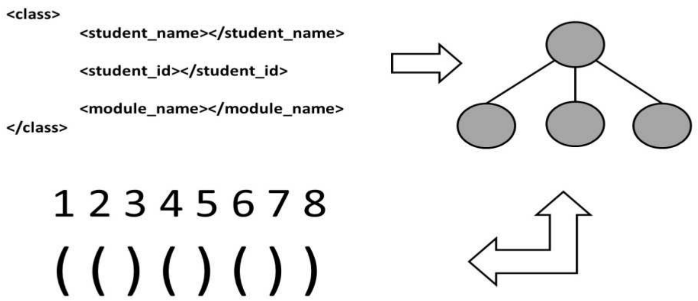


FIGURE 5.1: SiXDOM Node Tree Representation. For example, the double number of the <student_id> element node is (3, 4) since it is the third node in document order and the corresponding open parenthesis is in position 4.

MacMill:ref Attributes

If a share node points to another node, we just replace it by its node number as we mentioned above. However, accessing a node by document-order number is not in the DOM API. Therefore, an additional non-standard operation `GetNodebyNumber(x)` is implemented in SiXDOM by Stelios Joannou, in order to obtain a Node object from just the document-order number of the node. Using `GetNodebyNumber(x)` we are able to strip all `macmill:ref` attributes from the CS as we mentioned above

In the *Auxiliary Attributes Removal Phase* we will read the MacMill file and traverse this file in document-order. For every node in the CS, stripping `macmill:id,` `macmill:ref` and `macmill:multi` attributes creates a new shrunken XML file (shrink skeleton), and at the same time sets up three auxiliary text files as follows:

- *Id_ref.txt:* store `-1` if a normal node, else the node number (document-order)that the share node refers to (after following `macmill:ref`).
- *Multi.txt:* store the multi value (from `macmill:multi`).
- *Stsize.txt:* store sub-tree size for all nodes.

In Figure 5.2 we show the auxiliary attribute removal phase and the new data input format.

As we have shown in Figure 5.2, we will represent in main memory the three auxiliary text files into three arrays of 32-bit integers and implement a new algorithm to use SiXDOM components instead of the Xerces DOM implementation. Then the navigation operations will be as in the previous version of our approach: `getFirstChild()`, `getLastChild()`, `getNextSibling()`, `getPreviousChild()`, `getParent()` call this *DAGDOM A*.

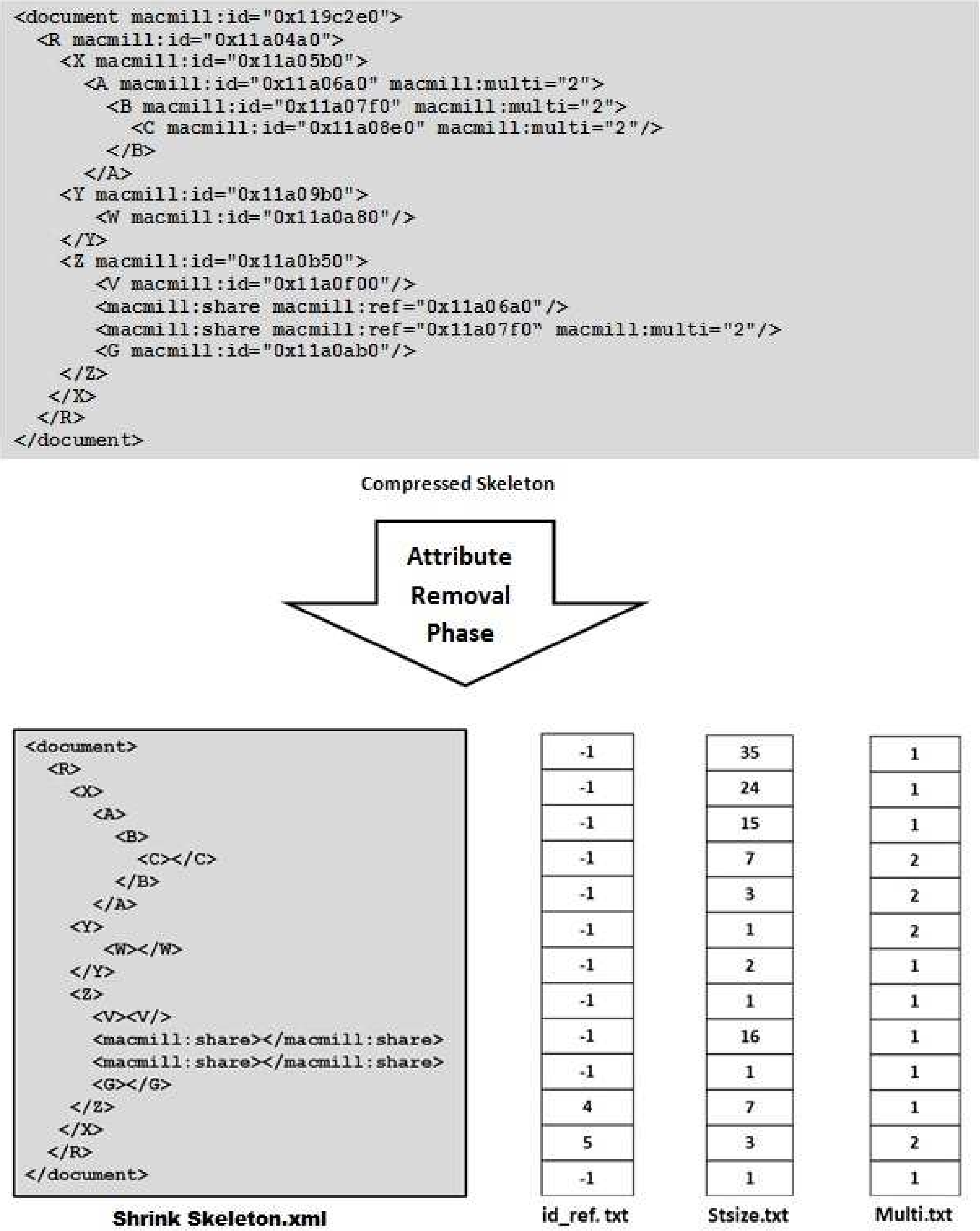
In Table 5.2 we show the pseudocode for `NextSibling()` as an example and to compare the same operation in DAGDOM with the new version, DAGDOM A.

Performance

The performance of in-memory representation of DAGDOM A in comparison to SiXDOM is shown in Figure 5.3. We notice from Figure 5.3 that DAGDOM A compared to SiXDOM reduced the main memory usage in most data files, but if we look at `treebank_e.xml` we can see that SiXDOM is still better than DAGDOM A but not by much. In Figure 5.4 we draw comparisons of running time (wall_clock time in *seconds*) between DAGDOM A and SiXDOM, which shows DAGDOM A is slower than SiXDOM, but much faster than DAGDOM.

5.2 Share Node Reduction Phase

DAGDOM A currently yields good results compared to Xerces but SiXDOM still provides better space usage in some files such as `treebank_e.xml`. Therefore we



```
<document macmill:id="0x119c2e0">
  <R macmill:id="0x11a04a0">
    <X macmill:id="0x11a05b0">
      <A macmill:id="0x11a06a0" macmill:multi="2">
        <B macmill:id="0x11a07f0" macmill:multi="2">
          <C macmill:id="0x11a08e0" macmill:multi="2"/>
        </B>
      </A>
    <Y macmill:id="0x11a09b0">
       <W macmill:id="0x11a0a80"/>
    </Y>
    <Z macmill:id="0x11a0b50">
       <V macmill:id="0x11a0f00"/>
       <macmill:share macmill:ref="0x11a06a0"/>
       <macmill:share macmill:ref="0x11a07f0" macmill:multi="2"/>
       <G macmill:id="0x11a0ab0"/>
    </Z>
   </X>
  </R>
</document>
```

```
<document>
  <R>
    <X>
      <A>
        <B>
          <C></C>
        </B>
      </A>
    <Y>
       <W></W>
    </Y>
    <Z>
      <V><V/>
      <macmill:share></macmill:share>
      <macmill:share></macmill:share>
      <G></G>
    </Z>
   </X>
  </R>
</document>
```

id_ref. txt	Stsize.txt	Multi.txt
-1	35	1
-1	24	1
-1	15	1
-1	7	2
-1	3	2
-1	1	2
-1	2	1
-1	1	1
-1	16	1
-1	1	1
4	7	1
5	3	2
-1	1	1

Figure 5.2: Auxiliary Attributes Removal Phase

TABLE 5.2: Pseudocode for NextSibling() in DAGDOM A and SiXDOM

DAGDOM: NextSibling()

```
Let the top of the stack point to node a.
    Let b be the next sibling of a in the CS[1].
IF a has no next sibling in the CS THEN
 v has no next sibling.
Return FALSE
        IF CopyNumber less than OfNumber THEN
            Set CopyNumber to 1
        IF b is a share node THEN
        share node then b = idref[b]
        Set CurrentPosition = CurrentPosition +Stsize[b]
    ELSE
        Set CurrentPosition = CurrentPosition +Stsize[b]
    Return TRUE
ELSE
    b is a share node
        share node then b = idref[b]
        Let c be the next sibling of b in the CS.
    IF b has no next sibling in the CS THEN
 v has no next sibling.
Return FALSE
        Move the stack node, make it point to c
            Set CurrentPosition = CurrentPosition +Stsize[b]
        Set CopyNumber to 1
            IF no Mullti Attribute THEN
        Set the OfNumber to 1
        ELSE
Set the OfNumber to the value of Multi
Return TRUE.
```

DAGDOM A: NextSibling()

```
Let currentWaker the top of the stack.
Let the nodeNum is the Node Number In VS
IF CopyNumber less than OfNumber THEN
    Set CopyNumber = CopyNumber +1
    Set CurrentPosition = CurrentPosition+ Stsize[nodeNum]
    Return TRUE
ELSE
    IF currentWaker has no next sibling in the CS THEN // Tested using DOM NextSibling method
 v has no next sibling.
Return FALSE
    Set CopyNumber = 1
    Set CurrentPosition = CurrentPosition+ Stsize[nodeNum]
Set OfNumber = Multi[nodeNum]
Return TRUE
```

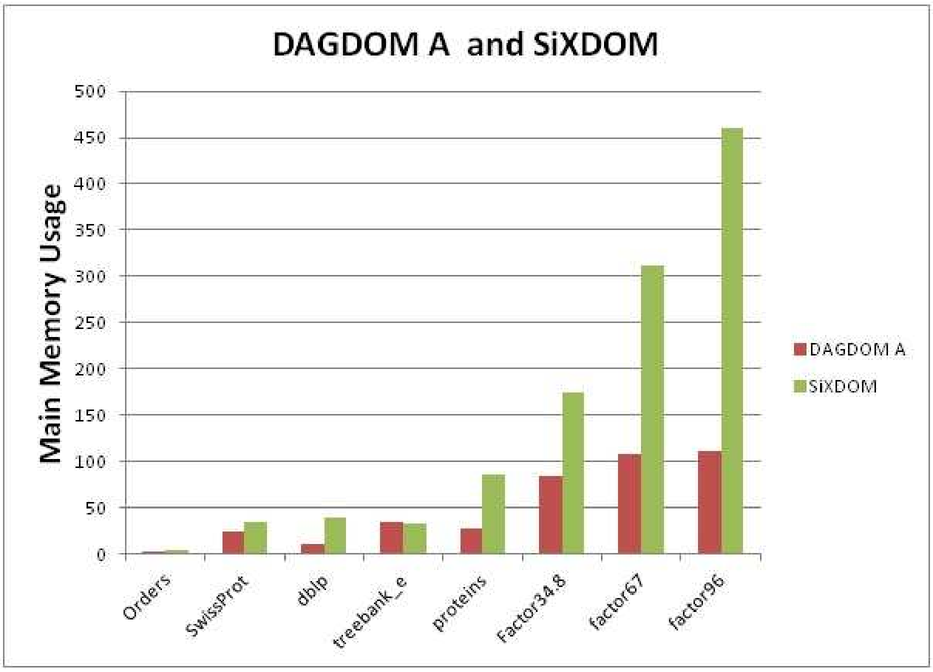


FIGURE 5.3: In-Memory Representation of DAGDOM A vs SiXDOM

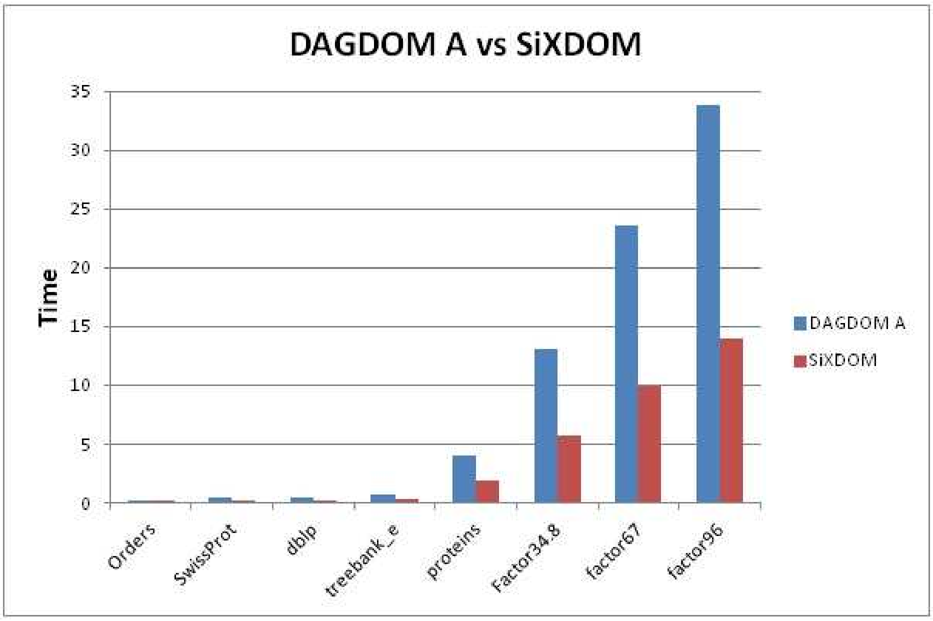


FIGURE 5.4: Running Time for DAGDOM A vs SiXDOM

Table 5.3: Statistics for Sample of the XML files. Column four shows how many values in Multi are greater than 255, and column five shows how many nodes are shared with stsize and multi equal to 1

XML File	**No.Normal**	**No.Share**	**No.Multi >255**	**No.Share(Stsize & Multi=1)**
Orders	11	0	1	0
SwissProt	58610	719261	1	331180
Treebank_e	471312	823812	7	501751
Dblp	4700	274799	9	35691
Proteins	77186	787275	38	81963
Factor1	73264	307913	1	174350
Factor2	100635	469299	1	240141
Factor38.4	206197	3299344	1	456853

investigate further optimisations to the auxiliary files of the auxiliary attribute removal phase.

We investigate two directions:

- In order to improve the speed of our approach we investigate removal of the unnecessary shared nodes.
- In order to improve the space usage of our approach we investigate using a threshold-based variable-length storage for `multi` and `stsize`.

We calculated some statistics for samples of the XML files (five XML files created by MacMill) as we have shown in Table 5.3. The number of normal and shared nodes for each file are shown in column 2 and 3 respectively. In column 4 we show how many nodes in each file have a value of multi greater than 255, and finally in the last column we show how many nodes are shared with the sub-tree size and multi value equal to one.

From Table 5.3, we make the following observations:

- The Multi array contains values which are most likely to be less than 255, and we used an array of integers for each node. Therefore, instead of using integer array we can use unsigned char array to store all values that are less than 255 and use a small hash map to store the values that are greater than 255. The same optimisation can be performed on sub-tree size.
- In the array of id_ref that contains `-1` for the normal nodes and the actual node number for the share node, we can replace all the share nodes that have `multi` and sub-tree size which is equal to `1` with normal node (actual node name); after that we notice more than 50% of those arrays will contain `-1`. Therefore, we modified the MacMill to create a new output called MacMill2. As a result, this modification will improve the speed of DAGDOM.

We created a new version of our approach after we made the developments based upon the above observations and called it *DAGDOM B*.

Performance

We draw comparisons of the in-memory representation and running time between DAGDOM A, DAGDOM B and SiXDOM. In Figure 5.5 we show the performance of in-memory representation between the above implementations.

The experimental results in Figure 5.5 show the in-memory representation of DAGDOM B achieves greater space reduction than DAGDOM A and SiXDOM. At this stage DAGDOM B is better than DAGDOM A and outperformed SiXDOM in all data files; however, the running time in Figure 5.6 shows approximately two times slower in CPU time for navigational operations (`FirstChild, NextSibling and Parent`) in comparison with SiXDOM, but it is better than DAGDOM A.

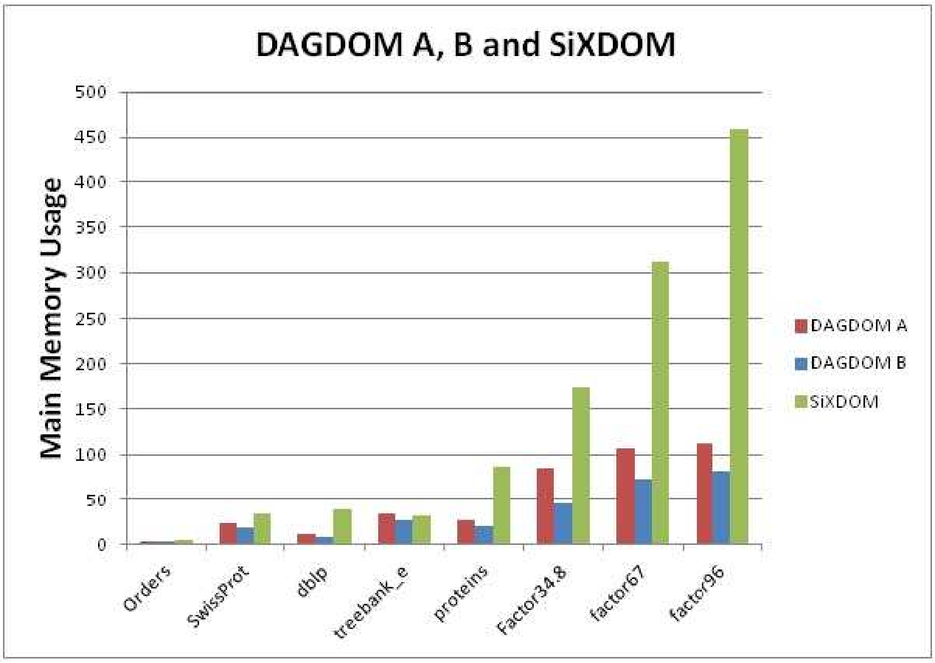


Figure 5.5: In-Memory Representation of DAGDOM A, B and SiXDOM

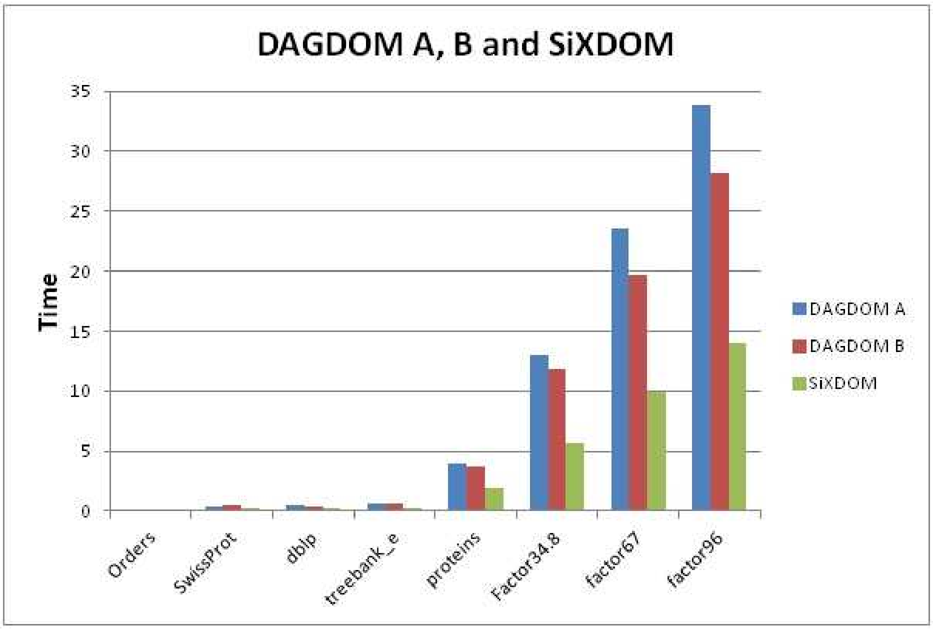


Figure 5.6: Running Time for for DAGDOM A, B and SiXDOM

TABLE 5.4: The behaviour of real and synthetic data files regarding the idref values. The test is performed to calculate the actual number of bytes to represent the id_ref number (in binary) for share nodes; the last column is the average bytes per node

File Name	Shared%	1-Byte%	2-Bytes%	3-Bytes%	Avg-Bytes%
SwissProt	0.5	0.71	0.26	0.03	1.32
Treebank_e	0.25	0.33	0.50	0.17	1.84
dblp	0.86	0.02	0.67	0.32	2.32
proteins	0.88	0.41	0.43	0.16	1.75
Factor1	0.35	0.27	0.26	0.48	2.23
Factor2	0.40	0.21	0.24	0.55	2.34

In the next section we discuss more details about another development to our approach in order to improve space usage.

5.3 Variable Length Encoding

In addition to the shrunk skeleton, we currently have three integer arrays; for example in `treebank_e.xml` the in-memory representation is approximately equal to 16MB. As a result, the size of these arrays is a significant part of the overall space usage. Therefore in this section we will consider *Variable Length Encoding (VLE)* in our approach in order to reduce space usage.

Suppose that we encode the id_ref values as follows: **1** byte for all `idref` values which are less than or equal to 255, **2** bytes for all `idref` values which are greater than 255 and so on. The average space usage is as shown in Table 5.4. We note that using VLE, the benefit of reducing share nodes is clearer. We also note that VLE is more effective on real than synthetic files.

In VLE we implement a representation to encode any sequence of n integer numbers, and to access the *ith* number in an efficient way, support the operation below:

- `access(i)`, where `i` is the positive integer number from`1` to `n`

This representation contains five approaches of encoding: Naive, Threshold, Fixed Bit Based, Byte Based and Bit Based. We note that in Section 5.2 we already have thresholding in multi and Stsize. We will consider an input array of size 5 as follows: **2, 314, 117, 6, 410**, and we will encode this array by our approaches below. We note that the values of this array are `idref` values. The array of`idref` maybe contains **-1** value; therefore we need to store it as **0**'s.

We discuss the details of the five approaches of encoding as follows:

- Naive approach: represent any number as an integer by using an integer array; support `access(i)` by accessing the index `i` of that array.
- Threshold: represent some of the numbers in a char array if less than the threshold; for example in Section 5.2 we used 255 as a threshold for multi array. Otherwise, store the numbers in an integer array as we show in the `Threshold_access(i)`.

Threshold_access(i)

```
IF idref[i] <= Threshold THEN
    RETURN idref[i]
ELSE
    FOR J=1 TO index.lenght
        IF index[j]=i THEN
    RETURN value[j]
```

From the above example which is the array of size 5 integers(**2, 314, 117, 6, 410**) we will consider three arrays: `idref` of type unsigned char, `value` of type unsigned integer and `index` of type unsigned integer. Now, if we apply the Threshold approach on the example with **255** as a Threshold, the above arrays will be as follows:

- `idref` will contain: **2, 0, 117, 6, 0**.
- `value` will contain: **314, 410**.
- `index` will contain: **2, 5**.

We note that node numbers start at **1**, while array indices start at **0**. If we ask for `Threshold_access(2)`, the `idref[2]` is equal to **0**, (which means the value is greater than **255**), then it will go through index array to find the index of value **1**, which is **0**. The last step is to find the `value[0]` which is equal to **314**.

- Fixed BitBased: represent any number based on how many bits it needs for the maximum number in all sequences to be represented in binary. We used the same data structures as in a bitbased approach but in a fixed way, which means that all the numbers have the same length in the Bit-Vector as `Fixed_access(i)`:

Fixed_access(i)

```
IF i == 0 THEN
    Set str=1
    Set end= str + Fixed.length-1
ELSE
    Set str= (i* Fixed.length)+1 //Fixed.length= length of index.
    Set end= str + Fixed.length
  RETURN bitString.subString(str,end)-2
```

From the above example: the maximum number is 410, and will take 9 bits to represent in binary, then the Bit-Vector will contain 8 **0s** followed by **1** for each value from our input as follows:

000000001 000000001 000000001 000000001 000000001

By `Fixed_access(2)` then:

`str = 2*9(Fixed.length)+1= 19`, `end = 19+9(Fixed.length)-1 = 27`

The result of `bitString.subString(19,27)-2 = 117`. Note: decrease the value by **2**, because we increased the input values by two in case we have a **-1** value.

- Byte Based: represent each number in one byte. In this option we used a bit_vector data structure and used `RANK` and `SELECT` operations to support access as shown in `ByteBased_access(i)`:

```
ByteBased_access(i)
Set ref1= rsd.Select(i, 0) //start from 0 (x)
  Set ref2= rsd.Select(i+1, 0) //start from 0 (x+1)
  Set str= ref1-(i), Set len= (ref2-ref1)-1
  //check if Select0(x+1)-Select0(x)==1 for normal node
  IF ref2-ref1 == 1 THEN
    RETURN Num=-1 //For Normal Node
  ELSE
```

```
        IF len >1 THEN //For Share with number >=256
            Set i=(str+len)-1
            RETURN Num tmp= NoB[i]//Array of Char Stored the REF Value//After Convert to Integer Value
        ELSE //For Share with number<=256
            RETURN Num = NoB[str]+1 // Array of Integers stored the REF Value
```

From the example above, we need one byte for the first, third and forth values and two bytes for second and fifth values. The Bit-Vector will contain:

01 011 01 01 011 (**0** followed by **1** if we need **1** byte, **0** followed by two **1**s if we need two bytes, etc.). We have an array of unsigned char called `NoB` which will contain: **2, 58, 1, 117, 6, 154, 1** (**1** represents **255**).

By `ByteBased_access(1)` then: `SELECT (1,0) = 5`, `SELECT(2,0) = 7`. `len = 7-5=2` (greater than **1**), then `Num = 58+(1*255)+1= 314`

- BitBased: represent any number based on how many bits is needed to represent it in a binary. In this option we used Bit_Vector data structure in addition to Bit-String class [52]. See `BitBased_access(i)`:

```
BitBased_access(i)
IF i == 0 THEN
    Set str=0
    Set end= rsd.Select(i, 1)
ELSE
```

Table 5.5: Main Memory Usage for Stsize File After Applying the VLE (how many bits per node for each data file)

Options	**Stsize**				
	Swissprot	**Treebank**	**proteins**	**factor1**	**factor2**
BitBased	**5.92**	**7.62**	**8.69**	**6.84**	**7.09**
FixedBased	**31.81**	**31.81**	**35.47**	**30.58**	**31.81**
ByteBased	**10.41**	**10.41**	**10.42**	**10.41**	**10.41**
Threshold	**8.04**	**8.00**	**8.19**	**8.01**	**8.00**
Naive	**64.00**	**64.00**	**64.00**	**64.00**	**64.00**

```
        Set str=(rsd.Select(i-1, 1)+1)
        Set end=rsd.Select(i, 1)
    RETURN bitString.subString(str,end)-2
```

From the above example we need **2, 9, 7, 4, 9** bits respectively for our input values. The Bit-Vector will contain number of **0s** based on the number of bits minus one, and followed by **1** for each value from our input as follows:

01 000000001 0000001 0001 000000001

By `BitBased_access(2)` then: `str = SELECT(1, 1)+1) = 11` `end = SELECT(2, 1)= 17`. The result of `bitString.subString(11,17)-2 = 117`

Tables 5.5, 5.6, and 5.7 show the space usage of our auxiliary files after applying the above representation options. From Tables 5.5, 5.6, and 5.7, we notice that overall the best option is to use a BitBased representation, but like ByteBased shows better results id_ref (most of the numbers are **-1**) . So, we can apply the proper option based on the data files.

Tables 5.8, 5.9, and 5.10 show the running time for our auxiliary files after applying the above representation options to perform VLE.

Table 5.6: Main Memory Usage for id_ref File After Applying the VLE (how many bits per node for each data file)

Options	**id_ref**				
	SwissProt	**Treebank**	**proteins**	**factor1**	**factor2**
BitBased	**8.42**	**7.41**	**14.86**	**9.97**	**11.36**
FixedBased	**29.31**	**30.58**	**29.31**	**28.05**	**29.31**
ByteBased	**7.27**	**5.39**	**14.30**	**16.53**	**9.83**
Threshold	**85.43**	**118.24**	**87.49**	**116.64**	**117.81**
Naive	**64.00**	**64.00**	**64.00**	**64.00**	**64.00**

Table 5.7: Main Memory Usage for Multi File After Applying the VLE (how many bits per node for each data file)

Options	**Multi**				
	SwissProt	**Treebank**	**proteins**	**factor1**	**factor2**
BitBased	**5.23**	**4.43**	**4.55**	**4.55**	**4.54**
FixedBased	**15.09**	**8.81**	**19.00**	**16.44**	**17.75**
ByteBased	**10.41**	**10.41**	**10.41**	**10.41**	**10.41**
Threshold	**8.00**	**8.00**	**8.01**	**8.00**	**8.00**
Naive	**64.00**	**64.00**	**64.00**	**64.00**	**64.00**

Table 5.8: Running Time (*seconds*) for Stsize File After applying the VLE

Options	**Stsize**				
	SwissProt	**Treebank**	**proteins**	**factor1**	**factor2**
BitBased	**0.17**	**0.33**	**0.25**	**0.14**	**0.15**
FixedBased	**0.45**	**0.76**	**0.57**	**0.34**	**0.33**
ByteBased	**0.2**	**0.36**	**0.25**	**0.15**	**0.16**
Threshold	**0.09**	**0.17**	**0.11**	**0.07**	**0.07**
Naive	**0.04**	**0.07**	**0.04**	**0.03**	**0.04**

TABLE 5.9: Running Time (*seconds*) for id_ref File After Applying the VLE

Options	**id_ref**				
	SwissProt	**Treebank**	**proteins**	**factor1**	**factor2**
BitBased	**0.23**	**0.32**	**0.38**	**0.19**	**0.19**
FixedBased	**0.45**	**0.76**	**0.51**	**0.33**	**0.34**
ByteBased	**0.22**	**0.34**	**0.3**	**0.18**	**0.17**
Threshold	*	*	*	*	*
Naive	**0.04**	**0.07**	**0.05**	**0.03**	**0.03**

TABLE 5.10: Running Time (*seconds*) for Multi File After Applying the VLE

Options	**Multi**				
	SwissProt	**Treebank**	**proteins**	**factor1**	**factor2**
BitBased	**0.17**	**0.26**	**0.17**	**0.11**	**0.12**
FixedBased	**0.31**	**0.35**	**0.38**	**0.24**	**0.25**
ByteBased	**0.21**	**0.32**	**0.21**	**0.14**	**0.14**
Threshold	**0.09**	**0.16**	**0.1**	**0.07**	**0.06**
Naive	**0.04**	**0.07**	**0.05**	**0.03**	**0.03**

After we tested all the variable length encoding options separately with our auxiliary files, we integrated the best options of representation to DAGDOM. Next, we evaluated the performance of in-memory representation running time; we called this version *DAGDOM C* and in Section 5.4 we show the experiential evaluations.

5.4 Experimental Evaluation

In this section we draw comparisons of the space usage and running time between Xerces, DAGDOM, and SiXDOM.

5.4.1 Basic Setup

The test machine was an Intel(R) Pentium(R) with 8GB RAM, 2.80GHzCPU and a 3MB cache size, running Ubuntu 10.04.4 LTS. The compiler was g++.

- For `RANK` and `SELECT` we used rsdic library [53].
- In Xerces and SiXDOM, we take an XML file such as `orders.xml` file then create a new file called `VSorders.xml`, considering the element node only and ignoring the other components of the XML document (attribute, text node, etc). This functionality was implemented using C++.
- In DAGDOM we take an XML file, such as `orders.xml`, by using MacMill software create a compressed file called `Morders.xml`, considering the element nodes only and ignoring the other components of the XML document (attribute, text node, etc).
- We create XML files after modifying MacMill, and those files are called MacMill2.
- In the Auxiliary Removal Attribute phase we take an XML file, such as `orders.xml`, and create a new shrunken XML file called `Sorders2.xml` (from MacMill2). These files consider the element nodes only and ignore the other components of the XML document (attribute, text node, etc). In addition, we create three text files: id_ref, Stsize and multi. This functionality was implemented using C++.
- After that, using C++, we traverse the files and we find the main memory usage (as shown in Table 5.11) and the running time navigation operations (`FirstChild, NextSibling and Parent`) shown in Table 5.12.

Table 5.11: Main Memory Usage of XML Representations in Xerces, DAGDOM, DAGDOM C, and SiXDOM (MB)

Version	Orders	Swiss.	dblp	Treeb.	prote.	fact1	fact2	Fact34.8	fact67	fact96
Xerces	28.93	553.71	615.04	452.80	3920.80	310.95	621.22	11775.88	20542.48	·····
DAGDOM	0.00	540.00	185.09	949.91	563.96	265.04	394.97	2253.48	3539.11	4794.40
DAGDOM C	2.62	10.29	6.33	16.11	13.00	6.57	9.53	40.44	78.09	77.25
SiXDOM	4.54	33.75	39.23	31.76	86.01	19.03	37.82	220	311.41	459.93

- We tested on `orders.xml`, `SwissProt.XML, dblp.xml, treebank_e.xml and proteins.xml` [9], and on some synthetic XML data files such as `factor1.xml`, `factor2.xml` .etc from the xmark files [54].

5.4.2 Main Memory Usage

In Table 5.11 we draw comparisons of the main memory usage (MB) between Xerces, DAGDOM (Using Xerces with MacMill File), DAGDOM C (DAGDOM using SiXDOM after Auxiliary Attribute Removal, Share Node Removal phase and VLE) and SiXDOM. We notice that DAGDOM achieves significant space reductions over Xerces (15 times space reduction at least), and as we can see `factor96.xml` (VS size is 2.9 GB) in our tests was not able to be processed by our test machine; this is because it exceeds the maximum virtual space. Compared to SiXDOM, as we show in DAGDOM C, the main memory usage was reduced typically 2 to 5 times.

5.4.3 Running Time

In Table 5.12 we draw comparisons of the traversal time (wall-clock time in seconds) between Xerces, DAGDOM (Using Xerces with MacMill File), DAGDOM C (DAGDOM using SiXDOM after Auxiliary Attribute Removal, Share Node Removal

Table 5.12: Running time (*seconds*) of XML representations in Xerces, DAGDOM, DAGDOM C and SiXDOM

Version	**Orders**	**Swiss.**	**dblp**	**Treeb**	**prote.**	**fact1**	**fact2**	**Fact34.8**	**fact67**	**fact96**
Xerces	0.01	0.11	0.12	0.11	0.83	0.07	0.14	360.00	936.54	·····
DAGDOM	0.19	6.69	7.35	6.03	44.85	3.78	7.50	143.52	259.77	357.44
DAGDOM C	0.03	0.87	0.84	1.10	6.95	0.61	1.15	22.45	39.93	57.49
SiXDOM	0.01	0.21	0.20	0.28	1.97	0.14	0.29	5.70	9.86	13.91

phase and VLE) and SiXDOM. Our test used wall-clock time to measure the CPU time.

Table 5.12, shows three times slower in CPU time for navigational operations in DAGDOM C compared with SiXDOM, and we notice that are 4-5 times slower in small data files in comparison with Xerces implementation, but DAGDOM C showed extreme successes on the large files such as `factor67.xml`.

5.5 Summary

In this chapter we have presented an approach called DAGDOM to support navigation operations with space-efficient in-memory representation. Further optimisation of the naive implementation of DAGDOM has been considered by careful application of succinct data structures and variable length encoding. DAGDOM yields a good result in comparison to Xerces and SiXDOM implementations particularly in main memory usage, but slower in CPU time for navigational operations, but the experimental results show that CPU time in DAGDOM is extremely successful on large XML files but is slightly slower than SiXDOM.

Chapter 6

Representing Attributes and Textual Data

In Chapter 5 we optimised DAGDOM to obtain significant space reductions over an existing space-efficient Xerces DOM implementation [1] (typically 2 to 5 times space reduction), with 4-5 times slower in CPU time for navigational operations. However, DAGDOM was based on the MacMill compression method [2] and as MacMill removes text and attributes, DAGDOM cannot represent these components of an XML document.

In this chapter we will introduce a version of DAGDOM with complete representation of an XML document, particularly the attribute and text nodes, and we have called this **DAGDOM+**. Furthermore, we will present a novel approach to representing the textual data with a mapping strategy, which maps the text nodes in the structure of the XML document to the textual data values in a space-efficient way.

This chapter is organised as follows. In Section 6.1 we discuss how we handle the attribute and text nodes in DAGDOM and we perform an experimental evaluation of the space usage and traversal time for DAGDOM+. Next, in Section 6.2 we will present the textual data representation, summarise some of the related work, and we will point out the differences of text document order and text element order. In Section 6.3 we will present the problem of storing textual data in XML documents and we give an explanation of how to apply our approach which is called the *Labelled String Sequence Problem (LSSP)*. In Section 6.4 we will give the solution to LSSP and will explain our LSSP mapping strategy.

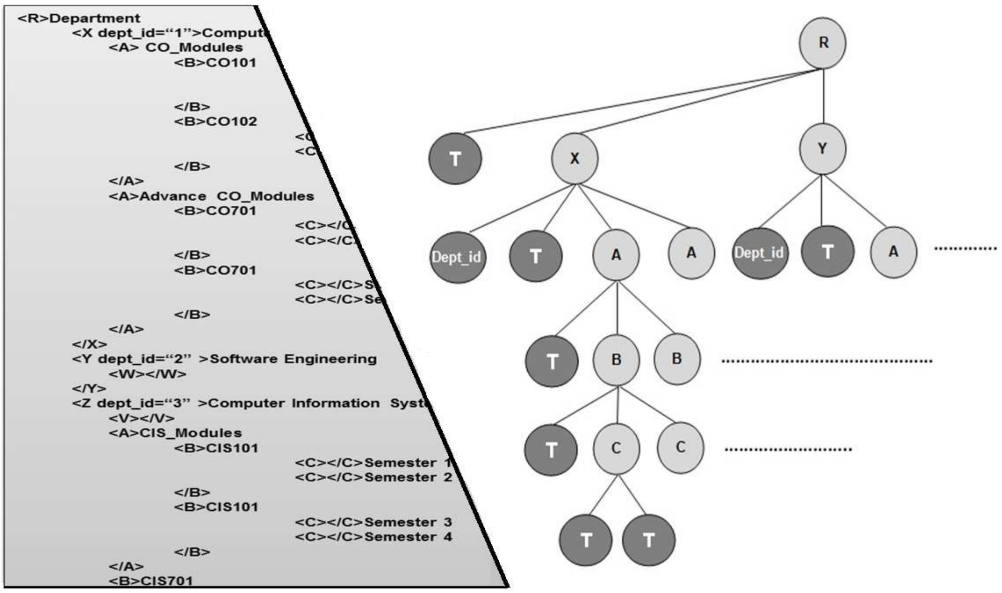
6.1 Attribute and Text Nodes

In this section we discuss DAGDOM+. There are some potential obstacles to overcome at this stage which are as follows:

- MacMill does not support attribute and text nodes as we saw in Chapter 4, which means we need to develop MacMill to handle attribute and text nodes in the skeleton.
- We need to ensure that compression performance is not affected by handling text and attribute nodes.
- We need to distinguish between the different attribute and text nodes, and for this we adapt the strategy introduced in Chapter 4 to distinguish between nodes in the *Virtual Skeleton (VS)* while navigating the *Compressed Skeleton (CS)*.

We therefore addressed the above points by a sequence of steps that begins from the original XML files as follows:

- Rewrite the XML files as follows: all text nodes are replaced by a single element <`T/`>. Then we introduce the attribute nodes of an element node as additional



```
<R>Department
        <X dept_id="1">Comput
            <A> CO_Modules
                    <B>CO101


                    </B>
                    <B>CO102
                                        <C
                                        <C
                    </B>
            </A>
            <A>Advance  CO_Modules
                    <B>CO701
                                        <C></C
                                        <C></C
                    </B>
                    <B>CO701
                                        <C></C>S
                                        <C></C>Se
                    </B>
            </A>
        </X>
        <Y dept_id="2" >Software Engineering
            <W></W>
        </Y>
        <Z dept_id="3" >Computer Information  Syst
            <V></V>
            <A>CIS_Modules
                    <B>CIS101
                                        <C></C>Semester 1
                                        <C></C>Semester 2
                    </B>
                    <B>CIS101
                                        <C></C>Semester 3
                                        <C></C>Semester 4
                    </B>
            </A>
            <B>CIS701
```

FIGURE 6.1: Virtual Skeleton with Attribute and Text Nodes

children of the element, and apply MacMill to the rewritten XML files. On the left part of Figure 6.1 we show part of the XML file and the new VS with attribute and text nodes on the right.

- Adding these nodes can hurt compression; this is because the nodes may have the same element name but different attribute name, and then MacMill will not compress these elements together as before. In Figure 6.2 we show an example of two sub-trees that are considered the same in original MacMill but not when we have the text and attribute nodes. In order to see the compression performance in our data files we create the CS using MacMill after rewriting the XML file with attribute and text nodes, then apply all the pre-processing phases we showed in Chapter 5; after that we measure some statistics on the new data files as we show in Table 6.1.

In Table 6.1 we consider in our measures the original size including textual data, CS without attributes and text (CS-Size), CS with attributes only (CS1-Size) and

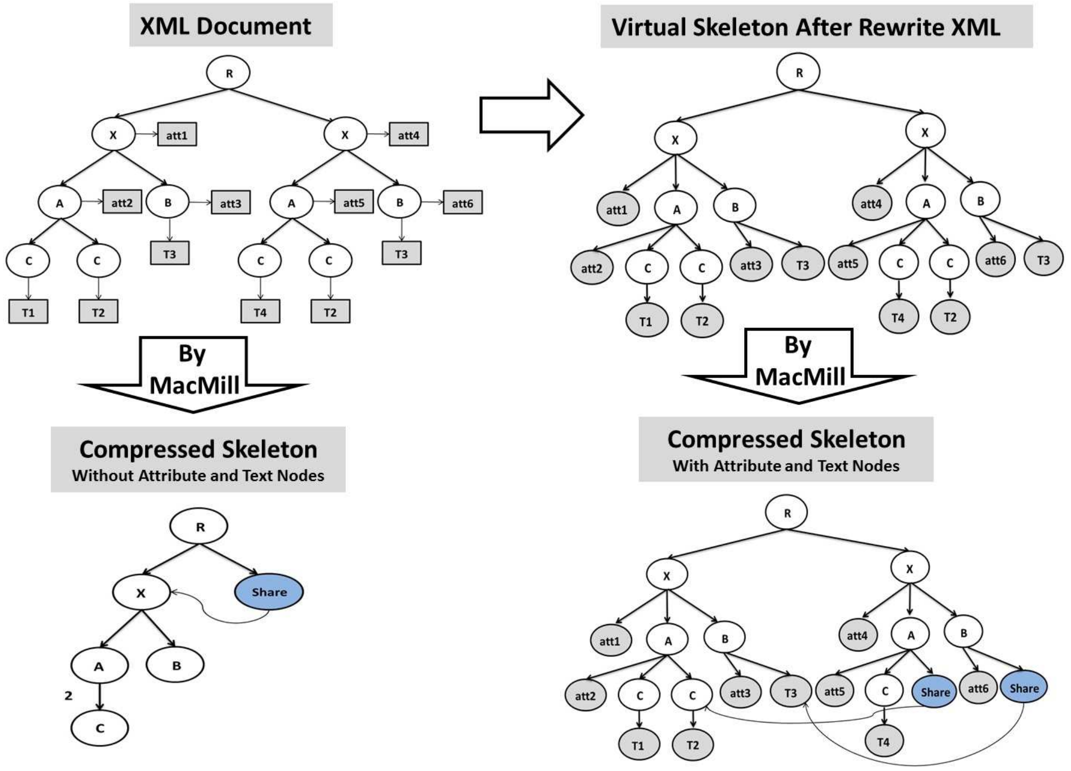


FIGURE 6.2: XML Document with two sub-trees that have the Same Element Names But with Different Attribute and Text Nodes (Considered the same in original MacMill but not once when we have text and attribute nodes)

TABLE 6.1: Sizes of XML Data Files: original, VS (elements only) and the sizes after we handled the attribute and text nodes in the VS. CS-Size (compressed skeleton without attribute and text),CS1-Size (compressed skeleton with attribute only), CS2-Size (compressed skeleton with attribute and text nodes) and No.Ele(number of elements in the XML file. All the CS Sizes after MacMill2)

File Name	orders	SwissP.	Treeb.	Prote.	Fact1.	Fact2.
Original-Size	5.1 MB	109.5 MB	82.1 MB	600.0 MB	116.5 MB	233.7 MB
CS-Size	272 Byte	30.5 MB	41.4 MB	34.8 MB	14.4 MB	21.7 MB
CS1-Size	272 Byte	39.2 MB	41.4 MB	38.8 MB	15.4 MB	23.1 MB
CS2-Size	964.7 KB	114.3 MB	97.6 MB	74.2 MB	32.9 MB	48.1 MB
No.Ele.	150001	2977031	2437666	21305818	1666315	3337649

Table 6.2: Analysis The Raw data of Table 6.1,

File Name	orders	SwissP.	Treeb.	Prote.	Fact1.	Fact2.
No.attr:Ele	0%	74%	0%	5%	104%	23%
No.Text:Ele	100%	182%	200%	174%	181%	181%
No.Ele-CS	0%	26%	53%	4%	23%	17%
No.Share: Ele.CS	0%	50%	25%	82%	35%	40%
No.Ele-CS1	0%	19%	53%	4%	12%	15%
No.Share: Ele.CS1	0%	59%	25%	76%	40%	44%
No.Ele-CS2	10%	32%	42%	4%	16%	14%
No-Share: Ele.CS2	50%	45%	27%	43%	97%	38%

CS with attributes and text nodes (CS2-Size); this size excludes textual data. In addition, we show the percentage of normal and share node to the number of original nodes in CS, CS1 and CS2.

We notice from Table 6.1 that when the number of attributes is low the compression is unaffected. Therefore, adding attribute nodes is usually cheap since CS1 size is rarely much bigger than CS. If we look at CS2 size, we notice in some XML file that the size is similar to the original size and larger than CS and CS1 as well; for example each element in order.xml has a text node and CS2 is much larger than CS, even though CS2 does not include the values of text nodes, but there is a lot of compression happening, as we show in Table 6.2.

Before discussing the results, we expect to see one of two cases as follows:

(a) Adding attribute and text nodes reduces the compression.

(b) Adding attribute and text nodes leaves compression unchanged.

Table 6.2 shows: Number of attributes to the number of Elements (No.attr:Ele), Number of text to the original number of elements (No.Text:Ele), Number of elements in CS to the number of original elements (No.Ele-CS:Ele), Number of share nodes to the number of elements in CS (No.share:Ele.CS), Number of elements in CS1 to the number of original elements plus number of attributes (No.Ele-CS1:Ele), Number of share nodes to the number of elements in CS1 (No.share:Ele.CS1), Number of elements in CS2 to the number of original elements plus number of attributes and text nodes, (No.Ele-CS2:Ele), Number of share nodes to the number of elements in CS2 (No.share:Ele.CS2).

From Table 6.1 and Table 6.2 we now discuss an example are of our data files in detail. If we look at `SwissProt.xml`, we can see the number of element nodes is equal to 2,977,031 and 74% of the elements have an attribute, and most of the elements the have more than one text node (182%: number of elements). When we compressed `SwissProt.xml` by MacMill, the number of element nodes reduced to 26% (50% of these nodes are a share nodes). Now, in the same example (`SwissProt.xml`) we show our measurements for both cases, one with attributes and the other with attribute and text nodes, as follows:

- If we consider the attribute nodes, then the number of nodes will increase by the number of attribute nodes, and the number of nodes on the CS is 19% of the number of element and attribute nodes together, which means we have more chance to compress more nodes together. In addition, the percentages of share nodes increased up to 10%.
- If we consider the attributes and text nodes, then the number of nodes will increase by the number of attribute and text nodes, and the number of nodes on the CS is 32% of the number of element, attribute and text nodes, and the

percentages of share nodes decreased by 5%, which means the performance of compression is approximately the same as without attribute and text nodes.

Based on the above observations, case (b) is the result in most data files (Adding attribute and text nodes leaves compression unchanged); for example, in `factor1.xml`, the number of nodes in the CS is 16% of the number of element, attribute and text nodes, when we include the attribute and text nodes (104% attributes and 181% text nodes), and the same in `factor2.xml`, `treebank_e.xml` and `proteins.xml`; which means the chance of compressing more nodes together is unaffected by including the attribute and text nodes. In `orders.xml` the number of nodes in the CS is 10% of the element, attribute and text nodes if we include the text nodes, and compared to 0% aproximately in CS, but not by too much.

On the other hand, by adding the attribute and text nodes to the skeleton, auxiliary information is needed in order to perform the navigation.

Although in Chapter 4 we kept track of the document order number of the current node in the VS while traversing the CS, we note that in fact doing so is unnecessary. There is no need to differentiate between identical element nodes, since DOM can only access their element name. However, attribute nodes with the same attribute name can have different attribute values, and text nodes with identical enclosing elements can have different values as well. Thus, when traversing the CS, it is necessary to keep track of the different attribute and text nodes. We do so by keeping track of the document order number of these nodes in the modified VS. To do this, we replace the sub-tree size data by attribute and text node number data to make the navigation.

For example, in the case of `NextSibling`; if we are at a particular element node and looking to move to the next sibling, instead of using sub-tree size, we only need to find how many attribute and text nodes are under this element node. Then adding

Table 6.3: Main Memory Usage of XML Representations in Xerces, DAGDOM, DAGDOM+ and SiXDOM

File Name	**Orders**	**SwissProt**	**treebank**	**proteins**	**factor1**	**factor2**
VS-Size	5.2	129.1	69.8	715.5	64.9	129.9
Xerces	56.78	1956.89	1349.38	10929.88	943.22	1873.37
DAGDOM	18.46	2151.43	2105.25	1402.63	654.08	948.22
SiXDOM	5.30	76.50	38.86	163.31	60.66	54.40
DAGDOM+	2.98	34.96	38.56	30.60	17.62	12.37

the number of attribute and text nodes to the element `id` number, will get the next sibling `id` number. It is noted that, compared to keeping the sub-tree size for each element in an integer array, the attribute and text node numbering did not take up much more space.

Performance

We evaluate DAGDOM+ as we show in Table 6.3, against DAGDOM (from Chapter 4), against Xerces, and against SiXDOM. The XML files were pre-processed to replace all text nodes with `<T/>` and all attribute nodes were added as children of the corresponding element node, as described above. The second row of Table 6.3 shows the VS size (with attribute and text nodes) for each file in MB

DAGDOM+ obtained significant space reductions over DAGDOM (the main memory usage was reduced on average 46 times) and SiXDOM (up to 5 times, such as `proteins.xml`). DAGDOM+ preserved the same performance as without attribute and text nodes and the results at this stage were very promising.

In Table 6.4 we draw comparisons of running time (wall_clock time in seconds) between Xerces, DAGDOM, DAGDOM+ and SiXDOM. Our test used wall_clock time to measure the CPU time.

Table 6.4: Running time of XML Representations in Xerces, DAGDOM, DAGDOM+ and SiXDOM

File Name	**Orders**	**SwissProt**	**treebank**	**proteins**	**factor1**	**factor2**
VS-Size	5.2	129.1	69.8	715.5	64.9	129.9
Xerces	0.01	0.40	0.27	360.65	0.19	0.37
DAGDOM	0.54	22.71	18.09	138.77	11.27	23.58
SiXDOM	0.02	0.77	0.75	5.18	0.43	0.87
DAGDOM+	0.11	3.77	2.89	34.15	1.83	3.72

We notice from Table 6.4 that the CPU time for basic navigational operations (`FirstChild, LastChild, NextSibling, PreviousSibling and Parent`) shows DAGDOM+ is much faster than DAGDOM, but is slower than SiXDOM.

6.2 Representing Textual Data

In this section, we present strategies to efficiently store and access textual data contained in XML documents. As we mentioned in Chapter 2, the basic API which is called `Node` contains functionality common to all nodes, including the operations to retrieve information associated with each node, such as the `getNodeValue()` operation which returns the textual value of the node that has a value. In fact, we note that some of our test files, such as `treebank_e.xml`, have textual data that occupied up to 80% of the document. Therefore, it is important to store text data carefully.

6.2.1 Compressing Textual Data

In Chapter 4 we saw how the MacMill [2] breaks down the XML document into containers of data values and a CS that describes the structure. Some other previous work has achieved an excellent compression ratio for storing the strings in a space-efficient manner; for example XMill [5] separates the structure and groups related textual data into a single container in order to apply an appropriate semantic compressor for each container.

The main shortcoming in the above approaches is that they do not support random access to the strings. Therefore, the aim of our approach is to store the strings in a space-efficient manner so that we can support random access to the `i-th` string.

6.2.2 Random Access To Compressed Textual Data

Delpratt *et al* [7]. formalised the problem of storing textual data in an XML document as the *string sequence problem (SSP)*. Given n (non-empty) strings $s_1, s_2, \ldots, s_n$, store this sequence of strings in data structure in order to support the `access(i)` operation which returns the *ith* string.

As noted in Chapter 3, Delpratt *et al.*'s approach [1] allows the text nodes to be numbered from $1..t$ in document order, where t is the number of text nodes. Their approach is summarised as follows. In the pre-processing phase, create a sequence of strings, which consist of the values of all text nodes in the document, numbered in document order. Store this sequence in a data structure for the SSP problem. When navigating the document, if we are at a text node, and the `getNodeValue()` operation is called, we get the document order number of the text node and use the `access()` operation of the SSP data structure to get the content of this text node.

We now look at two solutions to the SSP problem.

6.2.2.1 Delpratt *et al.*'s Approach

In [52] the Prefix-Sum data structure was considered (more details have been given in Chapter 2). SSP used the Prefix-Sums data structure to store the sequence of strings in order to support the operation of returning the *ith* string; if we are given n non-empty strings $s_1, s_2, \ldots, s_n$, then we concatenate $s_1, s_2, \ldots, s_n$ into a single character array, called `T`. Also the cumulative length of $s_1, s_2, \ldots, s_n$ is stored as the numbers $x_1, x_2, \ldots, x_n$ in the Prefix-Sum data structure, where $x_i = |s_i|$. Recall that the Prefix-Sum data structure supports the operation `SUM(x,i)` which returns $x_1 + \ldots + x_i$ (thus the value returned by `SUM(x,i)` is the `i-th` offset).

Blocked BZip and Caching Technique

We now discuss using Blocked BZip to represent the sequence of strings in a compressed manner. We let `T` denote the string which is a concatenation of $t_1, \ldots, t_n$ where t_i is the compressed *ith* string. In order to access the *ith* string we need to do the following:

- Compute $j = SUM(x', i)$, where $x'_i = |t_i|$.
- Compute $k = SUM(x', i + 1)$
- Return $substring(j, k - 1)$

This representation, which supports the `substring()` operation using a blocked BZip2 to divide the `T` into blocks of characters, uses BZip2 to compress each block of characters. As a result, when the individual string t_i needs to be retrieved, the block(s) containing it are decompressed. After that, compute the `subString(j,k)` which is required to copy the required characters from position `j` to `k` in order to retrieve t_i, where `j` is the starting position of the `ith` required string and `k` is the end position of the block(s).

Another development was achieved in [52], and is called the caching technique. The caching idea is to store text from the last decompressed block into a block cache of size `K` as an uncompressed block (using FIFO replacement mechanism when the cache is full). On the other hand, the subsequent accesses to a cached block do not require decompression so long as the block is not evicted from the text block cache. The implementations used a block size of 8 or 16 KB at most.

The caching idea improved decompression time, particularly if we used a large block(s) sizes of 8 KB or 16 KB and access text nodes consecutively. But if does not support random access to strings.

We now briefly digress to discuss the choice of block size by Delpratt *et al.* They only looked at block sizes of 8 KB and 16 KB. The experimental evaluation shows that the compression ratio of using blocked BZip2 with block size of 16 KB is generally better than using a block size of 8 KB, but with small differences. The result shows the compression ratio is roughly similar when the textual data is arranged in path order (the textual data with the same upward path from leaf node to root are arranged together) or document order.

In [7] reports were similar for upward path order, and document order and with smaller block size we can access individual data values faster because the decompression time is less than with large blocks, but when we increase the block size by more than 8 and 16 KB our results show the differences in a compression ratio; more details will be provided in Section 6.2.3. Thus, we considered the choice of block size in our approach.

6.2.2.2 Wavelet Trie

In [55] the compressed indexed sequence of strings was introduced. This includes the string sequence problem. In some applications such as website optimisation for database storage of telephone calls, there are strings that require a suitable compressed format. Therefore space-efficiency is highly desirable. Indexed sequences are stored by representing the sequence explicitly and indexing it using auxiliary data structures, for example B-Trees and Hash they gives a new data structure called *Wavelet Trie* for storing a sequence of strings that supports not only random access, but also supports searching, range and analytic operations.

The weakness of *Wavelet Trie* and Blocked BZip is that they do not get compression benefits by considering the tag that contains the text node. We now discuss this issue in greater detail.

6.2.3 Document Order Versus Element Order

In XML-specific compressors such as XMill [5], the importance of grouping text into containers based upon their parent element and applying appropriate compressors to each container was noted. The underlying compressor used was Gzip [56]. In contrast, Delpratt [7] used Bzip2 as the compressor, and reported little difference between compressing the text in upward path order (the textual data with the same upward path from leaf node to root are arranged together) and document order.

Our first objective was to reproduce these findings. To do this, we compared compressibility of textual data in element order (the textual data with the same element name are arranged together) versus document order using both Bzip2 and Gzip as compressors, and results are summarized in Table 6.5. We observe that using both Bzip2 and Gzip, there was an improvement in compression in element order versus

Table 6.5: Comparison between Gzip and BZip2 for Document and Element Order

File	**Uncompressed text (KB)**	**Doc-order**	**Element-order**	**Doc-order**	**Element-order**
		GZip		**BZip2**	
Orders	1634.586	430.5 KB	350.4 KB	266.7 KB	260.6 KB
SwistPros	41547.575	7.0 MB	4.6 MB	4.6 MB	3.4 MB
dblp	446278.506	113.7 MB	90.9 MB	78.8 MB	69.1 MB
Treebank_e	63517.666	26.4 MB	25.1 MB	24.1 MB	23.3 MB
Proteins	366109.163	88.6 MB	65.8 MB	67.2 MB	60.8 MB

document order. However, the improvement was significantly more when using Gzip versus Bzip. In all cases, though, Bzip with element order compressed the best.

The above test was performed using Bzip2 and Gzip on the entire concatenated text (in document or element order). However, since we need to perform random access, we need to look at using blocked Bzip2. Here, Delpratt reported that compression ratios degraded (in document order) when block sizes of less than 8KB were used, which justified their block size choice. Obviously, smaller block sizes are better for random access to strings. In order to understand the trade-offs in our case, we now study the compressibility of the textual data using blocked Bzip2 for a variety of block sizes, both in document and element order.

6.2.3.1 Compression Ratio: Document order versus Element order

We compare document order versus element order (the textual data with the same element name are arranged together) with Bzip2 as the underlying compressor and we consider blocked Bzip in our experiments.

- We ran the test on different block sizes (512, 1024, 2048, 4096 and 8192 Bytes).

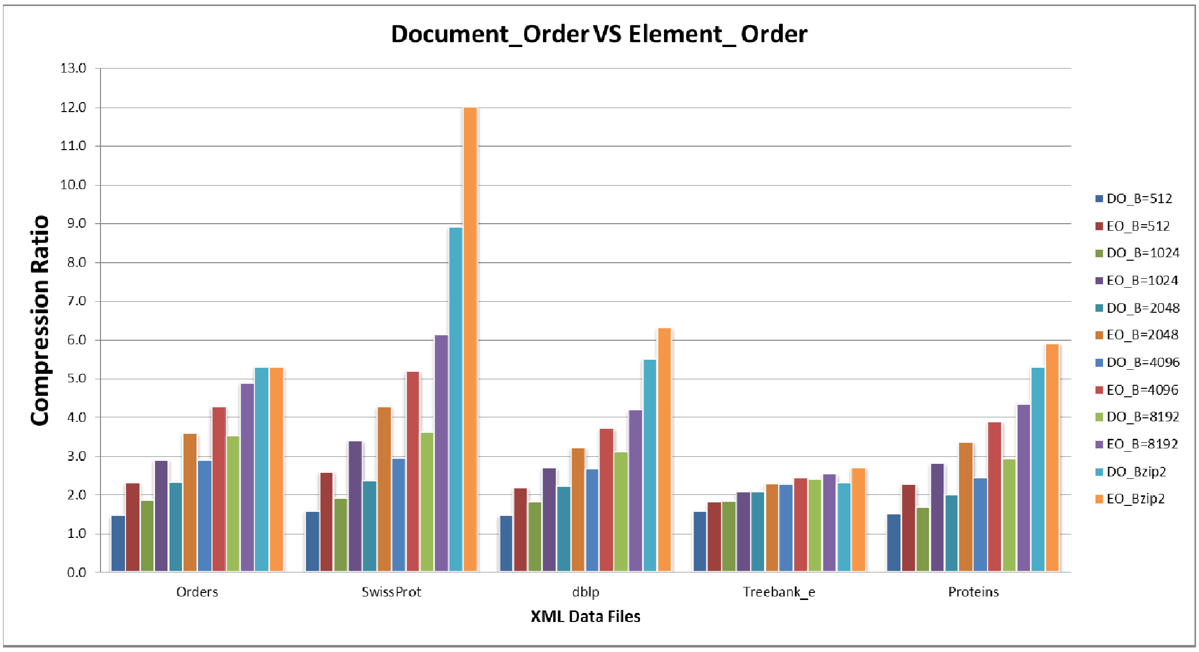


Figure 6.3: Compression Ratio in Document Order VS Element Order

- We tested different kinds of real data files (Orders, SwissProt, dblp, treebank_e and Proteins).
- In addition, we ran used BZip2 without dividing into blocks.
- The experiments were run on text document order and text element order.

Figure 6.3 shows the compression ratio in document order versus element order. We observe the following:

- The BZip2 with larger block sizes is generally better than smaller block sizes. But the compression ratio for all chosen sizes it is less than Bzip2. However for the file `treebank_e.xml` the compression with block size 8192 Byte was better than Bzip2 compression, but not by much. In both cases, document order and element order, applications would benefit from a smaller block size because decompression of a smaller block is faster.

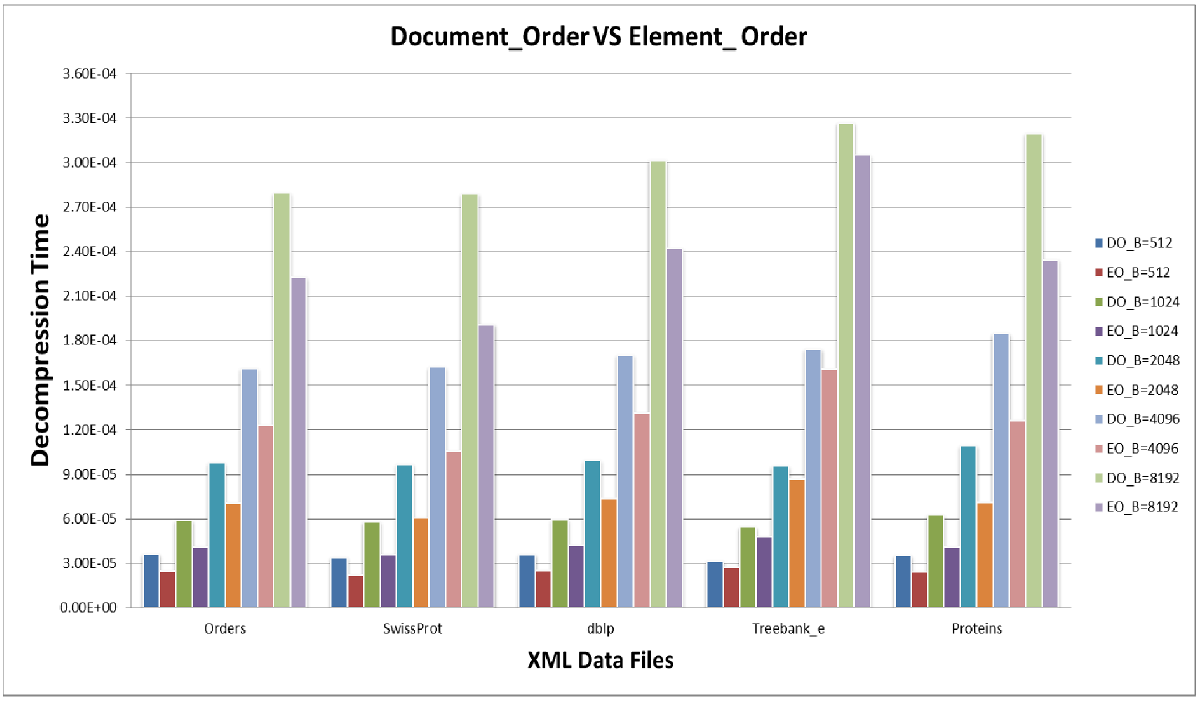


FIGURE 6.4: Decompression Time in Document Order VS Element Order

- Based upon the above figures, we notice differences in the BZip2 compression of text arranged in text document order and text in element order. We observe that the compression ratio of BZib2 in element order is better than in document order for all files. BZib2 uses the BurrowsWheeler transform which organises text into parts with similar contexts (details were provided in Section 2.5, Chapter 2). In element order we will find most of the similar contexts together and that is very useful because the text of approximately the same length will be set together and may be in one block(s).

6.2.3.2 Decompression Time: Document order versus Element order

Regarding decompression time, Figure 6.4 shows the decompression time after we ran our test with the same arrangement and data files as mentioned in Section 6.2.3.1. We observe the following:

- BZip2 with smaller block size is generally better than larger block size.
- The decompression time when we used the libBZip2 with smaller block size and in element order is generally better than with the larger block size in document order.
- When using BZip2 with smaller block sizes we can access individual data values quicker, because the small block size have less decompression time, especially for a collection of textual values that are small in length and where the text value begins far away from the start of the block. For such a case with larger block size, we may have to read double the number of characters as we do for smaller block size.
- The results show that decompression time in element order is less than in document order.

Our work leads us to conclude that when using BZip2 as an underlying compressor in a blocked compression scheme, storing data in element order is beneficial. At any given block size, text compressed in element order compresses better and decompresses faster than text compressed in document order. This helps us to use smaller block sizes in a blocked compression scheme, thus speeding up random access, without losing too much in compression ratio. In addition, we can figure out the suitable block(s) size in order to get a increase the compression ratio with faster access based upon the actual data.

6.3 Labelled String Sequence Problem

In the previous section we observed that storing text in element order is better even when using Bzip2 as a compressor. To model storing text in element order while preserving random access we now introduce the *Labelled String Sequence Problem (LSSP)*. Given `n` (non-empty) strings $s_1, s_2, \ldots, s_n$ and a label for each string $l_1, l_2, \ldots, l_n$, store the strings in a data structure which supports the operation *access(i, l)*, where l is a positive integer and $1 \le i \le n$. *Access(i, l)* returns s_i if $l = l_i$, otherwise either an error is flagged or an arbitrary string is returned.

The main difference between SSP and LSSP is that labels are considered in the `access` operation. Therefore, LSSP includes SSP on a special case, and any solution to the SSP will be a solution to the LSSP, in addition to the better space efficiency in LSSP.

Labelled String Sequence in XML DOM

We now describe how to apply LSSP to the storage of textual data in DOM and why LSSP is an appropriate approach to take.

We have already shown in Section 6.2.2 that we can number the text nodes from $1..t$ in document order, where t is the number of text nodes. Thus, our approach is summarised as follows. In the pre-processing phase, create a sequence of strings, which consists of the values of all text nodes in the document, numbered in document order, with label l for each string, where is the element `id` number (document order) of the parent element. Store this sequence of strings and labels in a data structure for LSSP. When navigating the document, if we are at a text node, and the `getNodeValue()` operation is called, we get the document order number and the label of the text node, where the label is the document order number of the parent for this text node. Then we use the `access()` operation of the LSSP data structure to get the content of this text node.

Similarly to the above approaches, such as XMill, LSSP compresses the textual data separately, but has a different way of grouping the textual data. Therefore, LSSP groups (element order) the textual data which have a parent with the same element *name* together and gives them the same label in order to maximise compression. Generally, labels can be ignored in the labelled strings sequence, but using labels generally gives a better space-time tradeoff.

6.4 Solution to the Labelled String Sequence Problem

In this section we will introduce the main technique which we used in our approach in order to obtain a solution to the Labelled String Sequence Problem.

6.4.1 Minimal Perfect Hashing

Perfect hashing is a technique for mapping a set of key values to the hash table with no collisions. In most general applications, we cannot know exactly what set of key values will need to be hashed until the hash function and table have been designed and put to use. However, when we know all the keys in advance, at this point there is a possibility to build a table which contains one entry for each key and no empty slot. Therefore the function is called minimal [57].

Minimal perfect hash functions are widely used for memory-efficient storage and fast retrieval of items from static sets, such as words in natural languages, reserved words in programming languages or interactive systems, universal resource locations (URLs) in Web search engines, or item sets in data-mining techniques [57] [58] [59] [60].

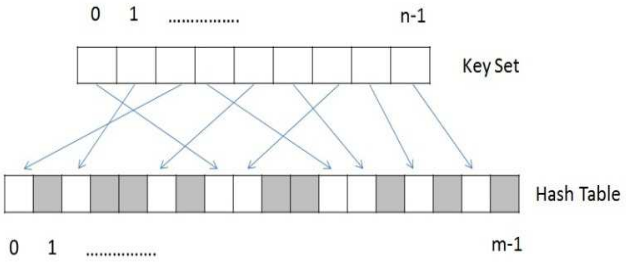


FIGURE 6.5: Perfect Hash Function

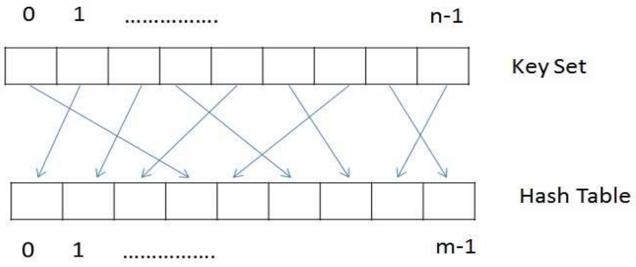


FIGURE 6.6: Minimal Perfect Hash Function

Let `U` be a universe of keys. Let `S` $\subseteq$ `U` be a set of keys given by the user, and let $|$`S`$| = n$. A minimal perfect hash function is a function `f` such that:

- `f` maps `U` to the range `1..n` [`f` is minimal]
- For any two different keys `x`, `y` in `S`, `f(x)!= f(y)` [`f` is perfect]

Figure 6.5 illustrates a perfect hash function and Figure 6.6 illustrates a minimal perfect hash function. In some hash functions, the range of `f` is slightly more than `n`. Then, we say the function is perfect with load factor `n`/(size of range of `f`).

There are different implementations of MPH that we can find in the CMPH library [57]. The CMPH library has efficient and newest algorithms in an easy-to-use, production-quality and fast API. The CMPH library works with huge entries and it has been used for sets with more than 100 million keys (even if it cannot fit in the main memory) for constructing minimal perfect hash functions. There are many algorithms supported in the CMPH library, but we used CHD and BDZ algorithms in our approach. The following are some of the distinguishable features of those algorithms:

- CHD Algorithm: it is considered the fastest algorithm for building PHFs and MPHFs in linear time. It can generate MPHFs that can be stored in approximately 2.07 bits per key, and for a load factor equal to the maximum one that is achieved by the BDZ algorithm (81%), the resulting PHFs are stored in approximately 1.40 bits per key [57].
- BDZ Algorithm: it is considered to be very simple and efficient and it constructs both PHFs and MPHFs in linear time like CHD. The maximum load factor one can achieve for a PHF is 1/1.23. The resulting MPHFs can be stored in approximately 2.6 bits per key [57].

Our approach will use the minimal perfect hash function in order to maintain the mapping strategy between skeleton document order and textual element order numbering very space-efficiently. The next section will show the mapping strategy that we used in our approach, which is supported by MPH [57].

6.4.2 The Labelled String Sequence Mapping Strategy

The first step of our approach is to create an MPH for each different element name, which means that all textual data with the same element name in the XML document have the same MPH file. The MPH file contains a unique key for each element in order to maintain mapping between skeleton document order and textual element order numbering very space-efficiently.

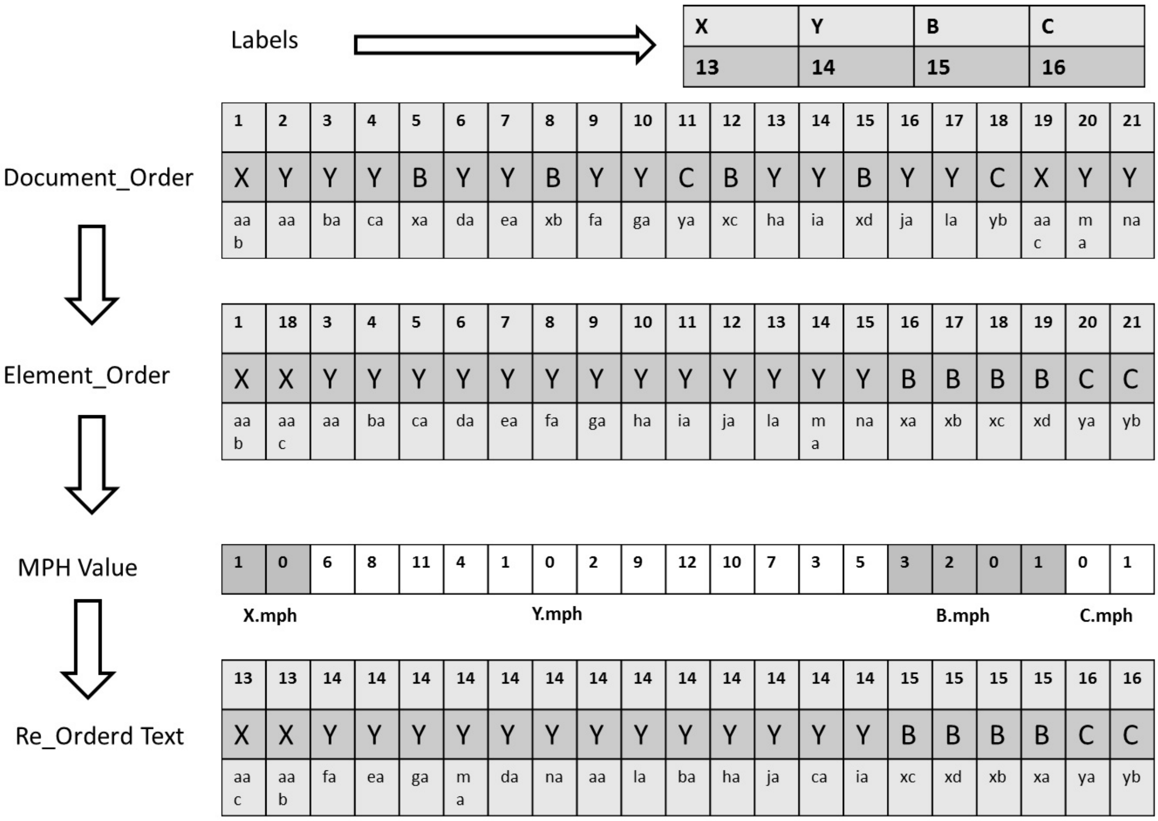
The input data for LSSP will be as follows:

- Text file contains the extracted textual data value from the XML file by the document order traversal and delimated by the symbol of greater than(" $>$ ").
- Labels auxiliary file which contains the label of each element in the XML file by document order traversal as well. We used the short-code value (see Section 3.4.1 for more details about the short-code data structure) for each element instead of using the element `id` number. The `short-code` value for elements starts from 13, as we show in Figure 6.7.

We now discuss the of LSSP as follows:

- Read the textual data and labels files for XML file.
- Re-order the textual data based on the element name order.
- From the CMPH library we use the supported algorithms such as CHD or BDZ in order to create an MPH on sets of skeleton document order numbers for each element in the XML document (which is the same order of textual data).
- For each element we re-order the text based on the MPH keys.
- Using the access operation we can read the `ith` string as skeleton, document order with label, return the `ith` string as stored and re-ordered based on the MPH keys.

Figure 6.7 shows the labelled string sequence mapping strategy for a small example.



Labels	X	Y	B	C
	13	14	15	16

Document_Order	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
	X	Y	Y	Y	B	Y	Y	B	Y	Y	C	B	Y	Y	B	Y	Y	C	X	Y	Y
	aab	aa	ba	ca	xa	da	ea	xb	fa	ga	ya	xc	ha	ia	xd	ja	la	yb	aac	ma	na

Element_Order	1	18	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
	X	X	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	B	B	B	B	C	C
	aab	aac	aa	ba	ca	da	ea	fa	ga	ha	ia	ja	la	ma	na	xa	xb	xc	xd	ya	yb

MPH Value	1	0	6	8	11	4	1	0	2	9	12	10	7	3	5	3	2	0	1	0	1
	X.mph							Y.mph									B.mph			C.mph	

Re_Orderd Text	13	13	14	14	14	14	14	14	14	14	14	14	14	14	14	15	15	15	15	16	16
	X	X	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	B	B	B	B	C	C
	aac	aab	fa	ea	ga	ma	da	na	aa	la	ba	ha	ja	ca	ia	xc	xd	xb	xa	ya	yb

FIGURE 6.7: Labelled String Sequence Mapping Strategy

Performance

To show the evaluation of BDZ and CHD algorithms on one of our data files and we will present the experiments on the labelled strings sequence by showing the evaluation of memory usage and the running times on different implementations: Naive, SSP and LSSP. The test machine was the same as we saw in Chapter 5, also with the same data files.

Firstly, we extracted the textual data from the XML file in document order, in addition to creating an auxiliary text file which contains the sequential textual node numbers in document order as well. After that we used the MPH algorithms in order to create the MPH files. Table 6.6 shows the result of `proteins.xml`. From

Table 6.6: Comparisons Between BDZ and CHD Algorithms. Elapsed Time:Sum is time for all elements.

Proteins	**BDZ_Algo.**	**CHD_Algo.**
Elapsed Time:Sum	431.449 SEC	447.003 SEC
Elapsed Time:Average/Element	6.7 SEC	7.5 SEC
Original Size	313.7 MB	313.7 MB
Size of (mph-file)	12.5 MB	19.1 MB

Table 6.7: Main Memory Usage of LSSP Compared to Naive and SSP

XML File	**Naive**	**SSP**	**LSSP**
Orders	2.06	0.06	0.03
SwissProt	31.24	0.90	0.40
Treebank_e	21.24	0.83	0.27
Proteins	239.45	6.06	3.05
Factor1	10.25	0.33	0.13
Factor2	20.50	0.66	0.26

Table 6.6, we notice that the size of auxiliary files for all textual data is 313.7 MB, and the MPH files reduced the size to 12.5 MB by BDZ algorithm and 19.1 MB by CHD algorithm. The average elapsed time of creating MPH files also shows that the BDZ is faster than the CHD.

In Table 6.7 we draw comparisons of the main memory usage (MB) between Naive, SSP and LSSP implementations.

We notice from Table 6.7 that LSSP obtained a good space reduction over Naive and SSP (typically 2 to 3 times) and improved the compression performance of storing the textual data in an efficient way in order to associate these data to the text node in the CS. In Table 6.8 we notice that LSSP is slower than Naive, but up to 2 times slower than SSP, thus trading off time for space.

Table 6.8: Running Time Performance of LSSP compared to Naive and SSP in *seconds*

XML File	Naive	SSP	LSSP
Orders	0.01	0.08	0.10
SwissProt	0.03	0.91	1.33
Treebank_e	0.02	1.31	1.61
Proteins	0.27	1.64	2.97
Factor1	0.01	0.47	0.78
Factor2	0.02	0.91	1.45

6.4.3 Summary

We have shown that DAGDOM+ gives very promising results after we handled the attribute and textual data into the VS. The experiments show that DAGDOM+ preserved the performance of DAGDOM without attribute and text nodes (DAGDOM C in Chapter 5), and typically reduced the space usage up to 5 times more than SiXDOM.

We have shown our strategy to store and access textual data contained in XML documents (Labelled String Sequence Problem), where we are using blocked BZip2 to store the textual data and using Minimal Perfect Hashing to maintain mapping between skeleton document order and textual element order numbering.

The experiments show that LSSP obtained a good space reduction over Naive and SSP (typically 2 to 3 times), but trading off time for space (slower). We are now able to represent the textual data and support random access to these data efficiently.

Chapter 7

Conclusion

The main objective of this thesis was to represent XML documents in-memory in a highly compressed format, while supporting operations of the kind supported by DOM.

We achieved this by using a compression method; MacMill [2] was one of the first to provide a compression on the XML tree structure, which represents an XML document as a directed acyclic graph (DAG) by sharing common subtrees.

We add functionality to the basic approach of MacMill that allows us to navigate the virtual skeleton, slightly modified to represent the virtual skeleton of MacMill in a space-efficient manner, since the skeleton is sometimes (much) larger than the virtual tree. In addition, we add attribute and textual data to the representation. In our implementation, DAGDOM, we show that by careful application of *succinct* data structures and variable length encoding, we are able to obtain significant space reductions over existing space-efficient DOM implementations (typically 2 to 5 times space reduction), with 4-5 times slower in a small data files in CPU time for navigational operations.

7.1 Technical Contributions

The technical contributions made by this thesis are summarised as follows:

In-Memory Representation Based upon MacMill

We examined the XML compression method MacMill in detail. We explained how we support DOM-like navigation on the compressed skeleton such as for basic navigation operations: *(getFirstChild(), getLastChild(), getNextSibling(), getPreviousSibling() and getParent())*. We considered the main obstacles of using MacMill directly as a basis for our work; thus we stored the sub-tree size under each node, in the compressed skeleton to distinguish between normal and share nodes, and for each share node to know how many representative nodes they represent in the virtual skeleton. We used a stack for going up in the tree to find the parent. In addition, we built a hash table so that for any node we can retrieve the corresponding subtree size, and built a hash table for the `macmill:id` in order to find the representative for the share node.

In-Memory Representation of XML Document Using DAG-DOM

We developed a naive implementation of our approach by careful application of other space-efficient DOM implementations (e.g., SiXDOM) [1], to handle the attribute and text nodes in the skeleton, and we obtained significant space reductions over existing space-efficient DOM implementations. We made significant changes for navigation before using SiXDOM in our approach. A consequence of the numbering of

SiXDOM nodes is that we can use arrays instead of hash tables, which led to optimise what is stored in the skeleton by stripping `macmill:id` and `macmill:multi` attributes and create a new shrunken XML file, in addition to three auxiliary text files:`Id_ref.txt`, `Stsize.txt` and `Multi.txt`. Then we represented the new shrunken skeleton by using a SiXDOM component, and making the navigation operations navigation as in the previous version of our approach: `(getFirstChild(), getLastChild(), getNextSibling(), getPreviousChild(), getParent())`. Another modification to the shrunk skeleton was done after making some statistics to the behaviour of our XML files, which shows another improvement in our approach.

Further development was made regarding DAGDOM based on the different behaviour of real and synthetic data files. We implemented a new representation to encode any sequence of integer numbers and to access numbers in an efficient way and support the operation: $access(i)$, where i is the positive integer number from *0* to n. Finally, implemented DAGDOM with complete XML representation (with attribute and text nodes), and the results were extremely successful.

DAGDOM yields a good result with in comparison to DOM and SiXDOM implementations, particularly in space usage with 4-5 times slower than SiXDOM in CPU time for navigational operations and the experimental results show that CPU time in DAGDOM is better than DOM on large XML files.

Textual Data Representation

We presented new strategies to efficiently store and access textual data contained in XML documents. The *Labelled String Sequence Problem (LSSP)* included the strings sequence problem in addition to the label that was added to the user inquiry. When we call the `getNodeValue()` the LSSP will use `access(i,l)` operation, where i is the text node number in document order and l is the label for each text node.

Our approach suggests grouping together the textual data that share the same parent element and giving them the same label, and we use the minimal perfect hash function in order to maintain the mapping strategy between skeleton document order and textual element order numbering.

7.2 Future Work

There are a number of future tasks on which we still need to focus. The first task is looking to optimise our code to improve the speed performance, because all the optimisations in this thesis were to improve the performance of memory usage. An additional focus of future work will be creating a comprehensive evaluation of DAGDOM and to add more functionality. Furthermore, as opposed to keeping a stack to perform navigation, we can use the advanced data structures presented. Finaly, in addition to the tests that we have performed, it would be very interesting to develop DAGDOM to support the more than DOM navigation operations, such as updates made using DOM methods. In addition to investigate the performance of DAGDOM in real applications in [61].

Appendix A

XML Data Files

Our choice of data files is taken from [9];the corpus of XML documents. Our choice of files gives us a range of typical XML documents which are described in Table 3.1. We also used synthetic XML files that were created using the xmark data generator [54]. Xmark generates a typically well-structured XML document.

Table A.1: Description of XML files in our XML corpus taken from [9].

XML Documents:	**Description:**
`Elts.xml`	Describes chemical elements in the periodic table.
`w3c1.xml`	W3C specification documentation
`w3c2.xml`	W3C specification documentation
`Mondial-3.0.xml`	World geographic database integrated from the CIA World Factbook, the International Atlas, and the TERRA database among other sources. From FLORID-Mondial case study
`Partsupp.xml`	Part/Supplier relationship. TPC-H Benchmark, 10 MB version, in XML form. Converted to XML by Zack Ives. From Transaction Processing Performance Council (TPC).
`Orders.xml`	Orders. TPC-H Benchmark, 10 MB version, in XML form. Converted to XML by Zack Ives. From Transaction Processing Performance Council (TPC).
`xCRL.xml`	XML files using the Extensible Customer Representation Language format (xCRL) on customer relationship management
`Votable2.xml`	File created in the VOTABLE fML format defined Xor the exchange of data.
`Nasa.xml`	Datasets converted from legacy flat-file format into XML and made available to the public. From GSFC/NASA XML Project
`Lineitem.xml`	Line items. TPC-H Benchmark, 10 MB version, in XML form. Converted to XML by Zack Ives. From Transaction Processing Performance Council (TPC).
`XPATH.xml`	Is not in [73], but uses the LocusXML schema to represent geospatial information in an XML format, it stores annotated human genomic data.
`Treebank_e.xml`	English sentences, tagged with parts of speech. The text nodes have been partially encrypted because they are copyrighted text from the Wall Street Journal. This document has a deep recursive structure. University of Pennsylvania Treebank project.
`SwissProt.xml`	SWISS-PROT is a curated protein sequence database, which strives to provide a high level of annotations (such as the description of the function of a protein, its domains structure, post-translational modifications, variants, etc.), a minimal level of redundancy and high level of integration with other databases. From ExPASy - SWISS-PROT and TrEMBL.
`DBLP.xml`	The DBLP server provides bibliographic information on major computer science journals and proceedings. DBLP stands for Digital Bibliography Library Project. From the DBLP Homepage
`XCDNA.xml`	A cDNA library of a collection of cloned fragments converted into an XML form.

Appendix B

RSDic Library

We are using the C++ library in order to support `RANK` and `SELECT` efficiently, which is called RSDic. We use the RSDic [53, 62] as follows:

- RSDic Builder to push back the **0** or **1** bits(Fill the Bit-Vector with **0** and **1**, based on the way of encode the numbers)

```
RSDicBuilder rsdb;   RSDic rsd;   rsdb.PushBack(0); // To Push
back 0 rsdb.PushBack(1); // To Push back 1 . . rsdb.Build(rsd);
// Build the Bit-Vector
```

- RSDic retrieval to return the result of `RANK` and `SELECT` operations

```
rsd.Select( n, 1) // return the position of (i+1)-th one ,
n >= 0  rsd.Select( n, 0) // return the position of (i+1)-th
zero, , n >= 0   rsd.Rank ( n, 1) // return the number of 1s
between 0 and n   rsd.Rank ( n, 0) // return the number of 0s
between 0 and n
```

Appendix C

DOM methods supported by DAGDOM

Returns	**Method**	**DOM Level**	**SiXDOM component supported**	**DAGDOM, component supported**
Node	appendChild (Node newChild)	1	X	X
Node	cloneNode(boolean deep)	1	X	X
NamedNodeMap	getAttributes()	1	✓	✓
NodeList	getChildNodes()	1	✓	✓
Node	getFirstChild()	1	✓	✓
Node	getLastChild()	1	✓	✓
String	getLocalName()	2	✓	✓
String	getNamespaceURI()	2	✓	✓
Node	getNextSibling()	1	✓	✓
String	getNodeName()	1	✓	✓
Short	getNodeType()	1	✓	✓
String	getNodeValue()	1	✓	*
Document	getOwnerDocument()	1	✓	✓
Node	getParentNode()	1	✓	✓
String	getPrefix()	2	✓	✓
Node	getPreviousSibling()	1	✓	✓
Boolean	hasAttributes()	1	✓	✓
Boolean	hasChildNodes()	1	✓	✓
Node	insertBefore(Node newChild, Node refChild)	1	X	X
Boolean	isSupported(String feature, String version)	2	✓	✓
Void	normalize()	2	X	X
Node	removeChild(Node oldChild)	1	X	X
Node	replaceChild(Node newChild, Node oldChild)	1	X	X
Void	setNodeValue(String nodeValue)	1	X	X
Void	setPrefix(String prefix)	2	X	X
Short	compareTreePosition(Node other)	3	✓	✓
String	getTextContent() - missing minority nodes	3	✓	X
Void	isSameNode(Node other)	3	X	X
String	lookupPrefix(String uri, bool usedefault)	3	X	X

Bibliography

[1] O'Neil Delpratt, Rajeev Raman, and Naila Rahman. Engineering succinct DOM. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 49–60. ACM, 2008.

[2] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 141–152. VLDB Endowment, 2003.

[3] DOM. URL `http://www.w3schools.com/xml/xml_dom.asp`.

[4] W3C DOM API documentation,2004. URL `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`.

[5] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. In *ACM SIGMOD Record*, volume 29, pages 153–164. ACM, 2000.

[6] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S Muthukrishnan. Compressing and searching XML data via two zips. In *Proceedings of the 15th international conference on World Wide Web*, pages 751–760. ACM, 2006.

[7] O Delpratt. *Space efficient in-memory representation of XML documents.* PhD thesis, 2009.

[8] Naila Rahman, Rajeev Raman, et al. Engineering the LOUDS succinct tree representation. In *Experimental Algorithms*, pages 134–145. Springer, 2006.

[9] Repository.,U.X. URL `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`.

[10] W3c XML specification. URL `http://www.w3.org/TR/REC-xml/`.

[11] URL `http://www.ibm.com/developerworks/xml/tutorials/xmlintro/section2.html`.

[12] URL `http://pic.dhe.ibm.com/infocenter/iseries/v6r1m0/index.jsp?topic=/rzamj/rzamjintrouses.htm`.

[13] Web-Services Description Language. URL `http://www.w3.org/TR/wsdl`.

[14] Simple Object Access Protocol. URL `http://en.wikipedia.org/wiki/SOAP`.

[15] Universal Description Discovery and Integration. URL `http://en.wikipedia.org/wiki/Universal_Description_Discovery_and_Integration`.

[16] Votable documentation. URL `http://www.us-vo.org/`.

[17] Medline. URL `http://www.nlm.nih.gov/mesh/gcmdoc2004.html`.

[18] SAX Parser. URL `http://www.saxproject.org/`.

[19] XSLT. URL `http://www.w3.org/TR/xslt`.

[20] XQuery. URL `http://www.w3schools.com/xquery/default.asp`.

[21] XPath. URL `http://www.w3schools.com/xpath/`.

[22] Zorba: The XQuery Proccessor. URL `http://www.zorba-xquery.com/`.

[23] Xerces C++ Parser. URL `http://xerces.apache.org/xerces-c/`.

[24] Document type definition. URL `http://www.w3schools.com/dtd/dtd_intro.asp`.

[25] XML validator. URL `http://www.w3schools.com/xml/xml_validator.asp`.

[26] Fangju Wang, Jing Li, and Hooman Homayounfar. A space efficient XML DOM parser. *Data & Knowledge Engineering*, 60(1):185–207, 2007.

[27] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 479–490. ACM, 2006.

[28] Michael Kay. Optimization in XSLT and XQuery. In *a conference on XML*, page 29. Citeseer, 2006.

[29] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Efficient memory representation of XML documents. In *Database Programming Languages*, pages 199–216. Springer, 2005.

[30] Treewalker. URL `http://www.w3.org/TR/2000/REC-DOM-Level-2-Traversal-Range-20001113/traversal.html#TreeWalker`.

[31] Richard F Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.

[32] Dong Kyue Kim, Joong Chae Na, Ji Eun Kim, and Kunsoo Park. Efficient implementation of rank and select functions for succinct representation. In *Experimental and Efficient Algorithms*, pages 315–327. Springer, 2005.

[33] Dinesh P Mehta. *Handbook of data structures and applications*. CRC Press, 2004.

[34] Charles E Leiserson, RL Rivest, and C Stein. Introduction to algorithms, 1990.

[35] Albrecht Schmidt, Martin Kersten, Menzo Windhouwer, and Florian Waas. Efficient relational storage and retrieval of XML documents. In *The World Wide Web and Databases*, pages 137–150. Springer, 2001.

[36] bzip2. URL `http://www.bzip.org/`.

[37] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[38] Andrei Arion, Angela Bonifati, Ioana Manolescu, and Andrea Pugliese. XQueC: A query-conscious compressed XML database. *ACM Transactions on Internet Technology (TOIT)*, 7(2):10, 2007.

[39] Peter Buneman, Byron Choi, Wenfei Fan, Robert Hutchison, Robert Mann, and Stratis D Viglas. Vectorizing and querying large XML repositories. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 261–272. IEEE, 2005.

[40] James Cheney. Tradeoffs in XML database compression. In *Data Compression Conference, 2006. DCC 2006. Proceedings*, pages 392–401. IEEE, 2006.

[41] Markus Frick, Martin Grohe, and Christoph Koch. Query evaluation on compressed trees. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, pages 188–197. IEEE, 2003.

[42] Jun-Ki Min, Myung-Jae Park, and Chin-Wan Chung. A compressor for effective archiving, retrieval, and updating of XML documents. *ACM Transactions on Internet Technology (TOIT)*, 6(3):223–258, 2006.

[43] Wilfred Ng, Wai-Yeung Lam, Peter T Wood, and Mark Levene. XCQ: A queriable XML compression system. *Knowledge and Information Systems*, 10(4): 421–452, 2006.

[44] Pankaj M Tolani and Jayant R Haritsa. XGRIND: A query-friendly XML compressor. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 225–234. IEEE, 2002.

[45] Ning Zhang, Varun Kacholia, and M Tamer Ozsu. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 54–65. IEEE, 2004.

[46] Mathias Neumüller and John N Wilson. Improving XML processing using adapted data structures. In *Web, Web-Services, and Database Systems*, pages 206–220. Springer, 2003.

[47] Crimson DOM implementation. URL `http://xml.apache.org/crimson/`.

[48] Peter Buneman, Byron Choi, Wenfei Fan, Robert Hutchison, Robert Mann, and Stratis D Viglas. Vectorizing and querying large XML repositories. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 261–272. IEEE, 2005.

[49] Galax XQuery implementation. URL `http://www.galaxquery.org/`.

[50] Saxon. URL `http://saxon.sourceforge.net/`.

[51] Naila Rahman, Rajeev Raman, et al. Engineering the LOUDS succinct tree representation. In *Experimental Algorithms*, pages 134–145. Springer, 2006.

[52] Naila Rahman, Rajeev Raman, et al. Compressed prefix sums. In *SOFSEM 2007: Theory and Practice of Computer Science*, pages 235–247. Springer, 2007.

[53] Rsdic-Compressed Rank Select Dictionary. URL `https://code.google.com/p/rsdic/`.

[54] XMark- XML Benchmark Project. URL `http://www.xml-benchmark.org/`.

[55] Roberto Grossi and Giuseppe Ottaviano. The wavelet trie: Maintaining an indexed sequence of strings in compressed space. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 203–214. ACM, 2012.

[56] Gzip. URL `http://www.gzip.org/`.

[57] Minimal Perfect Hashing Library, . URL `http://cmph.sourceforge.net/`.

[58] Minimal Perfect Hashing, . URL `http://stevehanov.ca/blog/index.php?id=119`.

[59] Richard J Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23(1):17–19, 1980.

[60] Edward A Fox, Lenwood S Heath, Qi Fan Chen, and Amjad M Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, 1992.

[61] Philip Bille, Gad M Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 373–389. SIAM, 2011.

[62] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *Experimental Algorithms*, pages 295–306. Springer, 2012.